The Pennsylvania State University
The Graduate School

EFFICIENT SERVICE DEPLOYMENT ON PUBLIC CLOUD: A COST, PERFORMANCE, AND SECURITY PERSPECTIVE

A Dissertation in
Computer Science and Engineering
by
Ataollah Fatahi Baarzi

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2021

The dissertation of Ataollah Fatahi Baarzi was reviewed and approved by the following:

George Kesidis
Professor of Electrical Engineering and Computer Science
Dissertation Advisor
Chair of Committee

Mahmut Kandemir
Professor of Computer Science and Engineering

Timothy Zhu
Assistant Professor of Computer Science and Engineering

Uday V. Shanbhag
Professor of Industrial and Manufacturing Engineering

Chitaranjan Das
Department Head of Computer Science and Engineering

Abstract

As internet services are becoming more and more popular, business owners need more IT resources in order to supply the demand from their users. However, the cost of maintaining and operating the IT resources is also increasing. In order to focus on the business, rather than managing the private IT resources, businesses (a.k.a tenants) have started to move to the public cloud [1]. The public cloud providers provide virtually unlimited amounts of resources in a variety of types including Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), and more recently Function as a Service (FaaS) (i.e. serverless computing). Tenants might choose one or more than one type of resource offerings to operate their services on the public cloud. The success of the tenants' business is profoundly impacted by how efficient their services are. We argue that the efficiency itself can be decoupled into three main aspects: cost, performance, and security. In this thesis, we propose, develop, and implement mechanisms and systems to achieve cost and performance efficiency while operating on the public cloud and mechanisms to develop attack-resilient cloud-deployed services which are subject to application-layer DDoS attacks.

In the first part of this thesis (chapter 3) we describe our system *BurScale*. BurScale is an autoscale system that exploits the cheap price of burstable instances in order to minimize the cost of virtual machine resource (i.e. IaaS) provisioning in the public cloud. *BurScale* uses the results from queueing theory known as "square root staffing rule" to decouple the cluster of virtual machines into two parts: regular VM instances and burstable VM instances. By combining burstable and regular instances, BurScale is able to save in cost by 50% for both stateless and stateful applications.

In the second part of this thesis (chapter 4) we present *SHOWAR* to improve the efficiency of deploying microservices on public cloud through right-sizing and efficient scheduling of the containers. *SHOWAR* consists of three major components: a vertical autoscaler, a horizontal autoscaler, and a scheduling affinity (and anti-affinity) rule generator. For vertical autoscaling, *SHOWAR* utilizes the empirical variance in the resource usage of containers to determine the size (e.g. number of CPUs and Memory size) of each container. *SHOWAR* uses results from control theory for horizontal autoscaling of microservices. Finally, using the resource usage correlation between different microservices, it generates affinity (and anti-affinity) rules for the scheduler to better schedule the microservices. *SHOWAR* is able to save in cost by 22% compared to the state of the art autoscalers for microservices.

In the third part of this thesis (chapter 5), we present our mechanisms and methods for detecting and defending application-layer DDoS attacks on microservices. We leverage the capabilities of Kubernetes, the state of the art container orchestrator tool, to detect and defend the attacks against cloud-deployed microservices. Our experimental evaluations show that our mechanisms can efficiently detect and isolate the attacks and reduce the impact of the attack on the legitimate users by 3×.

Finally, in the last part (chapter 6), we advocate for a multi-cloud serverless model where this model aggregates multiple cloud providers' services to achieve the best cost and performance for the FaaS workloads using a virtual serverless provider (VSP). We first discuss the merits of such a model and then present the viability of a multi-cloud serverless platform that seeks for cost and performance efficiency. Our results from evaluating an initial prototype of a VSP show that VSPs can potentially save more than 50% in costs for deploying FaaS workloads.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Many people have played an important role in supporting me succeed in my PhD program. First and foremost, I would like to thank my advisor Professor George Kesidis, for all his help and support. I will always be indebted to George's trust in me which boosted my confidence and made me courageous enough to explore my own research ideas well beyond the scope of his active research projects. George has far exceeded my expectations of an advisor. I would also like to thank Professor Bhuvan Urgaonkar for supporting and co-advising me in the first two years of my PhD. During my time in graduate school, I enjoyed working with Professor Timothy Zhu both as a research assistant and a teaching assistant. I have learned tremendously from Timmy which helped me a lot in becoming a better researcher. I would like to thank the rest of my committee members Professor Mahmut Kandemir and Professor Uday Shanbhag for their time and valuable feedback. I am grateful to Professor Mohammad Shahrad for always being open for discussion and providing me feedback. Our collaboration with Mohammad was an amazing experience for me and I learned a lot from him.

I have interned three times during my PhD which I would like to thank my mentors. I am thankful to my mentors Jon Currey at HashiCorp and Ashvin Agrawal and Fotis Psallidas at Microsoft Gray Systems Lab. I am grateful to my mentor Pei-Lun Liao at LinkedIn which working with him was by far one of my best experiences I had during my PhD program.

To me, the most of important part of the PhD progam was starting and getting into it. For that, I am forever thankful to my undergraduate advisers and mentors at Sharif University of Technology. I would like to thank Professor Hamid Sarbazi-Azad and Professor Hossein Asadi. I have learned a lot from my mentors Dr. Arash Tavakkol and Mojtaba Tarihi. I worked on my B.Sc. thesis with my amazing friend and teammate Mahdi Khosravi. Working with Mahdi was full of learnings and fun which I am forever grateful for it. If it was not their help and support, starting a PhD would have not been possible for me.

I am always fortunate to have friends around me who have greatly supported me. I would like to thank my friend and Gym buddy (dedication: 10/10) Morteza Ramezani who is always the first person I reach out to whenever I have a question. Morteza has been helping me and answering my questions from day 1 that I arrived at State College until literally today that I am writing this thesis. I am thankful to my longtime friend Mahdi Shafiei for always being there for me and cheering me up at hard times. Sina

Gharebaghi, Foad Jafarinejad, Seyed Reza Fatahi (a single t suffices!), Mohammad Beigi, and Lexiang Huang have been a second family to me. I'd like to thank them all for their friendship over the years and the joy they've brought into my life.

I owe a great deal to my family for their continual love and support. Without their sacrifices, I would have not been where I am today. I owe all of my accomplishments to my lovely parents Ghasem and MahZari, my sister Gol, and my brothers Hashem, Hatam, Hafez, MohammadMahdi, and Kazem. I'm eternally grateful to them for their support and care throughout my life.

This thesis was supported in part by NSF CCF grant 2028929, NSF CNS grant 2122155, NSF CNS 1717571, Defense Advanced Research Projects Agency (DARPA) Extreme DDoS Defense (XD3) contract no. HR0011-16-C-0055, a Cisco Systems URP gift, and an AWS research credit award. Opinions, findings, conclusions, and recommendations expressed in these materials are those of the author and do not necessarily reflect the views of NSF, DARPA, Amazon, or Cisco.

Dedication

Dedicated to all of the 176 beautiful souls who perished in the flight PS752 on January 8th, 2020.

Chapter 1
Introduction

1.1 Motivation and Objectives

As the internet services are becoming more popular, the business owners need more IT resources in order to supply the demand from their users. However, the cost of maintaining and operating the IT resources is also increasing. In order to focus on the business, rather than managing the private IT resources, businesses have started to move to the public cloud [1]. One of the appealing features of public cloud that attracts businesses to move to the public cloud is resource elasticity. The public cloud providers provide virtually unlimited IT resources. Therefore, the service owner (i.e. the business) can provision enough resources at any time for the demand from its users.

The cloud providers offer the IT resources in variety of forms including IaaS, PaaS, SaaS, and FaaS (see chapter 2 for more details). Depending on the needs and business decision, the tenants choose to operate their services on any of these forms or a combination of them.

As the success of the businesses depends on their cloud-deployed services, it is critical for them to operate their services as efficient as possible. Therefore, we argue that archiving efficiency in public cloud is of a great importance.

While the meaning of efficiency can vary from business to business, we identify three major aspects of efficiency in the public cloud that applies to almost any type of business. The three main aspects of efficiency are **cost**, **performance**, and **security**. We discuss each of these aspects:

- **Cost:** One of the main reasons that businesses are moving to the public cloud is the cost of owning and maintaining IT resources and infrastructure. In the other hand, the pay-as-you-go pricing model of the public attracts businesses to move to the

public cloud. Even though this pricing model might appear to be cheap compared to maintaining private infrastructure, the overall cost of public cloud could form a major part of the business's costs [3]. Therefore, in order to optimize the costs, the resource usages has to be optimized. In this thesis, we present systems for cost-effective resource provisioning and service deployment on public cloud (see chapter 3 and chapter 4).

- **Performance:** The success of the businesses depends on the performance of their cloud-deployed service performance. For example, for an online shopping business which is operating in a competitive market, the performance of their user-facing web application has a profound impact on their revenue and hence the success of their business. Here the performance is translated to the request latency that the end users experience. In order to achieve such performance, since the business logic service uses and interacts with other cloud offered services, this usage and interaction has to be optimized. In chapter 6 we advocate for a multi-cloud serverless model in FaaS platforms where the tenant can aggregate the best services from each cloud provider to gain efficient performance and cost. In doing so we make a proposal for a virtual serverless provider (VSP) that aggregates the serverless offerings from multiple providers for achieving better cost and performance.

- **Security:** Operating on public cloud comes with some security concerns and issues. This is mainly because the infrastructure including the VM instances and network resources are managed by the cloud provider and the tenant has less control over them. Examples of security issues include compromising virtual machine by attackers which can lead to bringing down the service. Another example is the distributed denial of service (DDoS) attacks where the attackers try to exhaust the underline resources used by the application to bring down the service. Even though cloud providers implement mechanisms to protect virtual machines and also protect the services from volumetric DDoS attacks using firewalls, application-layer and application specific attacks can still be a major issue. Therefore, tenants need to implement their own attack detection and defense mechanisms beyond what cloud providers implement. In chapter 5 we propose some detection and defense mechanisms to protect microservices from application-layer DDoS attacks.

Note that even though the tenants can address each of these aspects separately, that does not mean these aspects are completely independent. Conversely, these aspects are dependent to each other and it is the responsibility of the tenant to optimize the

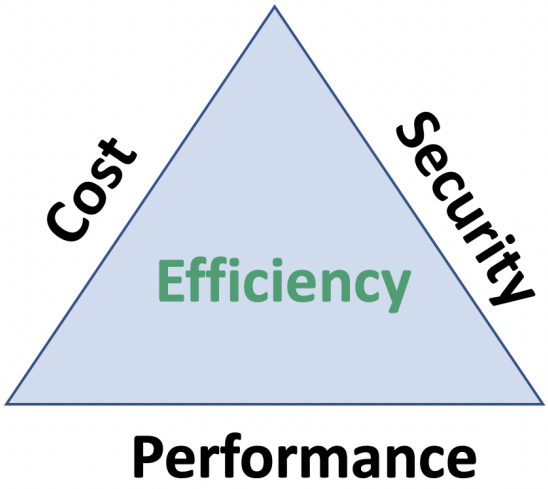

Figure 1.1. The Three Aspects of Efficiency

trade-offs between them. For example, in order to achieve high performance for a service, the more available resources, the more performant the service is. However, by increasing the amount of resources, the cost will be increased as well, per the pay-as-you-go pricing model. In the other hand, in order to protect the services from denial of service attacks, since their target is to increase the resource usage, one of the solutions is to increase the amount of resources and hence an increase in costs. As such it is important to mitigate the effect of DDoS attacks in a cost-effective way while operating on the public cloud.

In order to address these trade-offs, in this thesis we design, implement and present mechanisms, systems, and solutions to achieve efficiency for tenants of public clouds. Our contributions are as follows.

1.2 Contributions

In this proposal, we make the following contributions in respect to each aspect of the efficiency:

- **Cost-Effective Resource Provisioning**

  The amount of workload that a cloud-deployed application receives varies from time to time and it usually follows a diurnal pattern. In order to save in costs of resource provisioning, autoscaling systems change the number of VM instances in

the cluster in response to the change in the workload. BurScale is an autoscaling system which tries to save in resource provisioning costs beyond what a typically autoscaling system does. The idea is to replace some of the regular VM instances with cheaper VM instances called burstable instances. BurScale leverages the results from queueing theory known as *square root staffing rule* to determine how many burstable instances are needed. By using burstable instances, BurScale is able to save up to 50% in costs while guarantying the same performance as using all regular instances.

We evaluate BurScale for both stateless and stateful applications and show its effectiveness through extensive evaluation:

- Stateless Application: We deploy WikiMedia web application which is used to operate Wikipedia website on a cluster on AWS EC2 and use BurScale as an autoscaler for the cluster. Our evaluation results show that by combining burstable and regular VM instances, BurScale is able to save up to 46% in costs for such stateless user-facing web application.
- Stateful Application: We deploy a distributed memcached in memory key-value store as a stateful application and use BurScale as the autoscaler. By placing the “hot" keys (aka highly demanded keys) on burstable instances, BurScale is able to save 50% in costs compared to a baseline which autoscaler scale up the cluster using only regular instances.

- **Right-Sizing and Efficient Scheduling of Microservices**

  One of the challenges in deploying microservices is finding the optimal amount of resources (i.e. size) and the number of instances (i.e. replicas) for each microservice in order to maintain a good performance as well as prevent resource wastage and under-utilization which is not cost-effective.

  Towards addressing those deployment challenges, we present *SHOWAR*, a framework that configures the resources by determining the number of replicas (horizontal scaling) and the amount of CPU and Memory for each microservice (vertical scaling). For vertical scaling, *SHOWAR* uses empirical variance in the historical resource usage to find the optimal size and mitigate resource wastage. For horizontal scaling, *SHOWAR* uses basic ideas from control theory along with kernel level performance metrics. Additionally, once the size for each microservice is found, *SHOWAR* bridges the gap between optimal resource allocation and scheduling

by generating affinity rules (i.e. hints) for the scheduler to further improve the performance. Our experiments, using a variety of microservice applications and real-world workloads, show that, compared to the state-of-the-art autoscaling and scheduling systems, *SHOWAR* on average improves the resource allocation by up to 22% while improving the 99th percentile end-to-end user request latency by 20%.

- **Attack-Resilient Microservices**

  Cloud-deployed applications benefit from large scale cloud firewalls and content delivery networks (CDNs) in response to volumetric distributed denial of service (DDoS) attacks. However, a new class of DDoS attacks known as application-layer DDoS attacks are emerged recently. Unlike traditional volumetric DDoS attacks, these attacks are at low volume and target the application itself rather than the networking and connection resources. The goal of such attacks is to exhaust the resources used by applications to prevent them from serving legitimate requests. As more businesses are moving to public clouds, more application-layer DDoS attacks are happening and hence solutions are required. In this we work, we propose effective detection and defense mechanism in response to application-layer DDoS attacks. The detection mechanism relies on profiled resource usage data and using first-order statistic analysis tools, it detects the abnormal resource usages of the application and hence it deems the application as attacked. The response mechanism in the other hand, quarantines the attacked application on a reserved node in the cluster called quarantine node. By quarantining and fissioning the users of the attacked application, the attacker is identified and isolated.

  We evaluate our mechanisms on kubernetes the state of the art container orchestrator tool for microservices using a computer vision application. The experimental results show that our mechanisms detect and defense the attack efficiently. Furthermore, the average latency that legitimate users experience is 3× improved compared to a situation where there is not detection and defense mechanism in place.

- **Multi-Cloud Serverless Model**

  Today, all of the major cloud providers offer serverless computing through function-as-a-service offerings. However, they come with different cost and performance characteristics. To exploit these variances and achieve more performance and cost-effective serverless deployment, we envision virtual serverless providers (VSPs) to aggregate serverless offerings. In doing so, VSPs allow developers (and businesses)

to get rid of vendor lock-in problems and exploit pricing and performance variation across providers by adaptively utilizing the best provider at each time, forcing the providers to compete to offer cheaper and superior services. We discuss the merits of a VSP and show that serverless systems are well-suited to cross-provider aggregation, compared to virtual machines. We propose a VSP system architecture and implement an initial version. Our preliminary results show that a VSP can improve the maximum sustained throughput by $1.2x$ to $4.2x$, reduces SLO violations by 98.8% , and improves the total invocations' costs by 54%.

1.3 Organization

The rest of this thesis is organized as follows. In chapter 2 we present some background on cloud computing and cloud offerings. In chapter 3 we describe BurScale [4], our work on cost-effective resource provisioning in the public cloud. We present *SHOWAR* for right-sizing and efficient scheduling of microservices in chapter 4. We explain our detection and defense mechanisms for microservices against application-layer DDoS attacks in chapter 5. In chapter 6 we advocate for a multi-cloud serverless application model. Finally, in chapter 7 we conclude the thesis and explain the future work.

Chapter 2
Background

As motivated and explained in chapter 1, in this thesis we design and develop systems and mechanisms to achieve efficiency in public cloud for a tenant. We thus present a background on the public cloud resource offerings and also the ways that tenants deploy their services on the public cloud. This background shall give us a better understanding of the environment in which we seek for the efficiency.

2.1 Cloud Resource Offerings

The public cloud providers usually provide the IT resources in one of the following forms:

- **Infrastructure as a Service (IaaS):** The granularity of resources in IaaS is a virtual machine. The tenant can choose from a variety of VM options in terms of number of CPUs, the size of memory, the type and size of storage, etc. As an example, at the time of writing this proposal, AWS EC2 offer more than 100 types of virtual machine instances. These types are usually categorized based on the workload and the needs of the tenants. Categories include general purpose, compute optimized, memory optimized, GPU instances, spot instances, and burstable instances. The cost of each virtual machine directly depends on its size (i.e. number of CPUs, memory size, etc) and the charging model for almost all of these types is pay-as-you go. The rates for the cost of each virtual machine instances are $ per-hour usage and it is at minute granularity. In order to deploy their services, tenants provision enough number of virtual machines to from a cluster in one or more than one geographical zones. In order to save save more in costs, tenants deploy atuoscaler systems along with their clusters to change the size of cluster in response to the changes in the incoming workload. To protect

virtual machines from attacks such as DDoS attacks, cloud providers offer “virtual private cloud" (VPC) and distributed “content delivery networks" (CDN) which tenants can utilize these systems to futher protect their services.

- **Software as a Service (SaaS):** In order to operate an application which performs the business logic, some other services are required. Examples include caches, databases, queues, load balancers, etc. As these services are prevalent for most of the cloud-deployed applications, cloud providers offer such services in a managed way so that other applications can use them. In this model, the tenants need not to implement and maintain these services rather they rely on the managed services offered by the cloud provider. The cloud provider in turn, manages these services and guaranties on performance availability of the service as part of service level objectives (SLOs) and service level agreements (SLAs).

- **Platform as a Service (PaaS):** As explained above, tenants use IaaS to provision clusters of virtual machines to deploy their workloads and services. In order to deploy a service on a cluster virtual machines, some other services are required to deploy and operate the service. For example, if the service is a containerized application, a container orchestrator tool (i.e. a platform) such as kubernetes on top of the cluster of virtual machines is needed. As another example, if the service is a data analytic application, a data analytic framework (i.e. platform) such as Apache Spark is needed. Therefore, in order to operate the final service, these additional platforms are needed which tenants have to install and deploy them on the cluster of virtual machines. Same as SaaS, cloud providers offer a managed version of this frameworks so that tenants can use them to deploy their services. Examples include AWS EKS which is a managed kubernetes cluster that enables tenants to deploy their containerized applications on the cluster of virtual machines.

- **Function as a Service (FaaS):** Function as a Service (or serverless computing) is one of the fastest growing paradigms in the cloud industry. It aims to decouple infrastructure management from application development. Under the serverless paradigm, the application developer does not worry about provisioning and specifying the number and type of virtual machines (VMs) that it requires, which is particularly tricky in the shadow of varying demand. Instead, the provider conducts automatic and scalable provisioning. At the core of serverless computing lies the Function-as-a-Service (FaaS) model where functions form the computational

building blocks. Functions are typically stateless[1], and the application logic is performed in an event-driven style. Deployment of an application then becomes as easy as coding these functional building blocks and connecting them to each other, external triggering events, and/or storage services. Today, all major cloud providers offer FaaS: AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions. However, FaaS is still in its infancy, where there is no consensus on optimal pricing models, the best deployment strategies, the right mix of IaaS (infrastructure-as-a-service) vs. FaaS that developers should use, or even the applications that benefit from FaaS the most.

2.2 Service Deployment on the Public Cloud

Depending on technical and business factors, tenants may choose one or more than one type of cloud resource offerings explained in section 2.1 to deploy their services on the public cloud. Nonetheless, the most common approach is that for different parts of business logic, different resource offering is used. For example, the core logic is usually developed and deployed as containerized applications (i.e. microservices) on clusters of virtual machines. Due to the pricing model of serverless offerings, the sporadic workloads (those who happen once in a long time) are deployed as cloud functions. Finally, managed services such as queues or databases may also be used to fulfill the core application needs.

As the main part of the business logic is developed as containerized applications, here, we present a brief background of microservices. Microservices are containerized applications which the entire service is decoupled into smaller logical parts, each part is called a microservice. Microservices are loosely couple together to serve a request collectively. The communication between microservices is done by either HTTP calls or RPC calls. An example of an application with microservices architecture is the *Social Network* application from the *DeathStarBench* benchmark suite depicted in Figure 2.1.

Each microservice consists of one or more than one container. In order to orchestrate and schedule these containers on a cluster of virtual machines, a container orchestrator tool is used. The state of the art container orchestrator tool is Kubernetes [6]. In kubernetes terminology, each microservice consists of one or more than one “Pod". A Pod is basically an abstraction layer around a a container and it is the deployment unit in kubernetes (see Figure 2.2).

In order to manage the cluster, one of the nodes in the cluster is reserved as master

[1] A certain offering, called Azure Durable Functions, provides stateful functions [5].

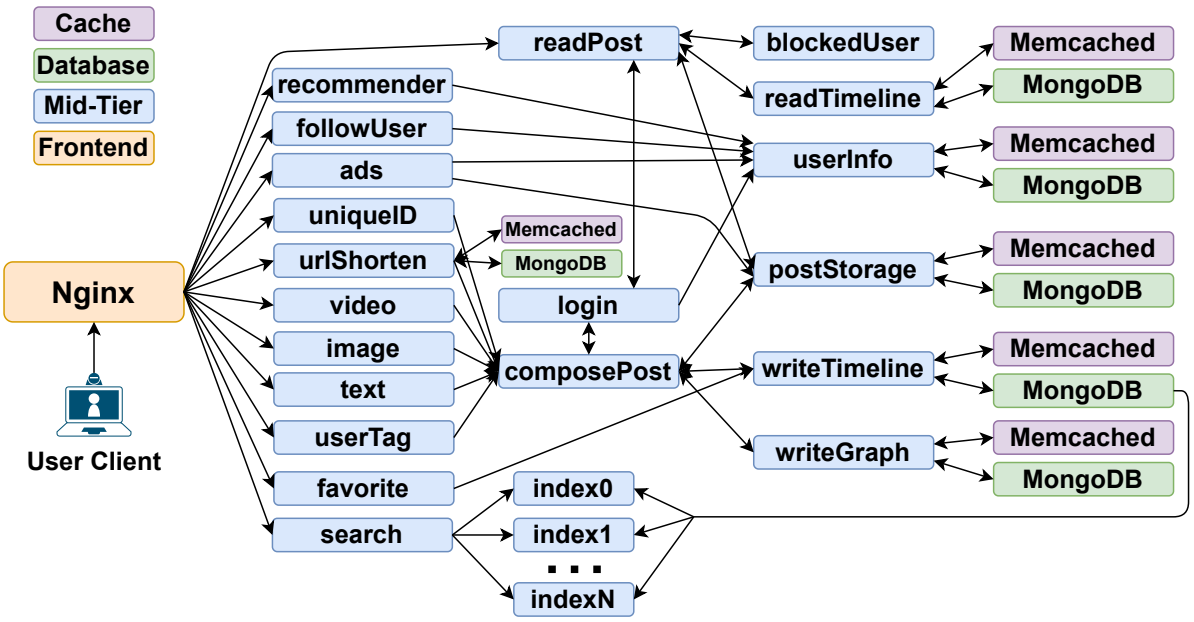

Figure 2.1. Social Network application as an example of microservices architecture

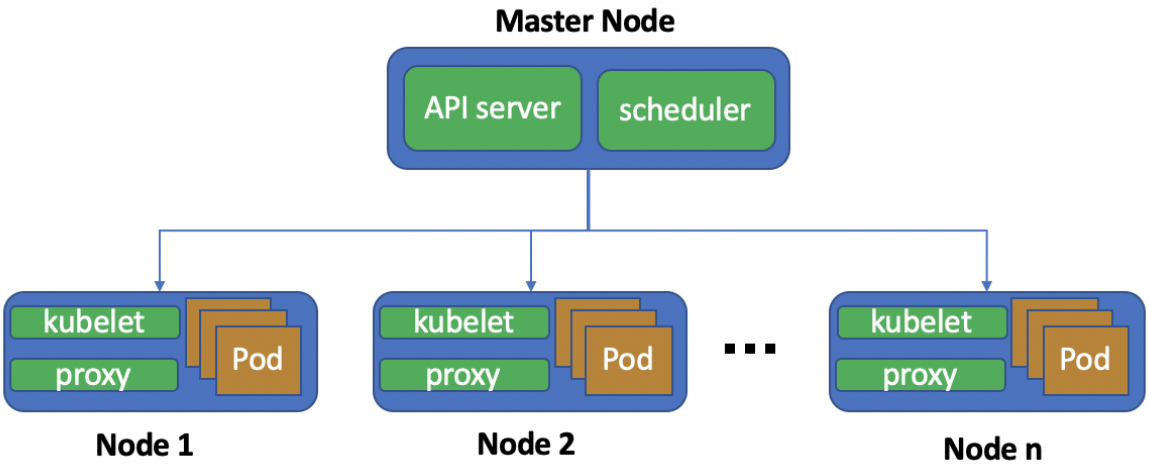

Figure 2.2. Microservice deployment using kubernetes on cluster of virtual machines.

node. At the heart of any container orchestrator tool is its scheduler and its API server. The scheduler basically schedules (i.e. places) the Pods (i.e. containers) on the nodes withing the cluster. The API server is used to deploy the application on the cluster. The nodes also use a daemon called “kubelet" to communicate with the master node. In order to enable the communication between microservices on the cluster, a proxy is running on each node which performs this communication on behalf of the Pods.

Chapter 3
Cost-Effective Resource Provisioning

3.1 Introduction

One of the key appeals of operating in the public cloud is autoscaling, the ability to elastically scale cluster capacity to match the workload's dynamically evolving resource needs. Autoscaling can help a cloud customer ("tenant") lower its costs compared to provisioning resources based on the peak needs at all times, which would be the case if the tenant were to use privately owned IT infrastructure. Many workloads exhibit predictable workload variations, e.g., time-of-day effects or seasonal patterns. Such workloads stand to benefit from an autoscaling facility. Autoscaling can also help a tenant respond effectively to less predictable phenomena, e.g., "flash crowds", particularly if it has high agility in the sense of allowing addition of new capacity (or removal of existing capacity) quickly.

Existing autoscaling solutions tend to employ regular instances whose resource capacities remain *fixed* over time, e.g., on-demand or reserved instances in Amazon Web Services (AWS). However, for interactive workloads, these fixed capacities are not utilized entirely. Fundamentally, provisioned capacity must be higher than used capacity to have enough spare capacity for handling transient periods of queueing and workload spikes. As these effects occur at fine time scales, there is not enough time for autoscaling, so the resources must be instantly available. We find that the recently introduced *burstable* instances offer a cheaper alternative to regular instances for procuring such spare capacity.

Burstable instances are virtual machines whose CPU (and sometimes network bandwidth) capacity is a mere fraction of that of a regular instance on average. However, they have the ability to occasionally burst up to a peak rate comparable to that of the

regular instance in a manner regulated by a token-bucket mechanism. That is, burstable instances accrue CPU credits during periods of CPU utilization lower than their long-term average ("base") allocated capacity. These credits are in turn spent during periods when CPU utilization exceeds the base capacity. Burstable instances are a lot cheaper than *comparable* on-demand (regular) instances, where by "comparable" we mean regular instances whose CPU rate equals the peak rate available to the burstable instance. For example, as of May 2019, an Amazon EC2 $m5.large$ instance (with 2 vCPUs and 8 GB of memory) was almost 10 times more expensive than a $t3.micro$ instance (with 2 vCPUs each at 10% sustained rate and the same peak rate and 1 GB of memory). Note that these savings come with the caveat of low memory usage. Despite this, we show how our techniques can apply both to a stateless web application and a stateful caching layer.

Our interest lies in devising an enhancement for autoscaling that combines burstable instances with regular instances for improving costs over state-of-the-art autoscaling. We use the term autoscaling to mean elastically provisioning enough resources to meet a workload's performance needs as demand changes over time. Our solution, BurScale, uses existing mechanisms to add and remove resources from the system as demand changes over time. While we do not innovate in all aspects of autoscaling, we focus on the core aspects of autoscaling pertaining to cost efficiency, which primarily involves resource provisioning. Our work considers two key research questions: (i) how many of the instances should be burstable, and (ii) how should requests be load balanced or unbalanced between regular and burstable instances? We address these questions in the context of a stateless web application, a stateful cache, and flash crowd scenarios. The key idea is exchanging a fraction of the regular instances for burstable instances and modifying the load balancer to only utilize burstable instances when there is substantial queueing. Thus, the burstable instances do not exhaust their CPU credits while still being able to mask the effects of transient queueing and the startup delay when autoscaling.

We make three primary research contributions:

- We introduce a new technique for reducing the cost of autoscaling systems by provisioning using a combination of burstable and regular instances. Our technique involves two main parts: (i) determining the number of regular instances that should be replaced with burstable instances[1], and (ii) dynamically adjusting the load balancer to avoid exhausting the burstable instances' CPU credits. We

[1]For simplicity, we assume the peak performance of burstable instances matches that of regular instances so that a burstable instance can replace a regular instance as long as its CPU credits are not exhausted.

implement and evaluate our technique in our prototype BurScale system, which is open-sourced at `https://github.com/PSU-Cloud/BurScale`.

- We show how our technique applies even to stateful caches, where memory is important for performance. As the cost benefit of using burstable instances depends on low memory usage, it is insightful to see how certain caching scenarios can take advantage of burstable instances.
- We adapt our technique for flash crowds and demonstrate how it masks the startup delay when acquiring new resources in response to a flash crowd (i.e., an unexpected traffic surge). By using burstable instances to supply the necessary overprovisioning, we see savings up to 50% compared to only using regular instances.

3.2 Background

In this section, we discuss the features of burstable instances and the context under which we utilize them.

3.2.1 Burstable Instances

In order to monetize their idle resources, public cloud providers like *Google Cloud Engine* [7], *Amazon Web Services* [8, 9], and *Microsoft Azure* [10] offer a variety of instance types that are cheaper on average than the more well-established on-demand or "regular" instances. These include reserved instances, preemptible (e.g., spot) instances, and **burstable instances**. Our interest in this work lies in burstable instances, which are cheap rate-limited instances that can sporadically burst to peak CPU rate for short durations. Specifically, we focus on burstable instances offered by AWS, the cloud provider with the oldest and most diverse burstable offerings. However, our ideas readily apply to offerings from other providers as well.

Currently, Amazon offers burstable instances in the *t2* and *t3* categories, each with 7 different options for resources ranging from 1-8 virtual CPUs (vCPUs) and 0.5-32 GiB RAM. The appeal of burstable instances is in their significantly reduced price. For example, a t3.nano burstable instance has 2 vCPUs and is only 5.4% the cost of a m5.large regular instance with 2 vCPUs. With such a low price, burstable instances also come with their drawbacks. For example, the t3.nano instance is rate-limited to an average of 5% of the CPU capacity and has 6.25% of the memory capacity as compared to the m5.large.

In a sense, a burstable instance represents a fractional CPU that is able to burst to 100% capacity for limited periods of time. Amazon controls this through a token-bucket rate limiter mechanism where a token is called a CPU credit. One CPU credit means the instance can use 100% of one vCPU for 1 minute. Alternatively, an instance can use one CPU credit for running a vCPU at 50% for two minutes or 25% for 4 minutes. Credit usage and CPU utilization are tracked at a millisecond granularity. As long as an instance has CPU credits it can use its whole vCPU capacity, otherwise its usage is limited to the baseline CPU rate for that instance type. Different burstable instance types are configured to earn different rates of CPU credits with higher rates being priced higher. Credits that are not used will accumulate up to 24×*Hourly Credit Earning Rate* credits. Amazon has also provided a mechanism to not run out of CPU credits by purchasing them when the token bucket is empty. However, the cost of CPU credits makes burstable instances more expensive than regular instances if one wanted to use them all the time.

Thus, burstable instances are not a good choice for batch workloads that need maximal utilization all the time. Rather, this dynamism in CPU performance and having a lower price compared to regular instances make burstable instances a good candidate for workloads that use low-moderate levels of CPU most of the time, but sometimes need higher levels of CPU. Thus, applications like micro-services, low-latency interactive applications, small databases, virtual desktops, development, build and stage environments, code repositories, and product prototypes can benefit from burstable instances.

3.2.2 Applications

In this work, we present novel use cases for using burstable instances to reduce cost while maintaining performance goals known as service-level objectives (SLOs). We focus on interactive workloads wherein a server application responds to requests from latency sensitive clients. In section 3.3, we demonstrate how our ideas generalize to two canonical examples of such an application: (i) a replicated web server, and (ii) an in-memory key-value cache.

Web Application. The first type of application we consider is a PHP web application serving a combination of static pages and dynamically-generated content. In our experiments, we use the Wikimedia application, which is the application used for operating the Wikipedia website. We find this application is both CPU and network intensive. So to support a high rate of requests, an application will often be replicated across many independent web servers with a central load balancer that directs each request to one

of the servers. Traditionally, all of these servers would be regular instances, and our work will show that a subset of these can be converted to burstable instances without significant loss of performance once appropriate changes are made to the load balancer.

In-Memory Cache. The second type of application we consider is an in-memory key-value cache, such as Memcached [11]. The key distinction with this type of application is its statefulness, which requires memory. Whereas CPU and network bandwidth can easily be shared between instances across time, memory capacity is not easily shared across time. Since a burstable instance has memory dedicated to it, this limits the number of other instances that a cloud provider can co-locate on the same physical machine. Consequently, a burstable instance with the same allocated memory capacity as a regular instance has nearly the same (only slightly lower) price. Thus, burstable instances prove to be cost-effective replacements of regular instances only in cases where, in addition to CPU/network needs being intermittent, the memory needs are also low.

One might imagine that the cost of memory would preclude the use of burstable instances with memory caches, and this is true in some scenarios. If more instances are needed in a system to increase the total memory capacity of the cache, then burstable instances have very limited benefits in these cases. However, there are cases such as in Microsoft's web cache [12] where a cache is replicated for a higher aggregate CPU or network bandwidth rather than memory capacity. In such scenarios, burstable instances are a perfect fit since they can still provide benefit when only replicating the most popular ("hot") key-value pairs. As items within caches often exhibit skewness in their popularity that can be captured by a heavy-tailed distribution (e.g., Zipfian [13]), the hot keys constitute a relatively small portion of the whole working set, but are used to serve a disproportionately large fraction of the incoming traffic. Our work investigates the issues arising from such a use case – namely the fact that burstable instances are unable to serve all of the requests, and adding a new cache server to the system will incur a warm up time to populate the cache.

3.3 BurScale Design

In this section, we show how BurScale uses burstable instances in combination with regular instances to improve the cost-efficiency of autoscaling in two applications: a web application and an in-memory key-value cache.

3.3.1 Our Autoscaling Approach

A typical autoscaling system monitors and records workload behavior over time and periodically adjusts the number of servers in the cluster to minimize costs while meeting performance goals. It is often composed of two components: (i) a component that estimates future workload properties based on current and previous behavior, and (ii) an application performance model that determines the number of servers, k, needed to satisfy the performance goal based on the estimated workload properties. As the desired number of servers changes over time, the autoscaling system adds or removes servers from the system and updates the cluster configuration accordingly.

Future Workload Predictor. Autoscaling systems are often categorized into predictive [14, 15], reactive [16, 17], and hybrid [18, 19] approaches. Predictive autoscaling systems utilize complex future workload predictors and sometimes need to build in mechanisms for protecting against mispredictions. On the other hand, reactive autoscaling systems are often much simpler and assume the future behavior will closely resemble the current behavior. *The techniques in BurScale can directly apply to any autoscaling system, but for the sake of simplicity in understanding experimental results, we employ a reactive approach where the future behavior is assumed to be the same as the current behavior.* In subsection 3.3.4, we explore how to handle flash crowd where the future behavior is drastically different from the current behavior.

Application Performance Model. The application performance model is used to determine the number of servers, k, needed to maintain low latency for a given request arrival rate, λ, and server service rate, μ. In this work, we consider systems composed of k servers receiving work from a central *Join the Shortest Queue* (*JSQ*) load balancer. Under the JSQ load balancing policy, prior work [20, 21] has shown that the system can be approximated by a $M/M/k$ queueing system, which is defined as a system with k servers processing requests from a central queue where requests arrive according to a Poisson process and request sizes follow an exponential distribution. Under the $M/M/k$ queueing model, we can apply the queueing theory result known as the Square-Root Staffing rule (SR rule) [22], which approximates the minimum number of servers needed to keep the probability of queueing, P_Q, below a user-defined threshold (SLO), α:

Theorem 1. ***Square-Root Staffing Rule (SR Rule)*** *[22] Given an $M/M/k$ queueing system with arrival rate λ and service rate μ, let $R = \frac{\lambda}{\mu}$, and let ${k_\alpha}^*$ denote the least number of servers needed to ensure that the probability of queueing ${P_Q}^{M/M/k} < \alpha$. Then ${k_\alpha}^* \approx R + c\sqrt{R}$ where c is the solution for the equation $\frac{c\Phi(c)}{\phi(c)} = \frac{1-\alpha}{\alpha}$ where $\Phi(.)$ denotes*

the c.d.f. of the standard Normal distribution and $\phi(.)$ denotes its p.d.f.

Here, the c parameter is directly related to the upper bound on the probability of queueing, α, which is in turn related to latency, our target metric. That is, higher values of c result in more instances and consequently reduced queueing and latency. To calculate c, one would perform a binary search on c until the equation in Theorem 1 is approximately[2] satisfied. To approximate a latency SLO via a probability of queueing (P_Q) SLO, one would use the following equation [22]:

$$E[T] = \frac{1}{\lambda}.P_Q.\frac{\rho}{1-\rho} + \frac{1}{\mu} \tag{3.1}$$

where $\rho = \frac{\lambda}{k\mu}$ is the load of the system.

We find that this $M/M/k$ model is reasonable for our workloads, but *we do not claim that it is the best performance model for all scenarios. Rather, it is a reasonable model that we use in BurScale to demonstrate our ideas on using burstable instances, which we believe can extend to other autoscaling models that calculate k differently.*

Preliminary Validation. To confirm that this model applies to our system, we experimentally determine the minimum number of servers needed to meet a latency SLO for different arrival rates (Figure 3.1). We compare this with two theoretical formulas: one using the SR rule with a fixed probability of queueing $P_Q < 10\%$ and the other based on constraining Equation 3.1 to be less than the SLO. As seen in Figure 3.1, the SR rule and Equation 3.1 are both reasonable heuristics for approximating the number of servers needed for maintaining a latency SLO.

For the purposes of this validation experiment, we use a simple CPU intensive PHP script for our application, and we generate requests with exponentially distributed request sizes and Poisson process arrival times. Our later experiments in section 3.5 show that the SR rule is a good approximation for real-world workloads where the arrival process is not a Poisson process and request sizes do not follow an exponential distribution.

3.3.2 Cost-effective Autoscaling

In this section, we develop a new technique for enhancing autoscaling systems by replacing some regular instances with burstable instances. We address two key research questions: (i) how many burstable instances should be used, and (ii) what dispatching policy should be used?

[2]Theorem 1 is an approximation, so the calculation of c need not be exact.

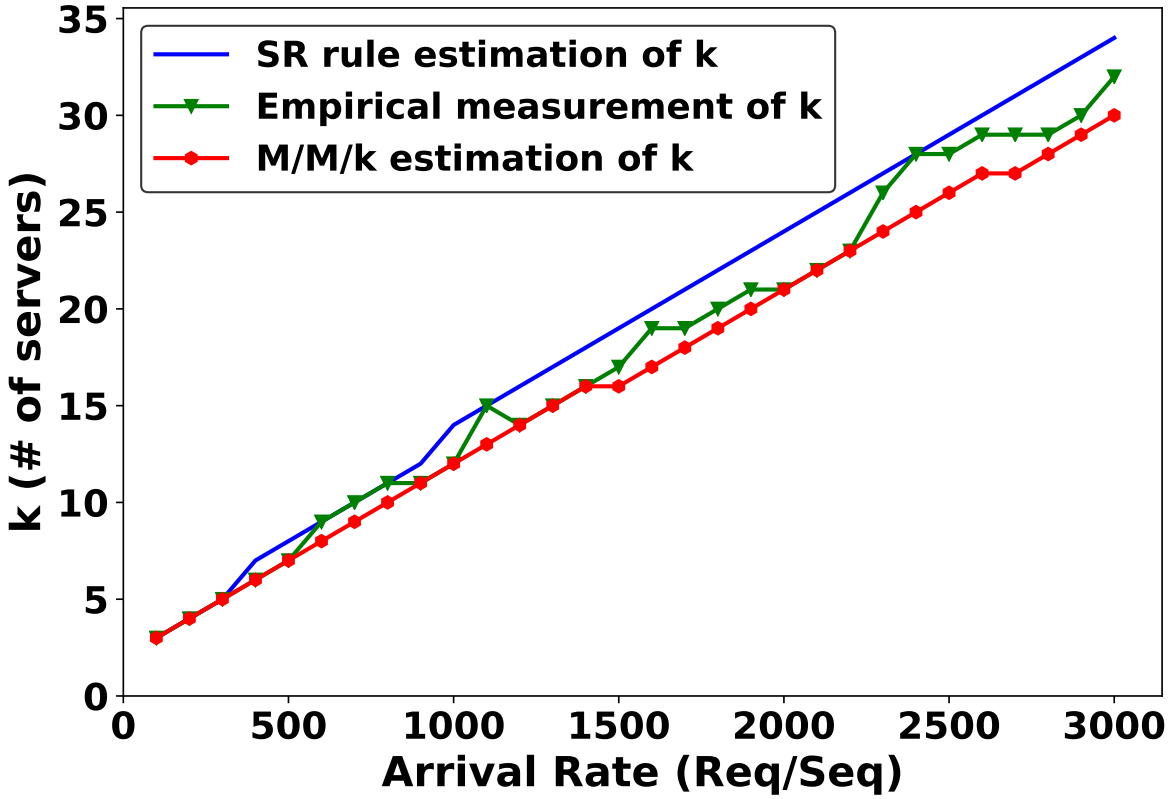

Figure 3.1. Comparing the Square-Root Staffing rule (SR rule) and M/M/k model with empirically determining the number of servers, k, needed to satisfy a latency SLO. Both theoretical models are reasonable approximations for capacity provisioning.

How many burstable instances? As described in subsection 3.3.1, we assume the autoscaling system provides information about the number of instances, k, needed to satisfy the SLO. For our experiments, we use a reactive policy based on the mean latency in a $M/M/k$ model, but our idea extends to other autoscaling policies that are predictive or based on tail latency or any other metric of interest. Thus, our contribution is in determining how many of these k instances should be of the burstable kind. If we use burstable instances too much, then they may run out of CPU credits and be unable to operate at peak performance when needed. For simplicity, we select burstable instances with the same number of vCPUs as our regular instances so that the peak performance of the burstable instances roughly matches that of our regular instances.

The key intuition behind our approach stems from Theorem 1, which suggests $k \approx R + c\sqrt{R}$. Here, the R term indicates the minimum number of instances needed to keep the system stable (i.e., not overloaded with more work than it can handle). With R instances, a system could theoretically handle all the incoming traffic without dropping

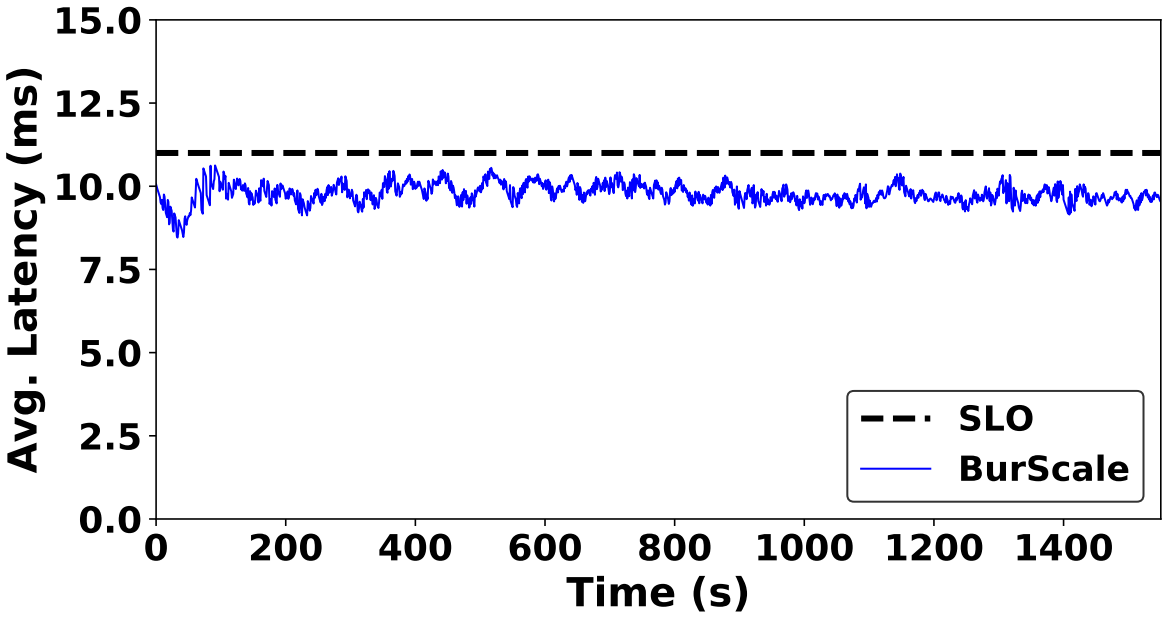

Figure 3.2. Measured latency in our microbenchmark web application stays under the desired SLO when BurScale uses R regular instances and $(k - R)$ burstable instances.

requests, but queues would grow undesirably long. The extra $c\sqrt{R}$ instances are then used to handle the transient queueing that results from variability of traffic (i.e., both arrival time variability and request size variability). Thus, we propose to use R regular instances and $(k - R)$ burstable instances. The R regular instances would be used to handle the bulk of the traffic, whereas the $(k - R)$ burstable instances would only be used during short transient periods with higher amounts of queueing. We will often refer to these two parts of the expression for the number of instances as the "R part" and the "$(k - R)$ part," respectively. Even if we use an oracular scaling policy that selects the optimal number of k instances to use at each moment in time, the key idea behind BurScale of splitting k into R regular and $(k - R)$ burstable instances is beneficial in reducing cost.

Request Dispatching Policy. To make the above simple idea workable, we need to be careful about how we distribute requests among the regular and burstable instances. On the one hand, we would like to avoid situations wherein the burstable instances are used too frequently and run out of credits; on the other hand, we do not want to underutilize them thereby causing performance to degrade during transient surges in the workload. To address this problem, we explore the use of a weighted version of the join the shortest queue (JSQ) dispatching policy. The key challenge here is to choose the relative weight for burstable instances such that they accumulate credits during periods of low workload

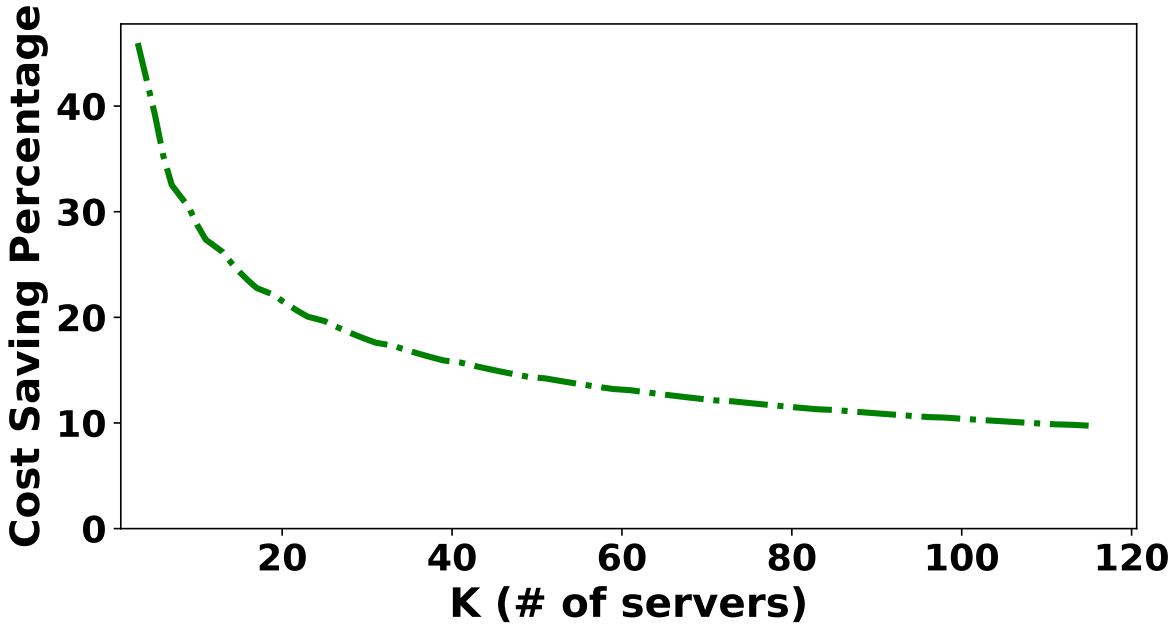

Figure 3.3. Percentage cost savings from using $(k - R) = c\sqrt{R}$ burstable instances vs. using all regular instances for different cluster sizes.

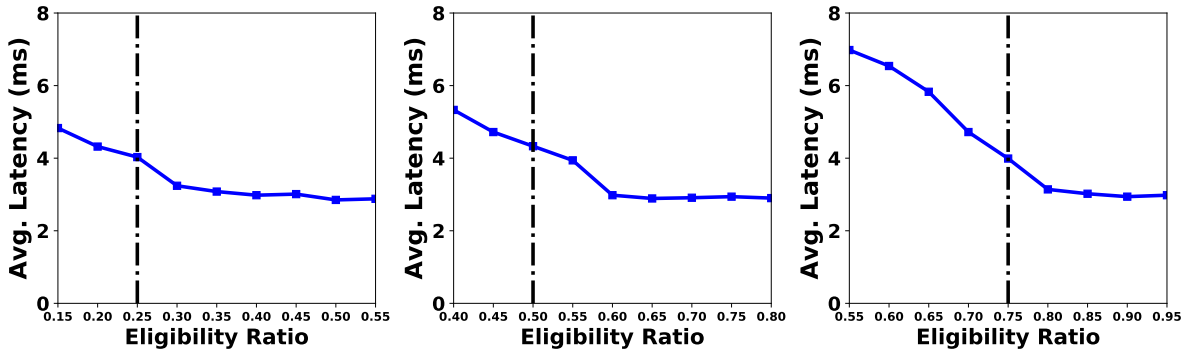

Figure 3.4. The effect of the eligibility ratio (i.e., fraction of traffic that can be handled by burstable instances) on latency in an in-memory key-value cache. Left: (3 regular, 1 burstable). Middle: (2 regular, 2 burstable). Right: (1 regular, 3 burstable). The latency stabilizes when the eligibility ratio exceeds $\frac{\#\ burstable}{total\ instances}$ (i.e., $\frac{k-R}{k}$), which indicates that the burstable instances have sufficient items cached to serve enough traffic for maintaining low latencies.

intensity and use credits during periods of flash crowds and high workload intensity. Towards this, BurScale periodically monitors the CPU usage of burstable instances and adjusts their weights until the CPU usage of the burstable instances is equal or less than the baseline credit earning rate allocated to them. This is because burstable instances have a net gain in CPU credits when their CPU usage is below the baseline rate.

Preliminary Validation. To validate our decomposition of k instances into R regular

and $(k - R)$ burstable instances, we run a preliminary microbenchmark with weights chosen for JSQ as described above. We run a workload with an average 800 reqs/sec arrival rate against our test web application (subsection 3.3.1) with a service rate of 113 reqs/sec. BurScale's autoscaling policy sets k to be 11, of which 8 are regular instances (R part) and 3 are burstable instances ($(k - R)$ part). BurScale dynamically updates the weights of instances, which prevents the burstable instances from losing credits in the long-term average while also meeting the SLO, as seen in Figure 3.2.

Cost Savings. From a cost saving point of view, we compare BurScale's approach to the case where all k of the instances are regular instances (m5.large). By converting some of the instances to burstable instances (t3.small), we are able to lower costs while still maintaining the desired SLO. Figure 3.3 shows the cost savings as a function of k for $P_Q = 0.1$ with current AWS pricing. As seen, small clusters may see up to 46% cost savings while larger clusters may see up to 10% cost savings.

3.3.3 Enhancements for Stateful Applications

We next enhance our technique for stateful applications such as an in-memory key-value cache (e.g., Memcached). As described in subsection 3.2.2, burstable instances are expensive when configured with a large memory capacity. Thus, applications that require a large amount of memory are unsuitable for burstable instances. One might think that this would preclude an in-memory cache, but surprisingly, there are scenarios when it is beneficial to use burstable instances for these types of applications. Specifically, when a large cluster of caching servers are needed to serve a high arrival rate of requests (e.g., as in [12]) due to network or CPU bottlenecks, then burstable instances can be used in the same way as in subsection 3.3.2. So in our experiments, we replicate items across caching servers and focus on the scenario where we use many caching servers to handle high request rates rather than increase hit ratios or the total cache size.

Existing works such as ElMem [23] and CacheScale [24] have demonstrated techniques for how to autoscale in-memory caches, so the focus of our work is on determining how to select the right type of burstable instance. The challenge here is that since the burstable instances will not contain as many items as regular instances, they will be incapable of serving some of the items. So in addition to choosing the number of burstable instances, BurScale also needs to select the amount of memory allocated to the instances. The way BurScale handles this is by first selecting the number of burstable instances as $(k - R)$ as before. Ideally, during periods where there is substantial queueing, one would expect them to handle a $\frac{k-R}{k}$ fraction of the traffic. Thus, we need the burstable instances to at

least be eligible to serve $\frac{k-R}{k}$ of the requests as compared to a regular instance. Practically, this needs to be slightly higher than this ratio since the requests arrive randomly, so we use $\frac{k-R}{k} + 0.1$ (determined experimentally below). We define the fraction of traffic that can be served by the burstable instances as the *eligibility ratio*, which is the ratio between the burstable and regular instances' hit ratios. Now once we have a desired eligibility ratio, we can then profile the workload offline to determine its popularity distribution. We then select the memory capacity of the burstable instances such that they achieve the desired eligibility ratio when replicating the most popular "hot" items. The hot items are determined through profiling, and we leverage mcrouter's *PrefixRouting* feature ("h" for hot keys and "c" for cold keys in our case) [25] to differentiate between the hot and cold keys so that the load balancer appropriately direct the traffic to the appropriate cache based on a weighted JSQ policy similar to our approach in subsection 3.3.2. Given that most popularity distributions exhibit skewness that can be captured by a heavy-tailed Zipfian distribution [13], only a small memory capacity is needed.

Preliminary Validation. To validate our selection of memory capacities, we run a microbenchmark where we vary the eligibility ratio and measure the resulting latency in BurScale. In this microbenchmark, network bandwidth is the constraining resource, and we use a fixed arrival rate of 15,000 requests per second, which results in $k = 4$. For the purposes of this microbenchmark, we experiment with all three combinations of regular and burstable instances: (3 regular, 1 burstable), (2 regular, 2 burstable), and (1 regular, 3 burstable). We adjust the cache in the burstable instances to only cache a hot subset of the items so as to achieve a given eligibility ratio. Figure 3.4 depicts the resulting average latency as a function of the eligibility ratio. With lower eligibility ratios, the burstable instances are unable to serve the colder items, and thus the regular instances experience greater queueing and latency as a result. The vertical dashed line in each graph indicates the $\frac{k-R}{k}$ ratio. As can be seen in these graphs, when the eligibility ratio is greater than $\frac{k-R}{k} + 0.1$, the burstable instances have enough items cached to serve a sufficient fraction of the traffic and maintain low latencies.

3.3.4 Cost-Effective Flash Crowd Handling

In this section, we show how BurScale yields even greater cost savings when a user is provisioning for a flash crowd, an unpredictable sudden increase in load. Flash crowds present a significant challenge for autoscaling systems as there is a delay in acquiring new resources [26, 27] during which performance suffers. Fundamentally, there needs to be hot spare capacity that is available for immediate use when a flash crowd occurs. *Our*

key insight is to use burstable instances for this hot spare capacity to mask the delay of starting and adding new regular instances.

A common approach to handling flash crowds is to overprovision by a factor m to mask the VM startup delay. For example, Netflix in the Nimble project [28] uses spare capacity (so called shadow resources) to handle the flash crowds that are caused by datacenter failures. That is, when a datacenter fails for whatever reason, they re-route the traffic to another datacenter, which is a type of flash crowd for the target datacenter. So the shadow resources act as an overprovisioning mechanism for this type of flash crowd. Like other autoscaling solutions [28], BurScale overprovisions resources to mask the VM startup delay. However, it does so more cost-effectively by using burstable instances for the overprovisioning.

As long as load does not increase by more than a factor m within the time period during which the system detects and incorporates new resources in response to a flash crowd, there is sufficient capacity to meet the SLO. Figure 3.5 illustrates this point when the load changes by a factor of $m = 1.75$ in a short time period. Provisioning for an arrival rate of λ in the top figure leads to periods of underprovisioning since the autoscaling cannot react fast enough to the increase in load. Provisioning for an arrival rate of $m\lambda$, on the other hand, can properly handle the flash crowd. BurScale uses the latter approach with an appropriately chosen m for handling flash crowds and hence does not suffer from underprovisioning in Figure 3.5.

Our work takes m as an input parameter and investigates *how to overprovision by a factor m in a cheaper fashion.* Note that selecting m is fundamentally a business decision based on the magnitude of a flash crowd that one is willing to spend money to overprovision for. Consequently, our results in section 3.5 demonstrate that BurScale works in various scenarios with different values of m.

Mathematically, to provision for an arrival rate of λ using the SR rule, we would use $k_\lambda = R + c\sqrt{R}$ instances with R regular instances and $c\sqrt{R}$ burstable instances. Provisioning for $m\lambda$ would consequently use $k_{m\lambda} = mR + c\sqrt{mR}$ instances with mR regular instances and $c\sqrt{mR}$ burstable instances.

We propose to utilize R rather than mR regular instances and $(k_{m\lambda} - R)$ burstable instances. In making this change, burstable instances are now responsible for handling the initial onset of a flash crowd. This introduces a new problem where the JSQ dispatching policy needs to be weighted such that the burstable CPU credits are not exhausted during normal operation, but are weighted during flash crowds to take advantage of the burst capability of burstable instances. We address this problem by designing BurScale to

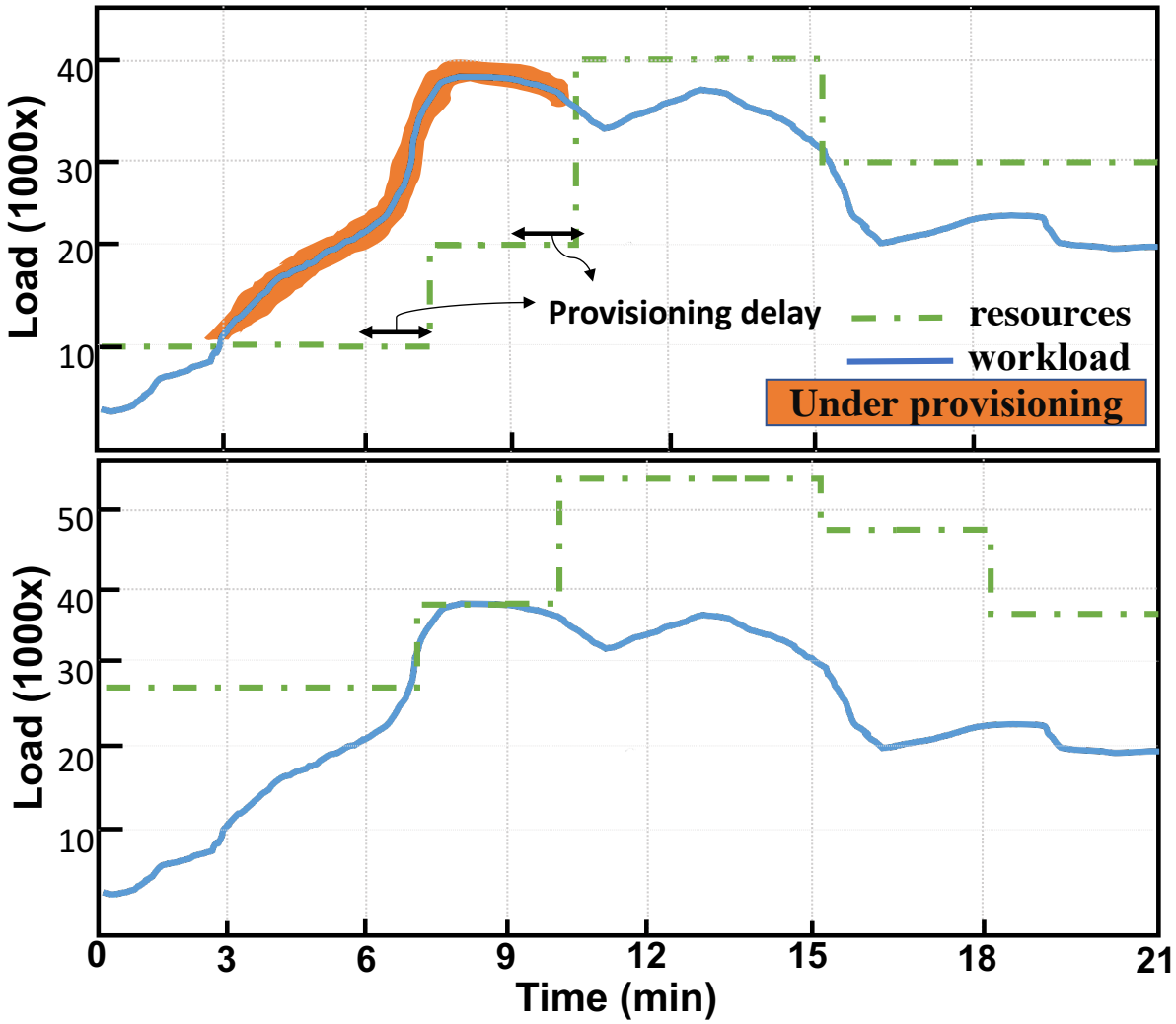

Figure 3.5. The effect of underprovisioning for flash crowds (top) and overprovisioning to handle flash crowds (bottom). BurScale overprovisions resources to mask the provisioning delay, but does so using cost-effective burstable instances.

detect flash crowds and dynamically change the weights of burstable instances during a flash crowd (details in section 3.4). That is, in a flash crowd, all the regular and burstable instances will be set to the same weight whereas during normal operation, burstable instances will be weighted to use them sparingly. Thus, burstable instances will collect CPU credits during non-flash crowd times so that they will have enough credits to use to cover the autoscaling delay when new resources are acquired for flash crowds.

Preliminary Validation. Figure 3.6 shows the effect of a flash crowd on a web application's performance (bottom graph) using a synthetic trace of a flash crowd (top graph). With normal provisioning for $\lambda = 200$ reqs/sec, there is substantial performance degradation during the flash crowd. By contrast, provisioning for $m\lambda$ ($m = 2$) with

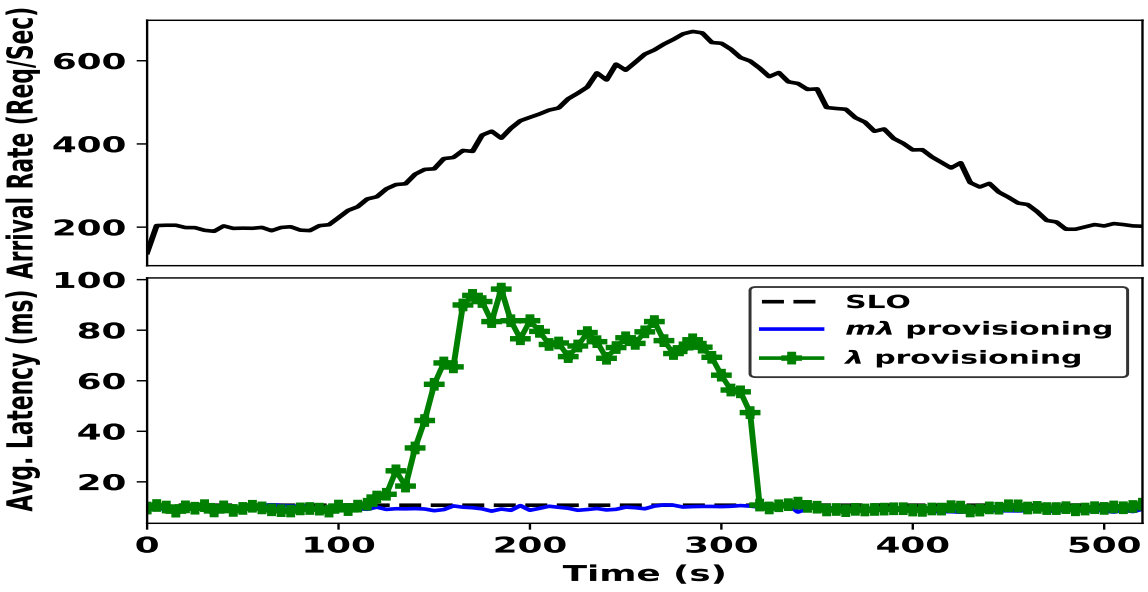

Figure 3.6. Overprovisioning is necessary to maintain low latency (bottom) during flash crowds (top). Even when overprovisioning with burstable instances, the average latency is below the desired dotted SLO line.

BurScale yields latency below the dotted SLO line.

Cost Savings. From a cost savings point of view, we compare our approach to the case where all $k_{m\lambda}$ of the instances are regular instances. Using burstable instances (t3.small) as hot spare capacity yields substantial cost savings in comparison to overprovisioning with regular instances (m5.large). Figure 3.7 shows the amount of savings as a function of $k_{m\lambda}$ for $m = 1.25$ and $P_Q = 0.1$ under the current Amazon AWS prices. As we can see, BurScale saves up to 50% in costs for small cluster sizes and over 20% for larger cluster sizes. We also compare this to the case where BurScale does not have the flash crowd enhancement (i.e., uses mR regular instances and $(k_{m\lambda} - mR)$ burstable instances). As expected, cost savings are not nearly as high, pointing to the importance of using burstable instances for flash crowds[3].

3.4 BurScale Implementation

We have implemented BurScale [4] on Amazon AWS EC2. Figure 3.8 shows BurScale's high-level architecture.

[3]While the percentage cost saving for BurScale without the flash crowd enhancement will approach 0 in the limit, BurScale will continue to provide savings even for very large clusters.

[4]The source code is available at: `https://github.com/PSU-Cloud/BurScale`

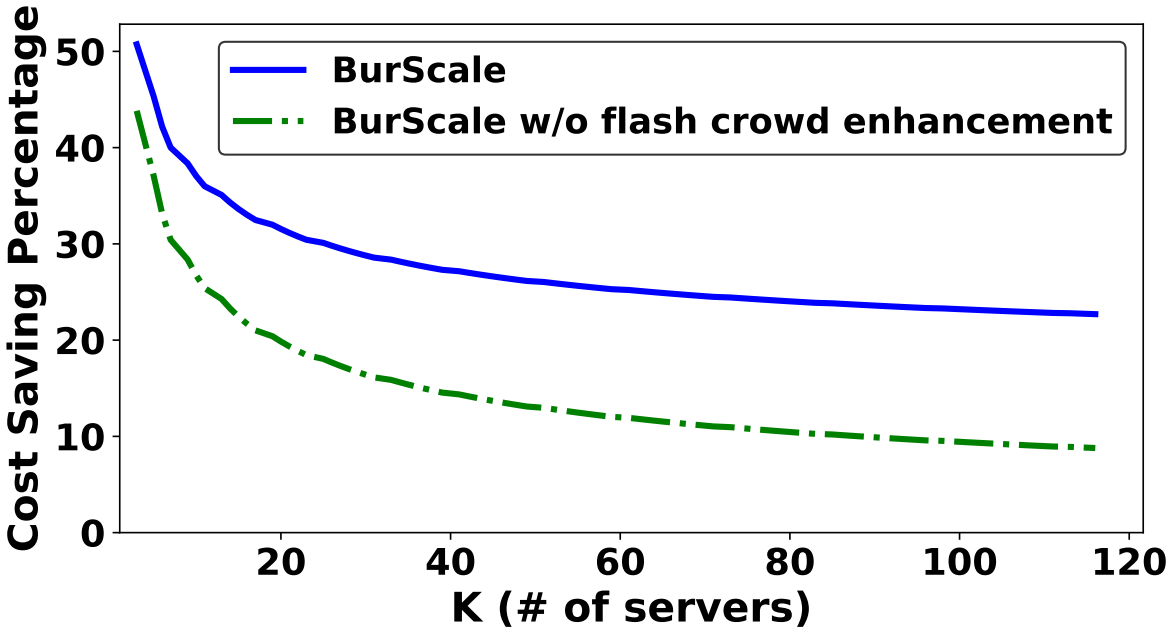

Figure 3.7. Percentage of cost saving using BurScale in flash crowd scenarios under an overprovisioning factor of $m = 1.25$.

Scaling Policy. The *Scaling Policy* module answers the question of *how many resources does our application need?* To make a decision and find an answer to this question, it needs a model along with some inputs. As described in Section 3.3.1, BurScale uses the *Square Root Staffing* Rule model, which only needs the current request arrival rate to the system and the profiled service rate of the application as inputs. The *Monitor* module sends the current request arrival rate to the *Scaling Policy* module. The *Scaling Policy* module then determines the number of instances, k, that the application needs. *BurScale can adapt any other scaling policy including those based on* predictive models, learning-based models, etc. One can easily implement a scaling policy of their own choosing - all the metrics that such a model may need as inputs are available from the *Monitor* module.

Monitor. The *Monitor* module periodically queries *AWS CloudWatch* [29] and the *Load Balancer*. It collects various metrics from the *CloudWatch* (e.g., CPU usage and credits) and from the load balancer (e.g., the current request arrival rate). It then provides these metrics to the *Scaling Policy* and the *Controller* modules.

Controller. The *Controller* module, which is the main component of BurScale, handles two primary tasks: (*i*) determining the number of regular and burstable instances based on the scaling policy, and (*ii*) adjusting the load balancer weights for the instances (LB

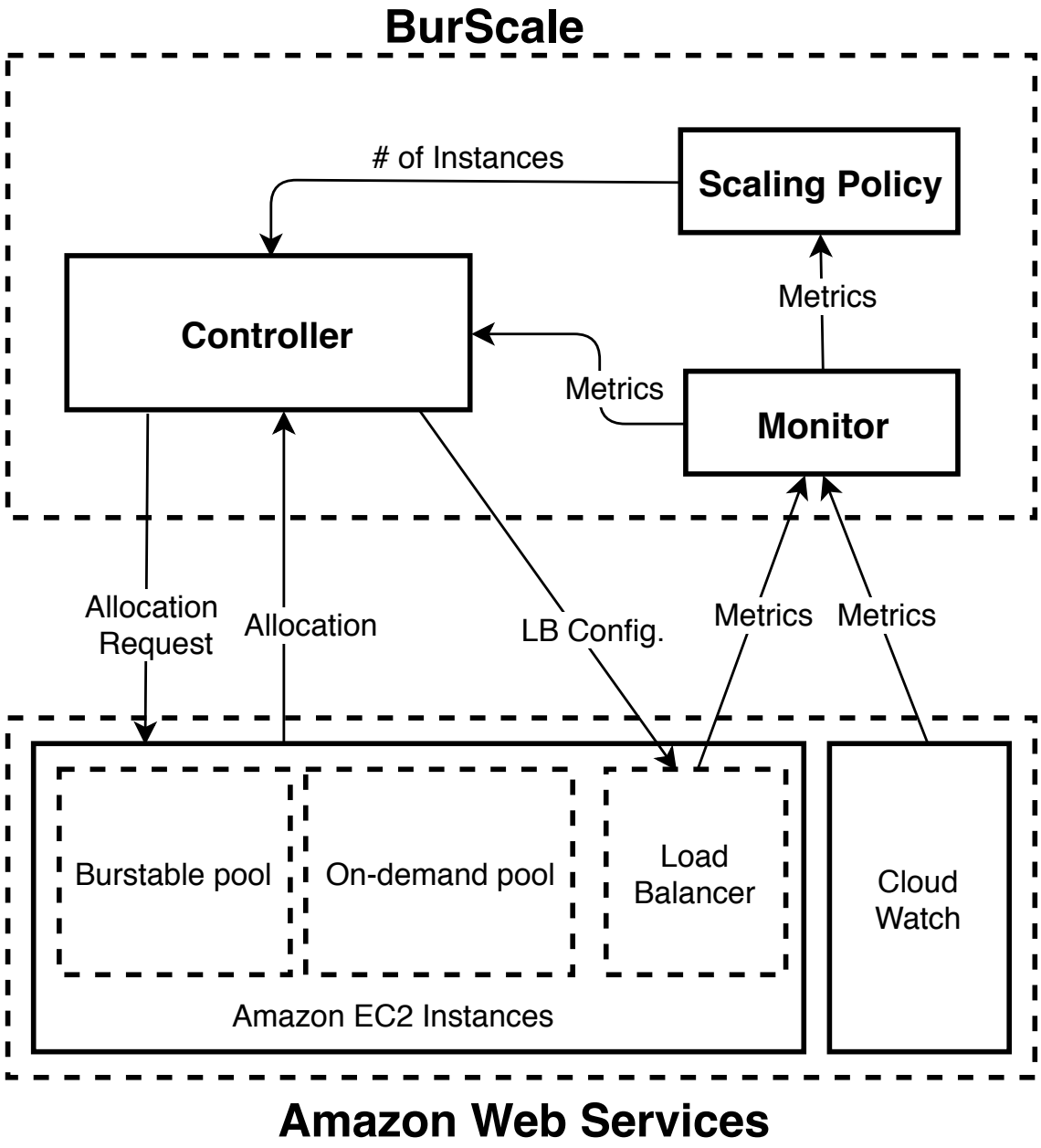

Figure 3.8. BurScale Architecture.

Config).

The *Controller* module keeps track of the state of current resources that are used. The state is maintained by having two pools of resources: an regular pool and a burstable pool along with their properties, such as their weights for the request dispatching policy and their state (e.g., available, unavailable, etc). This module maintains a total of k instances, where k provided by the *Scaling Policy* module. Based on the current arrival

rate, the controller calculates R, the minimum number of instances to keep the system stable. It then will procure R many regular instances and $(k-R)$ burstable instances. To maintain these numbers of instances, the controller will allocate and deallocate instances from the cloud provider and update the load balancer appropriately.

Flash Crowd Detection. The controller also detects flash crowds by maintaining a history of changes in the number of resources. If the changes in the number of servers is increasing in 3 consequent time windows (30 seconds each), the controller identifies it as a potential flash crowd. As the penalty for accidentally detecting a flash crowd is negligible (only a few extra burtable CPU credits would be spent), we opt for a simple mechanism that is not meant to be perfect. During a flash crowd, the controller will update the burstable weights in the load balancer dispatching policy so that the burstable instances will be utilized as much as the on-demand instances. During a non-flash crowd scenario, the controller will set the burstable instances weights so that the load balancer will only utilize them with sufficient queueing. This prevents burstable instances from running out of credits. The controller periodically gets the CPU usage of burstable instances from the *Monitor* module and decreases their weights until the CPU usage of the burstable instances is equal or less than the baseline rate allocated to them (40% for $t3.small$ instances in our experiments). This is because burstable instances start accumulating their CPU credits only when their CPU usage does not exceed the baseline rate.

3.5 Evaluation

We evaluate BurScale's performance using real-world applications and traces. We consider two case studies. First, we use the Wikimedia application, which is used for the Wikipedia website, along with a recent dump of the Wikipedia database of English articles. Second, we extend our ideas to the distributed key-value caching application Memcached. Our results show that BurScale performs well for these real-world applications - it is able to procure resources from EC2 cost-effectively and manage them such that the desired SLOs (in terms of mean latency) are met. We note that BurScale does not provide any guarantee on tail latency. Provisioning for tail latency may require selecting a different value for k. However, BurScale is primarily addressing replacing some of the regular instances from the k total instances with burstable instances, and we experimentally show that BurScale's tail latency is comparable to the case where we use only regular instances.

3.5.1 Web Application Case Study

Experimental Setup. We use Wikimedia $v1.31.0$ [30] as our web application. Wikimedia is a PHP-based application that is used by the Wikimedia foundation for operating the Wikipedia website. We use a dump of Wikipedia English articles from March 2018 [31] for populating the database tier of this application. To run a workload against the application, we use wikibench [32], a widely used benchamrking tool for the Wikimedia application. We use $m5.large$ instances with 2 vCPUs and 8 GB of memory each as our on-demand/regular instances and $t3.small$ instances with 2 vCPUs (with peak and sustainable rates of 100% and 20% of each vCPU respectively) and 2 GB of memory each as our burstable instances. As of May 2019, the unit price for $m5.large$ and $t3.small$ instances is \$0.096 and \$0.0208 per hour respectively. Our experiments use 20-70 instances, and are performed in the `us-east-2` AWS region.

Workload. For our workload, we use the most recent publicly available Wikipedia access traces from September 2007 [13]. We are unaware of other more recent public traces that provide fine-grain timing information (millisecond) along with request accesses (i.e., urls) and the application (i.e., Wikimedia) to execute the requests. The traces that we use are from September 19, 2007 in the access traces. Finally, the SLO target is set to 200ms.

3.5.1.1 Handling Transient Queueing

To evaluate BurScale in typical operating conditions, we replay 2.5 hours of the trace where the arrival rate increases by 50% during the trace. Figure 3.9(a) depicts the request arrival rate, which is scaled to fit our AWS cluster.

Meeting the SLO. We compare the performance of BurScale with *Reg-Only*, a baseline that only uses regular instances. As seen in Figure 3.9(b) BurScale is able to meet the SLO with performance comparable to *Reg-Only*, even though the burstable instances are lightly utilized for the transient queueing that develops during normal operating conditions.

Effect on Tail Latency. Figure 3.9(c) depicts the *95th* percentile of observed latency for BurScale and the *Reg-Only* baseline. BurScale has a comparable tail latency to *Reg-Only* while using cheaper burstable instances.

Cost Savings. Figure 3.9(d) depicts the total number of instances used throughout the experiment along with a break down of instances (i.e., regular and burstable) that are used. Since burstable instances are cheaper than regular instances, BurScale is able to save 16.8% in costs from this Figure 3.9 experiment.

How Burstable Instances are Used. Figure 3.9(e) depicts the CPU utilization of regular and burstable instances during the experiment. Since BurScale is operating under normal conditions without flash crowds, it sets the weights of burstable instances such that they accumulate credits. Hence, their CPU utilization must be less than their credit accumulation rate of 40%. Initially, their weights are equal to the regular instances, and BurScale dynamically updates the weights so that the CPU utilization is slightly under 40%. During the time that the weights are stabilizing, the CPU utilization is high, and the burstable instances consume some of their credits. As mentioned, one of the challenges of using burstable instances is that they can run out of credits and hence become less effective in serving the traffic. Therefore, our solution is to use them in a way where they slowly accumulate credits in normal operation so that they can be used in flash crowd situations. Figure 3.9(f) shows the credit state of burstable instances where *t3.small* instances start with 60 initial credits. As it can be seen, BurScale is able to set the weights so that the burstable instances start accumulating credits. The initial decrease in credits is because initially instances start with equal weights and BurScale updates the burstable weights so that their CPU utilization is low enough to accumulate credits.

3.5.1.2 Handling Flash Crowds

Next, we evaluate BurScale's handling of a flash crowd by artificially inducing a sudden surge in traffic at $t = 30$ minutes. As seen in Figure 3.10(a), the arrival rate suddenly increases by about a factor of 3, corresponding to a flash crowd with $m = 3$.

Meeting the SLO. Since resources are overprovisioned, BurScale is able to utilize the burstable instances at peak utilization during the flash crowd, thereby meeting the SLO as seen in Figure 3.10(b). As the *Reg-Only* baseline performs the same actions except with regular instances, it is also able to meet the SLO, but at a higher cost.

Effect on Tail Latency. Figure 3.10(c) depicts the *95th* percentile of observed latency for BurScale and the *Reg-Only* baseline. BurScale has a comparable tail latency to *Reg-Only*, which indicates that swapping some regular instances with lightly utilized burstable instances is acceptable for performance.

Cost Savings. Figure 3.10(d) depicts the total number of instances used throughout the experiment along with a break down of instances (i.e., regular and burstable) that are used. We calculate the total cost savings from Figure 3.10 to be 46.3%.

How Burstable Instances are Used. As seen in Figure 3.10(d), BurScale's controller identifies the increased traffic at $t = 30$ minutes as a flash crowd and initiates the

procurement of additional instances. During this time, the load balancer weights for burstable instances are set to the same as regular instances so that the burstable instances can effectively assist in the initial flash crowd rush. As can be seen in Figure 3.10(e), the CPU utilization of burstable instances increases during the initial flash crowd onset, during which new instances are being initialized and warmed up. Figure 3.10(f) depicts the credit state of the burstable instances. Initially the weights are equal and hence we see a decrease in the number of credits. BurScale periodically updates the weights so that burstable instances start slowly accumulating credits, and hence we see the increase in the credits in this graph. Once the flash crowd is detected at $t = 30$ minutes, BurScale sets the burstable and regular instance weights to be equal so that the burstable instances can be fully utilized to serve the traffic, and hence we see a significant decrease in the number of credits. When the new regular instances are added and warmed up in the autoscaling process at $t = 40$ minutes, BurScale updates the weights so that burstable instances start slowly accumulating credits again.

3.5.1.3 Sensitivity: Number of Regular vs. Burstable Instances

Given k, the total number of desired instances, BurScale assigns R regular instances and $k - R$ burstable instances, where $R = \lambda/\mu$, λ is the measured arrival rate, and μ is the measured service rate. We choose R based on intuition from queueing theory, but in this section, we evaluate the sensitivity of the split between regular and burstable instances. We run an experiment with the same workload as in Section 3.5.1.1 except that we vary the number of regular and burstable instances while keeping the total number k the same (e.g., R - 1 regular and k - R + 1 burstable instances). Figure 3.11 depicts the number of SLO violations for different choices for the number of regular instances. We count SLO violations by identifying how many 1-minute long intervals have an average latency exceeding the SLO out of a total of 143 intervals. As expected, provisioning more than R regular instances will work, but will be more expensive. More interestingly, we find that although provisioning slightly fewer than R regular instances results in some periods of SLO violation, the number of violations is low (below 5%). Thus, one could provision fewer regular instances if cost is more important than SLO violations. The extreme case of using only burstable instances would result in running out of CPU credits and substantially more SLO violations. Furthermore, if a burstable only cluster were provisioned to not run out of credits, then the cost would be higher than using regular instances as described in subsection 3.2.1.

3.5.2 Distributed Key-Value Cache Case Study

Experimental Setup. We deploy a web application that spans eight instances and uses a Memcached based cluster for its caching tier. We use the same $m5.large$ and $t3.small$ instances as before.

Workload. We use ETC, a read-dominant trace from Atikoglu et al. [2], for generating our workload. Figure 3.12(a) depicts its request arrival rate that we use in our evaluation. We set the working set size to be 3 GB. As the available memory capacity within our regular instances is more than the working set size, the hit rate is always 100% (ignoring capacity misses). Thus, the purpose of replicating instances in this experiment is for the ability to handle a high rate of cache traffic. We use a Zipf distribution for key popularity wherein 20% of the whole key set (i.e., 600 MB) are "hot" and account for 90% of the overall requests. We choose an average latency SLO of 5 ms.

Baselines. We compare BurScale's performance against two baselines. First, *RegRep* reactively procures additional capacity, but uses only regular instances and does not overprovision. Second, *RegOver* behaves similarly to *RegRep* except that it overprovisions resources. BurScale overprovisions just like *RegOver*, except it does so with burstable instances.

As the regular instances have enough memory to hold the working set, all regular instances replicate the entire working set. Thus, additional instances are not added for increasing the hit ratio, but rather the instances are used for serving the high rate of traffic. Burstable instances, on the other hand, do not have enough memory for the entire dataset, so they are only configured to contain a hot subset of the data.

Meeting the SLO. Figure 3.12(b) depicts the mean latency of BurScale and the other two baselines. After a fast increase in the arrival rate, the instances in *RegRep* get overloaded and result in high latency during the time when new resources are added and warmed up. By contrast, both *RegOver* and BurScale overprovision resources and are able to handle the fast increase. It is important to note that BurScale overprovisions using burstable instances, which only have enough memory to cache the hot content. Thus, these results show that even if a hot subset of data can be cached, the burstable instances can handle enough of the traffic to be useful.

Effect on Tail Latency. Figure 3.12(c) depicts the 95th percentile of latency for BurScale and the other two baselines. It can be seen that BurScale has a comparable tail latency with the *RegOver* baseline.

Cost Savings. Figure 3.12(d) shows the number and type of instances used by BurScale. As BurScale uses predominantly burstable instances in this experiment, it only spent

52.4% the cost of *RegOver*, which equates to a 47.6% savings. By contrast, *RegRep* spent 60.6% the cost of *RegOver* while still suffering from SLO violations. The reason for this is that BurScale achieves cost savings by using burstable instances for both (i) overprovisioning and (ii) transient queueing. By contrast, *RegRep* only saves costs by eliminating the overprovisioning.

How Burstables are Used. Figure 3.12(e) and Figure 3.12(f) show the CPU utilization and CPU credits of the burstable instances respectively. During the increase in traffic at around $t = 5$ minutes, the load balancer weights are adjusted to utilize the burstable instances more frequently while new resources are added to the cluster. This corresponds to the increase in burstable CPU utilization and decrease in CPU credits. Once the new resources have been added to the cluster, the burstable instances revert to just helping with the transient queueing that normally occurs, and hence they start slowly accumulating credits as it can be seen in Figure 3.12(f).

3.6 Related Work

We classify the related work along the following themes: (i) cost-effective resource scaling, (ii) exploiting burstable instances, and (iii) exploiting different cloud products/services in general.

Cost-effective Resource Scaling. Autoscaling has been extensively studied in the past decade with much research on reactive [16, 17], predictive [14, 15, 19, 27, 33], and hybrid [18, 19] approaches. Major cloud providers such as AWS, Google Cloud Engine, and Mircosoft Azure offer rule-based autoscaling systems [34–36]. For example, the AWS AutoScale service [34] autoscales resources based on tenant-defined rules. An example of a commercial entity configuring such a public cloud autoscaler is Spotify's BigTable Autoscaler [37]. This autoscaler keeps the cluster size within a pre-selected range while keeping the average CPU utilization at 70%. BurScale can be made to work with these autoscaling approaches by swapping a subset of the instances with burstable instances. Given the number of total instances, k, from the autoscaling policy, BurScale measures the load and system performance to determine the number of burstable instances to use. It then adjusts the load balancer to avoid overutilizing the burstable instances.

While the body of the related work focuses on determining the amount of resources (e.g., k in our case), the key novelty of BurScale is introducing a new technique for incorporating burstable instances into autoscaling systems while considering of cost-efficiency and performance.

Characterizing and Exploiting Burstable Instances. Wang, C. et al. [38] present a study of the token-bucket mechanisms that governs the dynamism in CPU capacity and network bandwidth of AWS burstable instances. Jiang et al. [39] present a unified analytical model for evaluating burstable services. Such a model enables tenants to optimize their usage and also allows cloud providers to maximize their revenue from burstable offerings. BurScale goes beyond theoretical analysis and proposes a practical approach for utilizing burstable instances in multiple use cases including flash crowd provisioning. Wang, C. et al. [40] explore two cases where using burstable instances leads to cost savings. First, it uses burstable instances for backup of popular Memcached content stored on cheap but revocation-prone spot instances. This helps mitigate performance degradation during the period following a spot revocation while its replacement is being procured. Second, it shows how multiplexing multiple burstable instances over time can sometimes offer CPU capacity and network bandwidth equivalent to a regular instance at a lower price. BurScale builds upon this work and demonstrates how burstable instances can benefit more general use cases such as web applications and distributed key-value stores.

Our work describes completely novel use cases for burstable instances that can bring new cost saving opportunities that are complementary to some of these earlier works.

Combining Different Cloud Products and Services. Several works have exploited the diversity of instance types to achieve cost-effective resource provisioning, and we discuss a subset here. In particular, researchers have successfully exploited the low prices of revocable spot instances [40–42] and reserved instances [43]. BurScale is complementary to these approaches since there are burstable instance types within the spot and reserved instance price models (i.e., it is possible to purchase a spot/reserved burstable instance). One would first use BurScale to partition the total number of instances between burstable and regular instance types. Then, one would apply these related works twice, once for provisioning the set of burstable instances and once for the regular instances, to determine the number of reserved/spot burstable instances and reserved/spot regular instances.

There are also systems that use complex optimizations to provision resources. For example, Kingfisher [44] is a cost-aware provisioning system that uses a combination of different instance types (although not burstable instances) using an integer linear programming model. Once a system such as Kingfisher decides the number of instances to provision of a given type, BurScale can then exchange some of them for corresponding burstable instances to reduce costs.

Comparison with Serverless Functions. Recent work has considered exploiting the agility of serverless offerings such as AWS lambdas for helping mask the latency of launching new VMs [45]. Burstable instances as used in BurScale are available nearly instantly whereas even warm-start lambdas take about 100 ms; cold-start functions may take a few minutes just like VMs [46]. BurScale requires minimal to no changes to the application (recall our changes were limited to only the load balancer) whereas using functions requires more extensive changes. In fact, some applications may not be portable with functions in their current restrictive form [47, 48]. Among the most prominent restrictions are their limited lifetime of a few minutes and their inability to engage in point-to-point communication (for state transfer) with other functions or from other VMs, which in turns necessitates the use of a slow external storage medium such as AWS S3.

3.7 Conclusion

The recently introduced burstable instances in the public cloud are advertised as being suitable for tenants with low intensity and intermittent workloads. By contrast, in this chapter, we present two novel usage scenarios for burstable instances in larger clusters with sustained usage. We demonstrate (i) how burstable instances can be utilized alongside conventional instances to handle the transient queueing arising from variability in traffic, and (ii) how burstable instances can mask the VM startup/warmup time when autoscaling to handle flash crowds. We implement our ideas in a system called BurScale and use it to demonstrate cost-effective autoscaling for two important workloads: (i) a stateless web server cluster, and (ii) a stateful Memcached in-memory caching cluster. Results from our prototype system show that via its careful combination of burstable and regular instances along with a dynamically adapting weighted JSQ request dispatching policy, BurScale is able to meet latency SLOs while reducing costs by up to 50% compared to traditional autoscaling systems that only use regular instances.

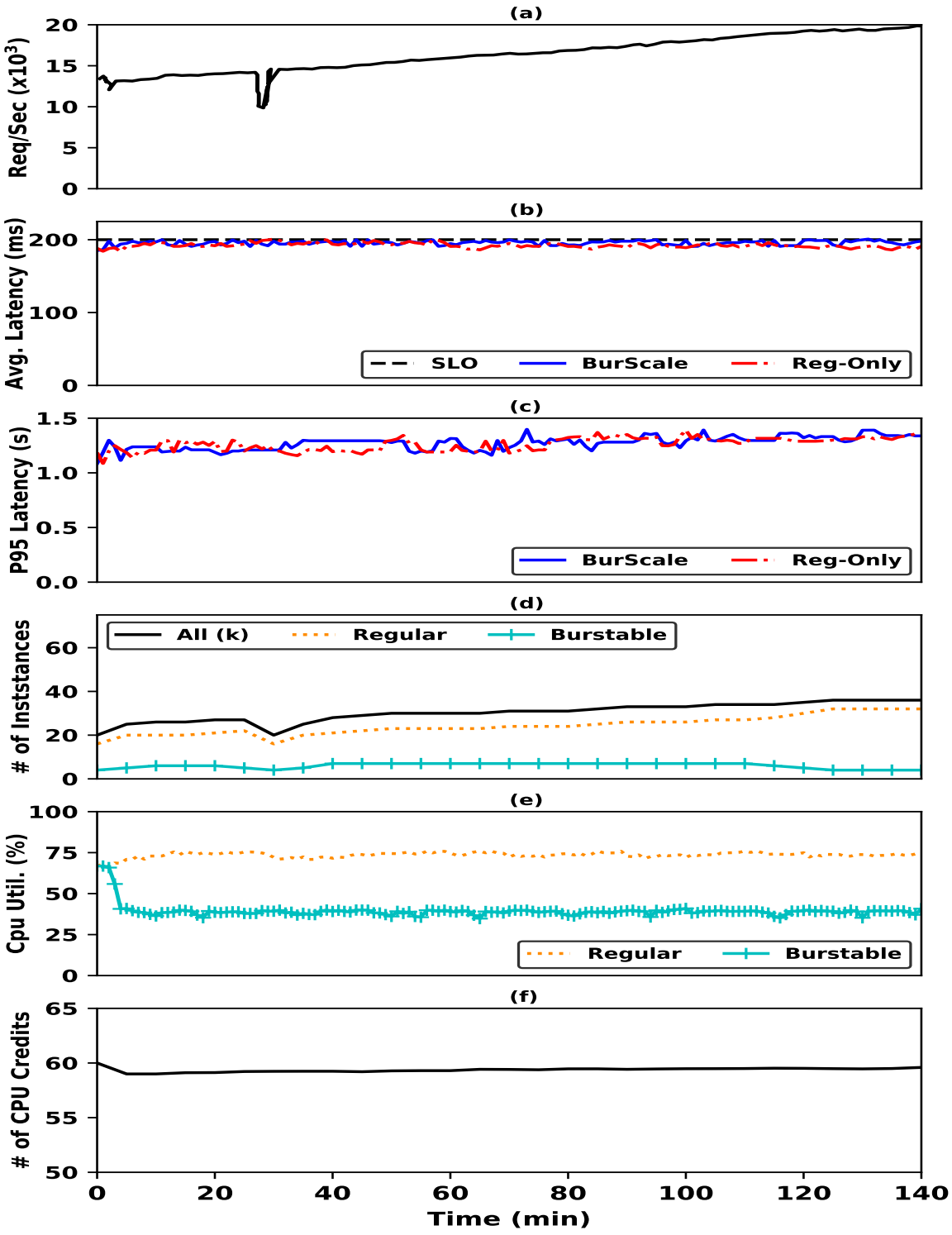

Figure 3.9. Results for the transient queueing experiment (details in section 3.5.1.1) using 2.5 hours of the Wikipedia trace (a). Switching some of the instances to burstable instances (d) results in 16.8% cost savings while matching the performance of using only regular instances (b & c). The burstable instances' CPU credits (f) are not exhausted as the load balancer avoids extensively utilizing the burstable instances (e).

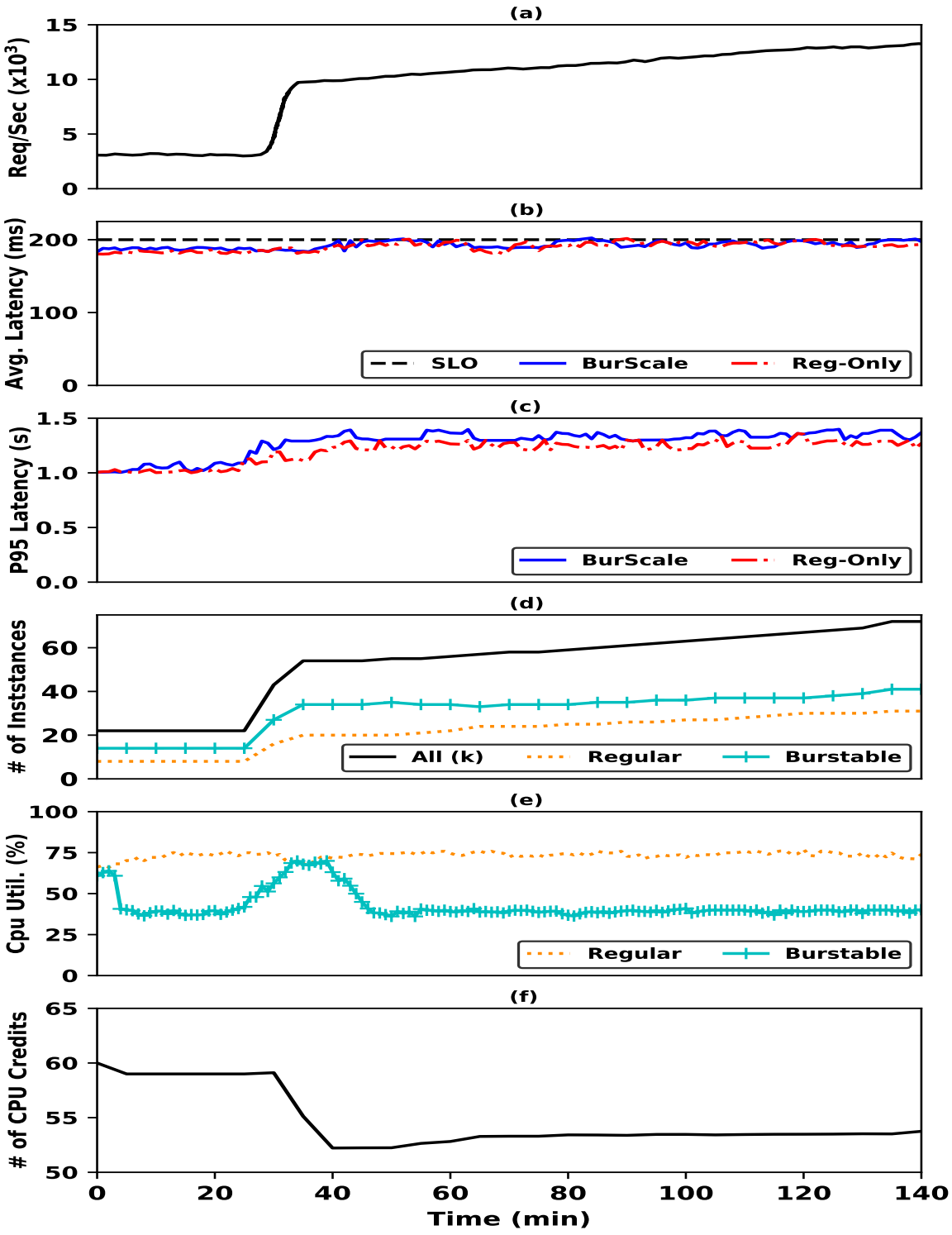

Figure 3.10. Results for the flash crowd experiment (details in section 3.5.1.2) with a flash crowd induced at $t = 30$ minutes (a). BurScale handles the flash crowd by overprovisioning using burstable instances (d), resulting in a 46.3% cost savings over using only regular instances while matching performance (b & c). The burstable instances' CPU credits (f) are only significantly utilized during the onset of the flash crowd where BurScale configures the load balancer to fully utilize the burstable instances (e).

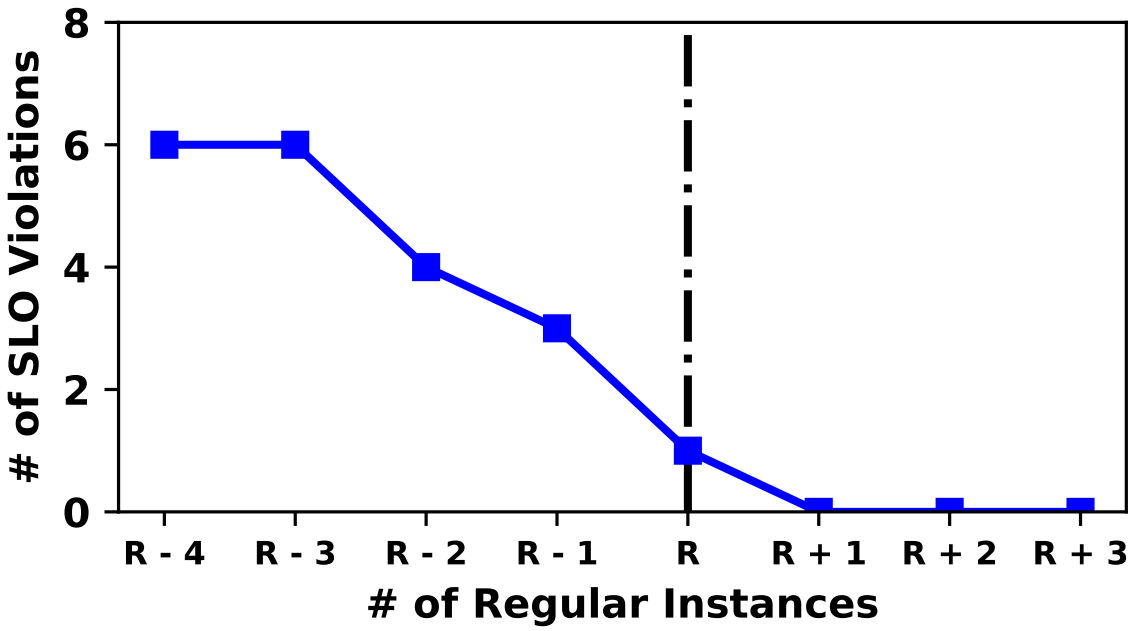

Figure 3.11. Sensitivity to the number of regular and burstable instances. We vary the number of regular (x-axis) and burstable instances while maintaining a fixed total of k instances. We measure the number of 1-minute intervals where the latency exceeds the target SLO (y-axis). This experimentally demonstrates that R, as defined in Theorem 1, is a good choice for the number of regular instances.

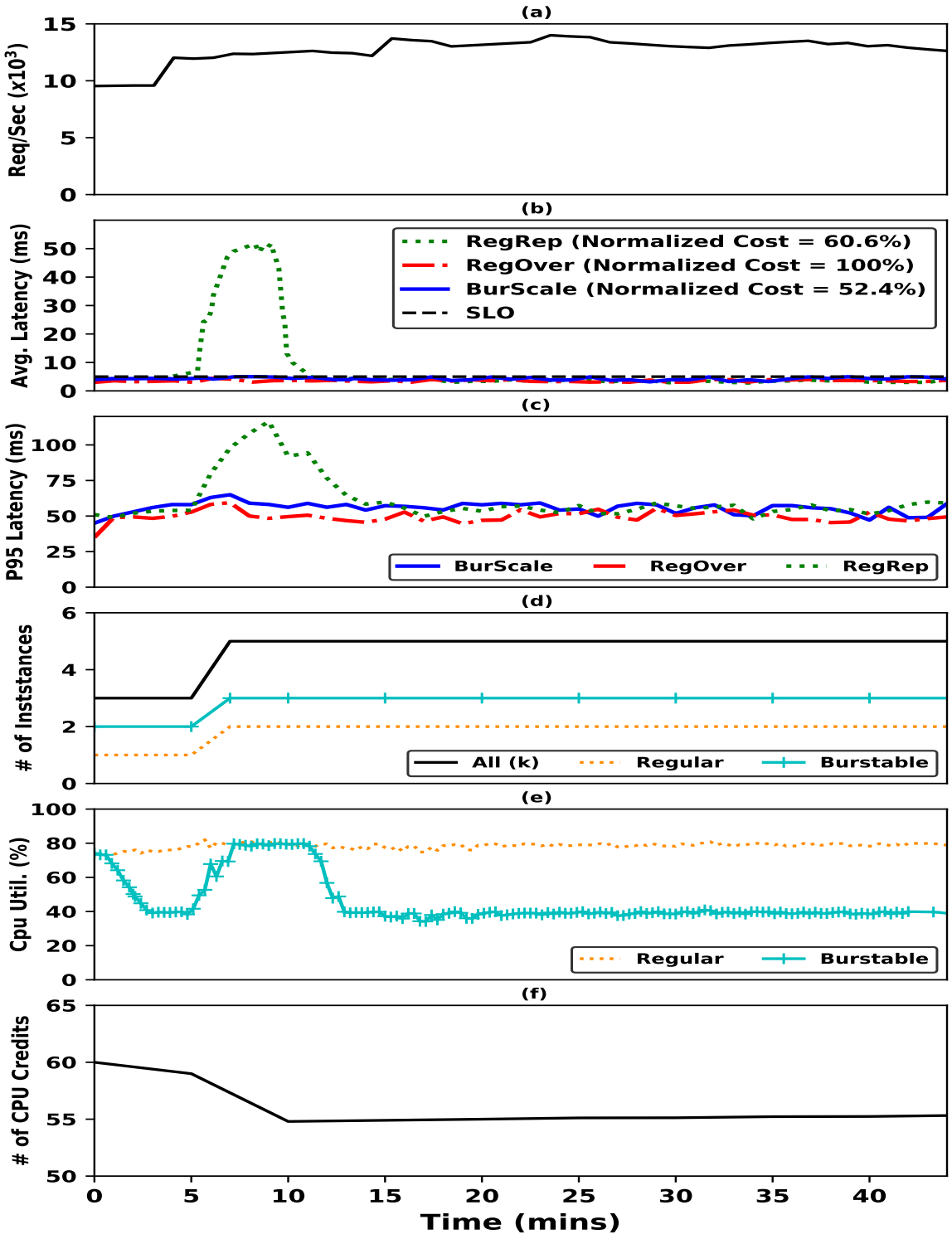

Figure 3.12. Results for the Memcached experiment (details in section 3.5.2) using the ETC trace (a) from [2]. Using burstable instances (d) results in a 47.6% cost savings compared to overprovisioning using only regular instances (RegOver). Even when the burstable instances cannot cache all the items, BurScale is able to match the performance of RegOver (b & c). Without overprovisioning (RegRep), latency can significantly increase (b & c) during periods of autoscaling ($t = 5 - 10$ minutes in d). During these periods, the burstable instances' CPU credits are significantly utilized (e & f).

Chapter 4
Right-Sizing and Efficient Scheduling of Microservices

4.1 Introduction

The microservices architecture is a recent and increasingly popular paradigm for designing interactive and user-facing services where hundreds of small and fine-grained components (i.e. "microservices") collectively work on serving end-user requests in a distributed setting [49–53]. Breaking down an application into small microservices brings several benefits. It allows different developer teams to independently work on (possibly technologically) different microservices [54]. Also, each microservice can scale and operate independently depending on its own state and incoming workload which results in better performance and reliability of the application as a whole [55]. Finally, a microservices architecture may facilitate debugging for performance and correctness issues [56].

One of the important challenges in deploying microservices is cluster resource allocation and scaling. To meet some performance and reliability goals, before deploying the microservices, the developers (application owners) have to specify the size of each microservice in terms of compute resources such as CPU and memory (i.e. "vertical sizing"), as well as the number of instances (or "replicas") for each microservice (i.e. "horizontal sizing"). However, the resource needs of microservices will depend on potentially complex demand processes, and may be difficult to predict a priori. On the one hand, allocating more compute resources than what is required for the microservices to operate normally leads to low cluster resource utilization which is not cost effective [57, 58]. On the other hand, allocating less resources than what is needed for the microservices can lead to performance degradation and service unavailability both of which can result in revenue loss for the application owner [59].

In this chapter, we present *SHOWAR*, a system designed for both horizontal and vertical autoscaling of microservices managed by Kubernetes [60], the state of the art container orchestrator platform. For vertical autoscaling, *SHOWAR* embraces the variance in the historical resource usage to find the optimal resource sizing of each microservice to maintain a good performance while avoiding low resource utilization. For horizontal autoscaling, *SHOWAR* uses metrics from Linux kernel thread scheduler queues (in particular eBPF *runq latency*) as its autoscaling signal to make more accurate and meaningful autoscaling decisions. At its core, *SHOWAR* uses basic ideas from control theory to control the number of replicas for each microservice based on signals from the microservice's run-time. In particular, we designed a *proportional–integral–derivative* (PID) controller [61,62] as a stateful autoscaler that uses the historical autoscaling actions and current run-time measurements to make the next horizontal autoscaling decisions and keep the microservice "stable." Additionally, by taking into account the dependencies between different microservices, *SHOWAR* prioritizes the dependee microservices[1] to prevent unnecessary autoscaling actions and low resource utilization.

In addition to its use of autoscalers to determine resources for the microservices, *SHOWAR* aims to bridge optimal resource allocation and efficient scheduling of microservices. That is, once the optimal size for a microservice is determined, *SHOWAR* assists the cluster scheduler to schedule microservice for better end-to-end performance. In particular, to prevent resource contention and manage noisy-neighbor effects on microservice performance, *SHOWAR* uses estimated correlations of historical resource usage between different microservices to generate rules for the Kubernetes scheduler. These rules may, e.g., suggest the scheduler to co-locate (scheduling affinity) the microservices that have negative correlation of a certain resource type, or otherwise distribute them (scheduling anti-affinity).

We evaluate *SHOWAR* by deploying a variety of interactive microservice applications on a cluster of virtual machines on the Amazon Web Services (AWS) public cloud. We compare the performance of *SHOWAR* against two state of the art autoscaling systems: A moving window version of Google Autopilot [63] and the default Kubernetes autoscalers [64]. Using real-world production workloads, our results show that *SHOWAR* outperforms these baselines in terms of efficient resource allocation and tail distribution of end-to-end request latency. In particular, *SHOWAR* on average improves the resource allocation by up to 22%, which can directly translate to 22% in total savings of cluster-related costs, while improving the 99th percentile end-to-end user request latency by 20%.

[1] i.e., microservices that others depend on

In summary, we make the following contributions:

- *Vertical and Horizontal Autoscaler Frameworks*: We present the design of framework for both vertical and horizontal autoscalers for microservices that aim to improve the resource allocation efficiency.
- *Scheduling Affinity and Anti-Affinity Rules*: We bridge the gap between right sizing the microservices for resource efficiency and efficient microservices scheduling by generating scheduling affinity and anti-affinity rules to assist the scheduler for better microservices placement and performance.
- *Implementation and Evaluation*: We implement the designed frameworks and mechanisms creating a system called *SHOWAR* that works on top of the Kubernetes container orchestrator platform. Using real-world production workloads and real-world microservices we demonstrate the efficiency of *SHOWAR* compared to the state of the art baselines.

4.2 Background and Related Work

4.2.1 Background: Microservices Architecture

In microservice architecture, the application service is broken-down into different individual microservices that work together to process a request. Figure 4.1 shows the Social Network application [65] as an example of microservices architecture. Each microservice is itself a containerized application that communicates with the other microservices using standard HTTP API requests or RPC calls. A user request arrives at the front-end layer of the application, and depending on the type of the request, it will span a subset of microservices (from front-end tier, middle tier, and back-end tier) to serve that request in the fashion of a directed acyclic graph (DAG). As can be seen in Figure 4.1, the microservices form a kind of dependency graph, moving from the front-end tier to the back-end tier, where some of the microservices obviously depend on others. This dependency graph information is used by *SHOWAR* to make better autoscaling decisions (see subsection 4.3.2).

4.2.2 Background: Microservices Deployment and Scaling

The increase in the popularity of microservices architecture has led to many industrial standards including container orchestrator tools for deploying microservices applications

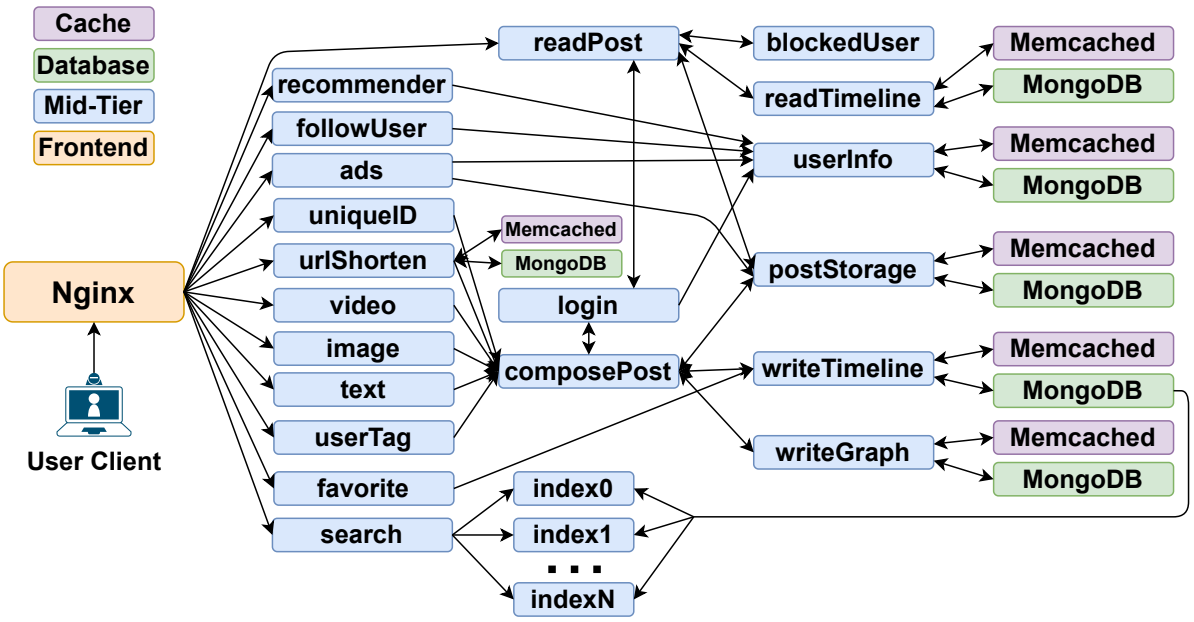

Figure 4.1. Social Network application as an example of microservices architecture

[60, 66, 67]. In this work, we use Kubernetes (k8s) as our container orchestration platform and build *SHOWAR* on top of it. In k8s, each microservice consists of at least one *Pod*, where each Pod consists of at least one application container, where a Pod is the smallest entity that k8s schedules. For better performance and availability, a microservice might have multiple replicated instances which we call service replicas or simply replicas herein. The size of each Pod in terms of compute resources such as CPU and Memory, and the number of replicas of it, are initially set (e.g., by the application owner, developer or user), and thereafter, the Kubernetes autoscalers adjust the size of Pods. There are two types of autoscalers: The *Horizontal Autoscaler* which determines the number of replicas for each Pod, and the *Vertical Autoscaler* which determines the Pods' compute resources such as the amount of CPU and Memory. The autoscalers base their decisions on the historical resource usage of each Pod.

Deploying microservices requires the application owner to specify the IT resource requirements, including CPU, memory and disk, for its Pod's containers. Setting fixed values for each resource requirement of an application may be wasteful as the application's usage pattern changes over time. On the one hand, worst-case provisioning results in underutilized compute resources (wasted from the application owner's point-of-view) which is not cost-effective. On the other hand, under-provisioning the resources for the application could lead to throttled CPU or, worse, out-of-memory (OOM) errors which adversely affect end-to-end workload performance and hence user experience. In such

situations, vertical autoscaling helps in finding the correct resource requirements for the application as workload and the usage patterns change over time, while meeting the application owner's service-level performance objectives (SLOs). By dynamically achieving optimal resource allocations for the application, an efficient vertical autoscaler a) minimizes the *resource usage slack* (i.e., slack = limit - usage) caused by over-provisioning and b) minimizes the number of OOM errors and the amount of time that the CPU is throttled due to under-provisioning.

Horizontal autoscaling refers to dynamically specifying the number of replicas for each microservice. As the load for a microservice increases, in order to prevent long backlogs in its request queues, horizontal autoscaling would increase the number of microservice replicas to maintain good responsiveness. In addition, in some situations, vertical autoscaling cannot increase the size of a service replica beyond a threshold. These thresholds can be the size of the largest virtual machine in the cluster, or the size beyond which the microservice does not benefit due to the lack of parallelism in the requests. Therefore, in such situations, one would use horizontal autoscaling to increase the number of replicas for the microservices.

In addition to CPU and Memory usage data, *SHOWAR* uses the extended Berkeley Packet Filtering (eBPF) [68] metrics data for its horizontal autoscaling decisions. eBPF is a recent Linux kernel technology that enables running secure and low-overhead programs at the kernel level to collect accurate metrics from the kernel level events such as CPU scheduler decision events, memory allocation events, and events of packets in the kernel's networking stack. It has been widely used for microservice observability toward a wide range of purposes such as performance improvements, profiling and tracing, load balancing, network monitoring and security [69–72].

4.2.3 Control Theory in Computer Systems

In order to achieve stability for many systems, a "controller" is used. A widely-used family of such controllers are known as feedback controller systems in which the controller uses the feedback signals from the system to perform some corrective actions to stabilize the system.

Example (among many) computer systems that use controllers for stability include caching systems. In caching example, the goal is to maintain a target cache hit rate. A feedback controller can thus be used that monitor the cache rate (i.e. the signal) and either increases or decreases the cache size (i.e. the corrective action) to reach the target cache hit rate. Another example of a computer system that uses feedback control system

to reach stability is autoscaling systems such as VM cluster autoscaler and application autoscaler.

In the following, we discuss different forms of feedback control systems for application Pod autoscaling that are used by Kubernetes [64], Google Autopilot [63], and *SHOWAR*for Pod horizontal autoscaling.

In autoscaling systems, unusually, the goal is to maintain a target performance metric such as latency or resource utilization such as CPU utilization. Therefore, for a target CPU utilization, the goal is to set the number of Pod replicas such that the observed CPU utilization is close "enough" to the target CPU utilization. To determine the number of replicas, an error term $e(t)$ is defined as follow:

$$e(t) = observation - target$$

The error term indicates how far the current observation is from the target. Hence, a "transfer function" is used to translate this difference to autoscaling actions. For example, if the error term is positive, it means more replicas are needed to bring the observed CPU utilization close to the target utilization. Similarly, if the error term is negative, it indicates are more than enough replicas and hence we can reduce the number of replicas for better resource allocation efficiency. Thus, the number of replicas will be a function the error term:

$$Number\ of\ replicas = k_P e(t) = F(e(t))$$

In control theory, this type of feedback controller is known as a "Proportional" controller because the output the controller is proportional to the error term (the proportionality is determined by the proportional coefficient k_P).

One of the fundamental problems with the proportional feedback controllers is that they can result in oscillation in systems behavior and hence they are slow in reaching the target stable point. This is illustrated in Figure 4.2.

To eliminate the oscillations in the proportional controller and hence reducing the noise in the controllers output, one can add an error integration term to the proportional controller and construct a "Proportional-Integral" controller as follows:

$$Number\ of\ replicas = k_P e(t) + k_I \int_{t-w}^{t} e(\tau) d\tau = F(e(t))$$

The intuition behind the integral term in the Proportional-Integral controller is that, in addition to the current error term, it bases on total accumulated errors. This strategy

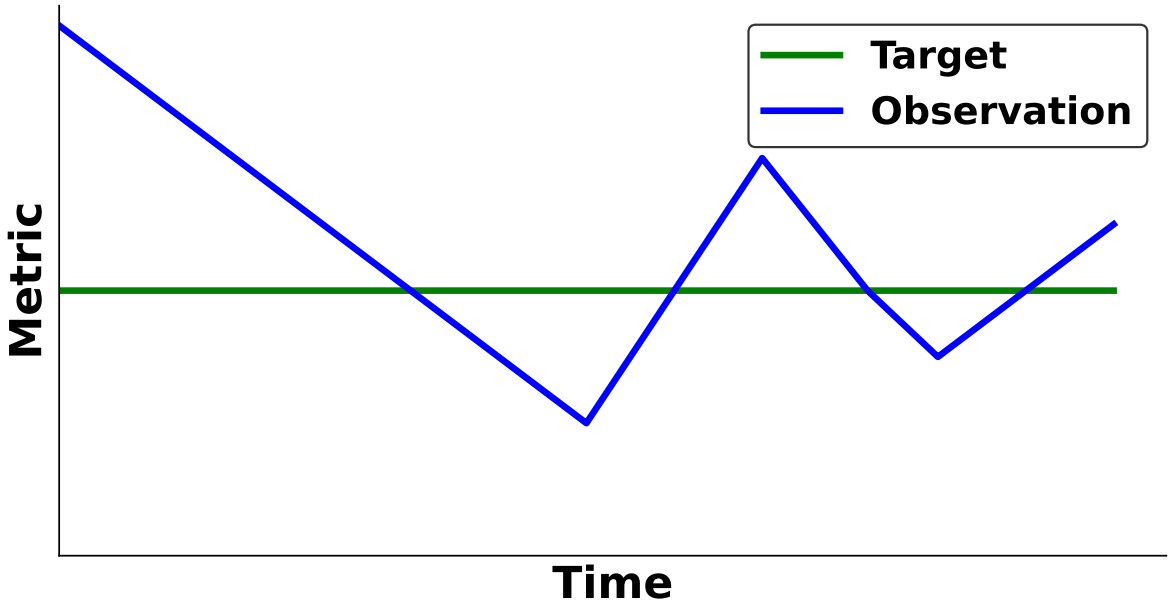

Figure 4.2. Systems behavior under a basic proportional controller.

amplifies the controller's output and hence it reduces the noise and eliminates the errors more quickly. This is illustrated in Figure 4.3 where the controllers output eliminates the oscillations and increases the stability.

While adding the integral term reduces the oscillations by incorporating the past memory of the controller, it is still slow in reaching the steady state. To improve the controller, one can add a derivative term as a predictor to increase the speed of the controller as follows:

$$Number\ of\ replicas = k_P e(t) + k_I \int_{t-w}^{t} e(\tau)d\tau + k_D \frac{d}{dt} e(t) = F(e(t))$$

This is known as a "Proportional-Integral-Derevatice" (PID) controller. The derivative term measures the rate of the changes in the observation. So, if it is positive, it indicates that the error is increasing and hence the autoscaler needs to add more Pod replicas to reduce the error term. Similarly, when it is negative, it indicates that the error term is decreasing and hence the controller can reduce the number of replicas.

In section 4.3, we discuss how we use the PID controller in *SHOWAR*'s design.

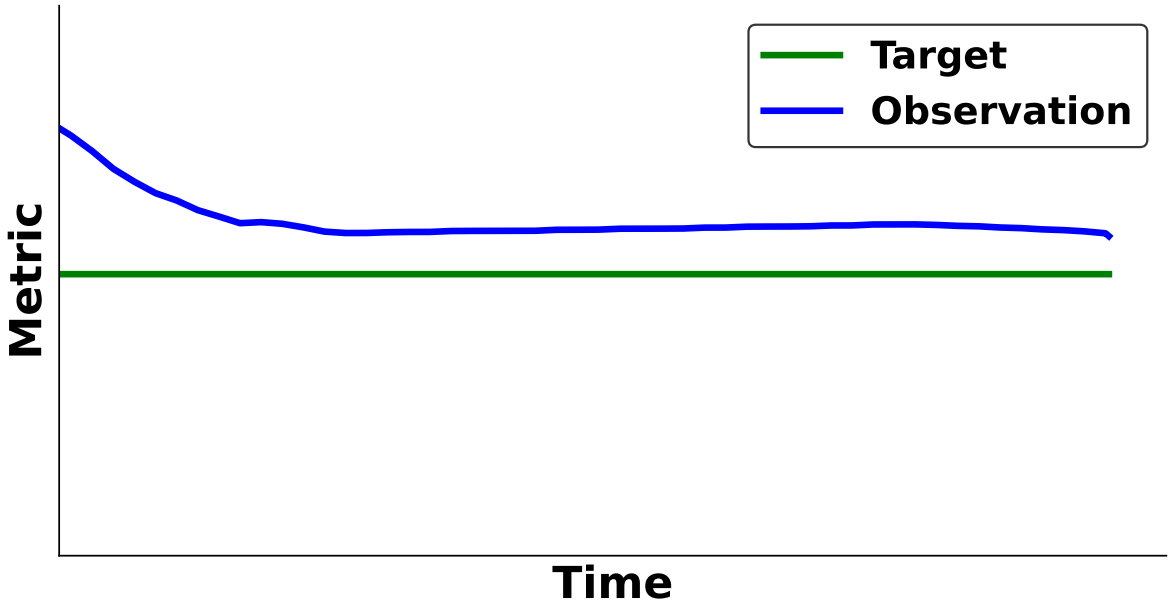

Figure 4.3. Systems behavior under a proportional-integral controller.

4.2.4 Related Work

Here, we give a brief overview of the state of the art related work on autoscaling and resource management for microservices which we build upon and use as our baselines.

Autoscaling Microservices. Autoscaling has been studied in the context of public cloud resources [73–76], including different types of workloads [4,77] and microservices [63, 64,78]. The autoscaling framework for microservices which is widely used by practitioners in industry are those of Kubernetes [64] and Google Autopilot autoscaler [63].

For vertical autoscaling, the Kubernetes vertical autoscaler uses the historical resource usage of both CPU and Memory and calculates the 90^{th} percentile (P90) of the aggregated resource usage across the replicas of each microservice for the last (sliding) window of N (a system parameter) samples. It then sets the resource (CPU and Memory) limits for the microservice for the next time window to be P90$\times$(1.15). The extra 15% is used as a safety margin to avoid under-provisioning resources for the microservice. Google Autopilot's vertical autoscaler takes the same approach with a slight modification. In particular, for the Memory resource, instead of the P90, it uses the *maximum* (max) Memory usage in the last N samples as the limit for the next window time. But for the CPU resource, it uses P95$\times$1.15 (not P90) for the next time window. In *SHOWAR*'s vertical autoscaler design, instead of using high percentiles in this way, we use the empirical variance in resource usage for improved resource allocation to the microservices

(see subsection 4.3.1).

For horizontal autoscaling, both Kubernetes and the Google Autopilot take the same approach. The application developer/user sets a target CPU utilization T^* for each microservice[2]. At any time t, for each microservice M, the number of replicas R_M is calculated as follows:

$$R_M = \frac{\sum_r \text{P95}_r}{T^*},\ r \in \{M\text{'s current replicas}\}$$

where P95_r is the 95^{th} percentile of the CPU usage of the replica r for the microservice M in the last N samples. In *SHOWAR*'s horizontal autoscaler design, we take a totally different approach and instead of using CPU utilization as an autoscaling metric, we use the CPU scheduler's eBPF metrics to design a more accurate and stateful autoscaler (see subsection 4.3.2). As another alternative to the sliding window approach, the Google Autopilot vertical autoscaler also employs an ensemble of machine learning models for its vertical autoscaler. The models try to optimize a cost function of autoscaling actions (e.g. under- and over-provisioning) and the model with the lowest cost is picked to perform a vertical autoscaling action on each microservice.

Autoscaling to meet SLOs. Another line of prior work, which is orthogonal to autoscalers such as *SHOWAR*, utilizes autoscaling actions as a means to prevent service level objective (SLO) violations due to microservice straggling [79–83]. For example, FIRM [79] is a machine learning (ML) based system for predicting and mitigating the SLO violations in microservices. It first localizes the SLO violation to determine which microservice in the dependency graph is contributing to the SLO violation the most, then using a pre-trained reinforcement learning (RL) agent, FIRM performs proactive autoscaling actions on different resource types (CPU, Memory, LLC etc.) to mitigate the SLO violation. Sage [80] is a supervised ML based system designed to mitigate performance problems in microservices. In particular, it models the dependencies between the microservices using causal Bayesian networks to identify the root cause of SLO violations and then performs autoscaling actions to mitigate the SLO violation. The fundamental limitation of this line of work is that they use pre-trained machine learning models that cannot be easily adapted to the changes in the workload (i.e. workload shift), and as a result they can take poorly chosen actions. Though some of these works propose incremental retraining, the cost and the time it takes to retrain the models could be a hurdle towards adapting sufficiently quick to the changes in the workload. In contrast,

[2]Usually $T^* \in [60\%, 75\%]$.

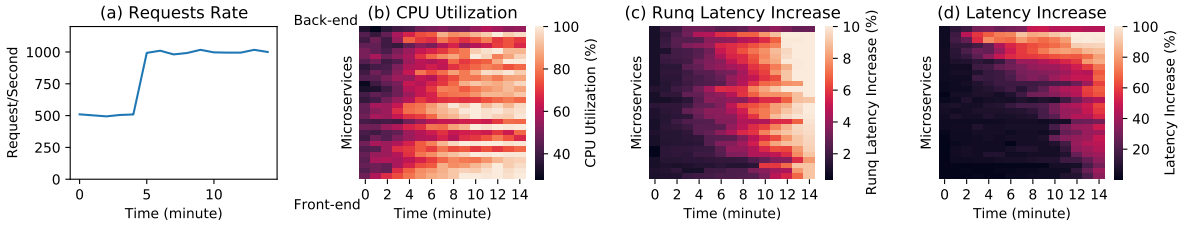

Figure 4.4. Heatmap of latency propagation from back-end microservices to the front-end microservices when there is a change in the workload. Lighter color denotes higher value (consult the individual color bars for exact value range). (a) Request arrival rate. The arrival rate increases at $t = 5m$ to make higher load on the application. (b) CPU utilization of each microservice from back-end tier (top) to front-end (bottom) tier. As the load increases, the CPU utilization of microservices in all tires increases. (c) The increase in *runq latency* of each microservice. As the load increases, the *runq latency* increases over time from back-end microservices to the front-end microservices. (d) The increase in request latency of each microservices. As the load increases, the back-end microservices face higher increase in their latency and propagates to the front-end microservices over time.

SHOWAR's autoscaler controllers adapt quickly to the workload changes since they use the recent resource usage statistics and workload changes as signals for their autoscaling actions.

Control Theory in Computer Systems. Control theory has been studied and used in computer systems by both academic and industrial communities.

[84] presents have controllers in general and PID controllers in particular are used at Amazon Web Services (AWS) for cluster autoscaling at scale.

In [62], the authors present the applications of control theory (and feedback PID controller in particular) for multiple computing systems including cache hit rate, Ads delivery system, and cluster autoscaling. Different types of PID controller design has been studied for autoscaling cluster of virtual machines and storage systems [85, 86]. Control theory has also been used for better power consumption in server clusters through autoscaling [16, 87, 88].

While the theme of all of these work is using some form of PID controllers in data centers (mainly for autoscaling), *SHOWAR* uses the PID controllers for autoscaling in another context that has not been applied before. In particular, in *SHOWAR*'s design, we use multiple PID controllers in a coordinated fashion to achieve resource efficiency. Additionally, all of the prior work mainly use the CPU utilization as the metric for the controllers signal, however, as we discuss in section 4.3, we use eBPF runq latency for the controllers signal.

4.3 *SHOWAR* Design

In designing *SHOWAR* we aim to bridge between vertical pod autoscaling, horizontal pod autoscaling, and pod scheduling and placement for efficient resource usage and good end-to-end performance. In the following, we describe the design of each part and discuss how they work in tandem.

4.3.1 Vertical Autoscaler

The state of the art of vertical autoscaling systems, specifically Kubernetes vertical autoscaler (and similarly Google Autopilot) [63, 89], take a conservative approach to set the CPU and memory requirements for microservices. In particular, by monitoring the historical resource (both CPU and Memory) usage in a past window of time (e.g. few minutes to several days), and then setting the resource allocation for the next window of time to be some percentile π of the usage and a safety margin α: $\pi(1+\alpha)$ where typically π is between 90^{th} and 99^{th} percentile and α is between 0.10 and 0.2. While this approach of taking a high percentile of the historical usage and adding some safety margin to it addresses the under-provisioning problem and minimizes the number of OOM errors as well as the time that the CPU is throttled, it still leaves the problem of underutilized and wasted resources which is not cost-effective for the application owner.

In *SHOWAR*'s vertical autoscaler we take a simpler approach while addressing both resource under- and over-provisioning and their implications. Particularly, *SHOWAR* uses a standard "three-sigma" rule-of-thumb to set the resource allocations of the microservices. To that end, the usage statistics of each resource type (CPU or memory) from the last window of duration W seconds (collected every second, see section 4.4) are used to recursively compute the mean μ and its variance σ^2 over that window. Then $s = \mu + 3\sigma$ is the estimated amount of that specific resource type currently needed by application. The amount s is evaluated every T seconds where $T \ll |W|$ and, if it has changed substantially (say more than 15%) since last evaluation, then the allocation is updated.

SHOWAR's vertical autoscaler simply prevents both under- and over-provisioning by taking the usage pattern's variance. Compared to the prior work (i.e. using $\pi(1+\alpha)$) where a fixed amount of resources is added to to its tail percentile usage, when there is a high variance in the resource usage, *SHOWAR* provisions the resource accordingly and hence prevents under-provisioning and performance degradation. In addition, when there is a low variance in the resource usage, *SHOWAR* does not over-provision and hence prevents over-provisioning and wasted resources.

To reiterate, while both $\mu + 3\sigma$ and $\pi(1 + \alpha)$ have clear statistical interpretation, the choice of using 3σ gives a more accurate view of the "spread" of the distribution about the mean. In particular, if the variance is very small, then the distribution is almost constant which is separately useful information about the resource usage of the Pods. However, in $\pi(1 + \alpha)$ method, when the variance is very small, the tail percentiles do not convey useful information. Additionally, the choice of the safety-margin hyperparameter α may be arbitrary which can result in poor resource utilization or more OOMs if it is not specified properly.

4.3.2 Horizontal Autoscaler

While being widely used in production, the state of the art of horizontal autoscaling systems [63, 90] suffer from a few shortcomings in their design. First, they *proportionally* react to the current autoscaling metric (i.e. CPU utilization) measurement (compared to the target metric value, i.e. observation minus target), and as a result they increase or decrease the number of service replicas to reach to the target metric value in a single shot and in a stateless fashion. This becomes problematic and inefficient when there are bursts and fluctuations in the load, which can result in extreme over-provisioning (so not cost effective) or under-provisioning (so performance degradation and poor quality of service), cf. section 4.5. To address such problems, some autoscaling systems introduce the notion of a cool-down period in which no autoscaling actions are performed for a period of time right after the most recent action[3]. While this can mitigate the number of abrupt autoscaling actions, a transient spike in the load can fool the autoscaling system to perform an unnecessary autoscaling action while missing the required autoscaling decision for the next time-window because of the cool-down period.

Second, these systems usually do not take the request's dependency graph for its microservices into the account and treat each microservice independently. Prior works have shown that resource allocation and autoscaling for microservices without considering correlations among dependent microservices results in inefficient resource allocation and does not necessarily help in coping with the load changes and maintaining a good performance [79, 91, 92]. To illustrate, we deployed the *Social Network* application to see how different microservices react to changes in the load by monitoring different performance metrics. As can be seen in Figure 4.4, as the load increases at time $t = 5m$ (in Figure 4.4a), the back-end microservices start experiencing high request 99^{th} tail

[3]Alternatively, one can add "hysteresis" to autoscaling rules to prevent too-frequent control actions.

latency (in Figure 4.4d) compared to normal operation. The high tail latency then progressively propagates from the back-end microservices to the middle tier and finally to the front-end microservice as time passes. An efficient horizontal autoscaling system would autoscale the back-end microservices first to prevent high tail latency propagation and possibly avoid unnecessary autoscaling for the front-end microservices.

Finally, previous approaches typically use CPU utilization as a metric for autoscaling decisions by striving to maintain a target CPU utilization across all the microservices. However, it's well-known that CPU utilization is not the most effective metric for autoscaling and resource allocation [22, 93–98]. As it can be seen in Figure 4.4b, as load increases, the CPU utilization increases for almost all the microservices, while the tail latency of front-end microservices does not always increase as CPU utilization increases. That is, the high CPU utilization does not always translate into high tail latency increase as there are microservices that have high CPU utilization but do not experience high tail latency. As a result, the state of the art autoscaling systems would perform unnecessary autoscaling actions for the microservices (front-end microservices for example) as they are not contributing to the request latency in a very significant way, cf. section 4.5.

In *SHOWAR*'s horizontal autoscaling design, we aim to address the shortcomings that we mentioned above. We propose to use a basic framework of control theory to design a stateful horizontal autoscaling system that maintains stability while meeting performance requirements as expressed by the metrics of SLOs.

The Autoscaling Controller. As mentioned above, reacting just proportionally to the observed autoscaling metric error,

$$e \;=\; \text{observation - target},$$

i.e., proportional control, is not enough and can result in poor and inefficient autoscaling decisions and instability owing to abrupt changes in the number of replicas. To improve this, we appeal to a more sophisticated controller: A *proportional–integral–derivative* (*PID*) controller [84, 99] which effects control based on a linear combination of three terms at time t:

$$u(t) = k_P e(t) + k_I \int_{t-w}^{t} e(\tau)d\tau + k_D \frac{d}{dt} e(t),$$

where, e.g., integration window can be set to $w = t$ in particular, and k_P, k_I, and k_D are the coefficients for the proportional (reacting to changes), integral (the memory for the past state), and derivative (predicting the future state) terms respectively [61, 62]. These

parameters are further discussed below.

Metric. As mentioned before, CPU utilization is not always the most effective basis for horizontal autoscaling. We propose to use more meaningful and low-level CPU performance metrics. In particular, we use eBPF Linux scheduler *runq latency* metric [100] which represents the time between when a thread is runnable, and the time when it acquires the CPU and is running. The longer the *runq latency*, the higher CPU contention among the threads and vice versa, therefore, as shown by prior work [92], this metric is a suitable choice for autoscaling purposes. In addition, from a control perspective, it is a metric that changes fast enough in response to an autoscaling action (i.e changing the number of replicas), as such stabilizes the system faster. Furthermore, this metric captures both changes in the incoming load for the service as well as the input signals (i.e., changing the number of replicas) from the autoscaling system controller. *Runq latency* is represented as a histogram of the latencies that threads experience. As a target point for the autoscaling system controller, we use, e.g., $\pi = 95$ percentile value of this histogram. The controller performs the autoscaling actions so that such target values for *runq latency* are met. To illustrate the effectiveness of *runq latency*, Figure 4.4c depicts the *runq latency* increase in microservices. As it can be seen, the *runq latency* shows a similar behavior to the request tail latency and it progressively increases and propagates from the back-end microservices to the front-end microservices. Unlike CPU utilization, the high *runq latency* is highly correlated with the high request tail latency for each individual microservice which shows that the *runq latency* can be used as a suitable metric for horizontal autoscaling to prevent request latency increase. Intuitively, the reason that *runq latency* is superior to CPU utilization is that it indicates how the application threads are *competing* for CPU resources and hence the need for more (or less) CPU resources [93].

Note that even though *runq latency* is a per thread metric, it is still applicable to single-threaded applications. This is because even in a single-threaded application, the thread can be interrupted by the OS and hence an increase in its *runq latency*. In addition, when the this metric is low for an application, and horizontal autoscaler cannot be triggered, the vertical autoscaler of *SHOWAR* can proceed and allocate resources for the Pod or trigger the horizontal autoscaler to scale the pod horizontally (see subsection 4.3.3 for more details).

High-level application latency SLOs are business decisions and generally may not directly translate to *runq latency*. However, as shown in Figure 4.4, the lower *runq latency* results in lower application latency and vice versa. As such, specifying a target

value for the *runq latency* here can result in meeting the target application latency SLOs. In current *SHOWAR*'s design and deployment, the user (i.e. the application owner) has to specify a target runq latency value as part of *SHOWAR*'s configuration. Since *runq latency* is an OS scheduler metric, the value for the target *runq latency* should not be higher than a few orders of magnitude of CPU scheduler time-quanta (where a quantum is configurable).

The Transfer Function The transfer function in our autoscaling case is simple and has the property that: If *runq latency* exceeds the target value, then the autoscaling system has to scale out and increase the number of replicas. Also, if *runq latency* is below the target value, then the autoscaling system has to scale in and decrease the number of replicas. The overall procedure is shown in Algorithm 1. To prevent performing too many autoscaling actions in response to fast changes as well as transient burstiness in the *runq latency* metric, we set a configurable bound α percent (20% by default) around the target value as a buffer and no autoscaling action is performed (i.e. NO-OP). The amount of increase or decrease in case of autoscaling is a configurable β percent (10% by default) of the number of current replicas of the microservice or is 1 if β is less than 1[4].

Algorithm 1: Autoscaling Transfer Function

```
1  M : microservice;
2  PID: pid controller for M;
3  R_M: Number of replicas for M;
4  α: PID's action bound;
5  β: Replica change step coefficient;
6  while True do
7  |   runqlatency = runq_sample_histogram(M);
8  |   observation = P95(runqlatency);
9  |   output = PID.output(observation);
10 |   if output > target * (1 + α/2) then
11 |   |   R_M = R_M + max(1, R_M * β);
12 |   else if output < target * (1 − α/2) then
13 |   |   R_M = R_M − max(1, R_M * β);
14 |   else
15 |   |   NO − OP ;
16 |   end
17 end
```

Tuning The Autoscaling Controller. Tuning the autoscaling PID controller refers to determining the values for the coefficients k [61, 84, 99]. Different values for the

[4]For most of the microservices at our evaluation scale, 10% of replicas had $\beta < 1$, cf. section 4.5.

coefficients can affect the performance of the controller in terms of speed (responsiveness), stability and accuracy. In particular, increasing k_P leads to an increase in the speed of the controller (to reach to a stable state), however high k_P values may correspond to instability which is a main problem of prior works relying only on proportional autoscaling control, e.g., [63, 90]. Increasing k_I increases the speed of the controller as well and may result in instability, but increasing k_I will lower the controller's noise (variation and fluctuations) and steady-state errors. Finally, increasing k_D increases the speed of the controller (to reach steady-state) as well as the potential for instability while amplifying the controller's noise profoundly. As a result, given the effect of the coefficients values on the controller's speed, stability and noise, a workload-aware and adaptive tuning approach is required.

Instead of using traditional and standard PID tuning methods [61, 99], we propose to use the following adaptive method mainly to cope with the variations in the incoming workload (i.e. workload shift). This is because failing to cope with the workload changes can result in poor resource efficiency or worse, service downtime due to insufficient resource allocations [59].

Initially, the controller starts with equal values for the coefficients. Subsequently, the coefficients are adaptively and incrementally self-tuned based on monitored workload performance and controller state. In particular, if the current metric value (especially *runq latency*) is far from the target metric value, k_P and k_I are increased in each iteration to improve stability as well as the speed at which the target metric value is reached. Also, if fluctuations in the metric value are observed (referred as noise in the controller), k_D is decreased gradually to reduce the noise introduced by the workload's burstiness.

Autoscaling Approach. As explained above, a fundamental problem with state of the art horizontal autoscaling systems is that they can perform unnecessary autoscaling actions that can result in poor resource utilization. To address this problem, in *SHOWAR*'s horizontal autoscaling design we take into the account two knobs: a) the sorted absolute values of the controllers' output for each microservice (cf. the "one for each design" case), and b) the (topologically sorted) dependency graph among the microservices. We propose two architectures for the horizontal autoscaling controller system:

- **One For All:** In this design, a single controller is responsible for autoscaling all of the microservice types. That is, at every autoscaling decision, all the microservices are scaled (up or down) at once depending on the average of current metric value observation across all of the microservices. While this approach benefits from the PID controller, it does not take into the account the microservices dependency

graph of the microservices.

- **One For Each:** In this design, a controller is responsible for each microservice. Each controller monitors the autoscaling metric (runq latency) of its corresponding microservice and performs the autoscaling action for that microservice (according to Algorithm 1), in a coordinated fashion as follows. The absolute value of controllers' outputs are sorted and those with the highest values (greatest scaling need) are prioritized. For equal controllers' outputs, we then take into the account the dependency graph of microservices, and prioritize the back-end services over the dependent front-end services (after a topological sort of the graph). In our benchmarks, we observe the back-end services typically are the ones with the highest controller's output values as well (see section 4.5). Motivated by the observations in Figure 4.4, once a microservice's controller has performed an autoscaling action, the controllers for all of its dependent microservices are postponed until the autoscaling is done (i.e. new replicas are added, or some of the replicas are removed) and then the dependent microservices' controllers try to perform autoscaling. This is because, in most of the cases, autoscaling the dependee microservice eliminates the need for autoscaling the dependent microservices and hence the latency increase propagation observed in Figure 4.4.

4.3.3 Tandem Vertical and Horizontal Autoscalers

The recommended approach for deploying vertical and horizontal autoscalers in state-of-the-art[5] platforms such as Kubernetes [89, 90] is to only deploy one autoscalers at a time to avoid interference from others. That is, in case of choosing the vertical autoscaler, the developers would set a fixed number for the number of replicas (instances) for each microservice and the vertical autoscaler will scale up or down the size of Pods running the microservice. On the other hand, if a horizontal autoscaler is chosen, the developers would, for each microservice, set a fixed size (CPU and memory), and the horizontal autoscaler will scale up or down the number of replicas for each microservice.

SHOWAR benefits from both vertical and horizontal autoscaling by allowing deploying them in tandem. First, we prioritize any vertical autoscaling decision over any horizontal autoscaling decision. We do this because, in case of memory autoscaling for example, if there is insufficient memory for the Pod, the application encounters an out of memory (OOM) error and stops the execution regardless of its replicas count, as such, the horizontal

[5]Google Autopilot allows deploying both vertical and horizontal autoscalers at the same time.

autoscaler cannot address the problem. Therefore, before a horizontal autoscaling controller acts, it first checks a shared channel to see if a vertical autoscaling is in progress for that microservice and, if so, it will not proceed. Similarly, before a vertical Pod autoscaler acts, it sends a message over the shared channel notifying the horizontal autoscaler and then performs its action.

Second, according to Google Cloud Platforms' Kubernetes best practices, due to lack of parallelism, it's recommended that for most of the workloads, no more than one core (i.e. $1000m$ core in kubernetes currency) is needed for each Pod [101]. We use this recommendation and incorporate it into the *SHOWAR*'s vertical autoscaler design. That is, if the vertical autoscaler decision is to set more than one core for a Pod, it signals the horizontal autoscaler through a shared channel instead, and will not proceed with the vertical autoscaling action.

These two mechanisms allow the vertical and horizontal autoscalers to be deployed at the same time and work in tandem for efficient resource allocation while maintaining the target performance goals.

4.3.4 (More) Efficient Scheduling

Again, the general aims of vertical and horizontal autoscaling are twofold: a) efficient resource usage and allocation while b) maintaining a good performance for the microservices. However, in addition to autoscaling, another factor that helps in efficient resource usage and meeting performance goals for the microservices is scheduling and placement of the microservice Pods on the nodes (VMs) in the cluster. In particular, the scheduler would place as many Pods as possible on a node to improve the resource usage once the Pods are right-sized both vertically and horizontally by the autoscalers. On the other hand, "tight" (efficient) placement of the Pods into nodes can result in resource contention (due to noisy neighbor effect) hence degrading the performance of Pods as well the microservice as a whole.

To bridge the gap between scheduling and autoscaling, and to improve the resource efficiency as well as the performance of the microservices even further, *SHOWAR* provides "hints" (or rules) for the Kubernetes scheduler's by generating inter-pod affinity and anti-affinity rules. An affinity of service S_2 for service S_1 implies that the scheduler will always (or preferably) try to schedule the Pods of service S_1 on the nodes which Pods of service S_2 reside on. Similarly, an anti-affinity of service S_2 for service S_1 implies that the scheduler will never (or preferably not) do this.

In doing so, *SHOWAR* monitors and uses the historical (i.e. last (configurable)

window of time) CPU, memory, and network (both out and in) usage of microservices and calculates the Paerson correlation coefficient [102] between each pair of microservices' usage pattern: Given the distribution of CPU (or memory or network I/O) usage of two microservice types X and Y, the correlation coefficient ρ between X and Y is:

$$\rho_{XY} = \frac{\sum_{(x,y)\in S} \sum (x - \mu_X)(y - \mu_Y)}{\sigma_x \sigma_y} \in [-1, 1],$$

where μ_X and σ_X are the (historical) sample mean and sample variance of X, respectively, each computed by standard recursive means.

For two microservices S_1 and S_2, the higher positive correlation in the usage pattern of a resource (say CPU or memory), the higher resource contention for that resource between them. Similarly, the lower the negative correlation, the lower the contention between the two services for that resource. This is the simple basis of *SHOWAR*'s affinity and anti-affinity rules for the compute resources such as CPU, memory, and network I/O.

Generating Affinity and Anti-Affinity Rules. Specifically, the mechanism for generating affinity and anti-affinity rules for the scheduler based on the correlation coefficient for each resource type is as follow:

- **CPU and Network:** For any pair of microservices S_1 and S_2, if they have a *strongly negative* correlation in their CPU and network I/O usage pattern (i.e. $\rho_{S_1S_2} \leq -0.8$) *SHOWAR* generates an *affinity* rule of S_2 for S_1 for the scheduler. This is because CPU and network bandwidth are resources that can be shared and throttled, and, as such, even if the usage pattern of the two service change and the negative correlation does not hold, the microservices may be able to effectively share such resources.

- **Memory:** If any pair of microservices S_1 and S_2 have a *strongly positive* correlation in their memory usage pattern (e.g. $\rho_{S_1S_2} \geq 0.8$) *SHOWAR* generates an *anti-affinity* rule for S_1 and S_2 for the scheduler. This is because when the memory usage pattern of two services is strongly correlated, given the limited physical memory bandwidth on the node, the microservices can suffer from insufficient memory bandwidth.

Note that an anti-affinity rule has a symmetric property that can completely prevent the scheduler from scheduling two strongly memory correlated microservices. In other words, if there is an anti-affinity rule of S_2 for S_1, unlike an affinity rule, when scheduling Pods of microservice S_1, the scheduler not only checks the presence of Pods of microservice

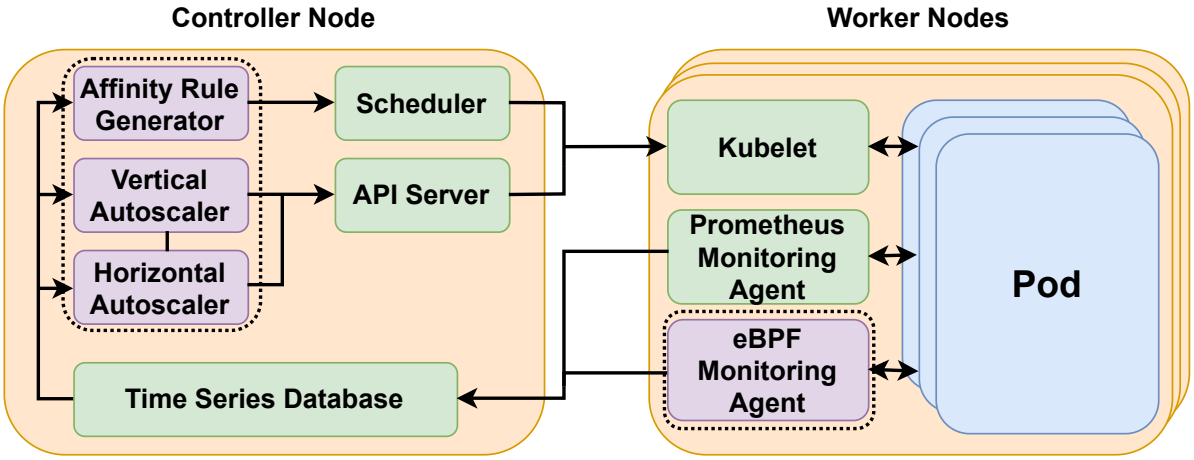

Figure 4.5. *SHOWAR* Architecture Overview. The resource usage logs as well as the eBPF metrics are collected using their corresponding agents on each node and are aggregated into the time series database. *SHOWAR* uses the collected metrics to make autoscaling decisions as well as scheduling affinity and anti-affinity rules by communicating with the Kubernetes API server and its scheduler respectively.

S_2, but also, when scheduling Pods of microservice S_2, checks the presence of Pods of microservice S_1 to not co-schedule/locate them on the same node (even though there is no affinity rule set of service S_2). As a result, to not make any scheduling conflicts for the scheduler, *SHOWAR* generates at most one affinity or anti-affinity rule for each microservice. That is, each microservice participates in at most one affinity or anti-affinity rule at any point in time.

4.4 *SHOWAR* Implementation

SHOWAR is implemented in GoLang as a cloud native programming language and consists of a set of modules plugged into the state of the art container orchestrator Kubernetes. Figure 6.1 depicts a high-level overview of the architecture and how *SHOWAR* is interacting with the kubernetes scheduler and its API server. *SHOWAR* is deployed as a service on the controller node and interacts with the kubernetes API server and its scheduler for autoscaling actions as well as applying the generated affinity and anti-affinity rules for the microservices.

Monitoring Agents. The monitoring and logging data are the most essential part of any application deployment. The monitoring data are used for observability, health check and autoscaling. We use the state of the art monitoring and metric collection tool Prometheus [103] to collect different metrics from nodes and containers. Prometheus

launches a monitoring agent on each node in the cluster to collect the container metrics such as CPU usage, Memory usage, Network bandwidth usage, etc. The agents are configured to collect and report the metrics every second (One second is the minimum period that Prometheus agents can collect the metrics. To obtain as many as data points possible, we collected data every second.). Prometheus comes with a time series database where the agents store the collected metric. In addition, a query language is provided to query the time series database which is used by the other modules to utilize the collected metrics.

In addition to the Prometheus standard metric collection agents, we have developed an eBPF program that is deployed as monitoring agent on every node in the cluster to collect the *runq latency* metric used by the horizontal autoscaler. This metric is a histogram of latencies that the CPU threads in each pod experience before acquiring the CPU. The program collects a histogram of *runq latencies* every 1 second and stores it in the Prometheus time series database.

The Vertical Autoscaler. The vertical autoscaler is a simple loop that takes place every minute[6]. That is every minute, it evaluates $s_r = \mu_r + 3 * \sigma_r$ over a window of the previous 5 minutes for each resource type r (CPU and memory) and if the value of s has changed by more than 15%, it updates the resource requirements of the service to be s. Another condition that triggers the vertical autoscaler is when a microservice reports an OOM error. Before applying the new resource requirements of the microservice, the vertical autoscaler sends a message over a shared channel to the horizontal autoscaler to not proceed with any horizontal autoscaling action as vertical autoscaling actions are prioritized over horizontal autoscaling. The vertical autoscaler also does not proceed with an autoscaling action for a microservice if the amount of CPU for that microservice is more than one CPU core (i.e. $s_{CPU} > 1000m$), in that case, it sends a message over another shared channel to the horizontal autoscaler to trigger a horizontal autoscaling action.

The Horizontal Autoscaler. As described in subsection 4.3.2, at its core, the horizontal autoscaler is a PID controller that aims to keep each microservice stable. That is, for a given target *runq latency*, it performs horizontal autoscaling actions for that microservice such that it always has a *runq latency* of the target value. The controllers make decision every 1 minute which the eBPF program collects 60 instances of the metric histogram (1 every second). For each histogram, the 95^{th} percentile is picked and the controller uses

[6]The decision frequency is configurable. We chose a frequency of once per minute since one minute is about the minimum amount of time over which an ample amount of metrics data are available to well inform an autoscaling decision.

the average of these 60 data points as its current observation (a.k.a measurement) to perform its controlling action. Each horizontal scaling action adds or removes at least 1 or a configurable percentage (10% by default) of current number of replicas of the microservice for scaling in and out respectively.

Recalling from section 4.3, the initial values for the PID control parameters are taken to be $k_P = k_I = k_D = 1/3$ (each parameter constrained to be $\in [0, 10]$). Incremental changes to these parameters is 10% (we found experimentally that 10% gives very good performance). Fluctuations in the controller's output, which are a basis to make such changes, is measured using the previous $N = 10$ samples. Also, the "speed" of the controller is measured as the number of iterations required to reach the interval $[\text{target}(1-\alpha), \text{target}(1+\alpha)]$ for $\alpha = 10\%$.

The Affinity Rule Generator. *SHOWAR*'s affinity rule generator uses the CPU, memory, and network utilizations every 5 minutes which is a vector consisting of 300 data points (each data point is the average over the microservice replicas) to compute the correlation coefficient of different resource types between every pair of microservices. Too eliminate the weak or no correlation instances, any value in $[-0.8, +0.8]$ is dropped. The other strongly negative and strongly positive correlated microservices are used to generate the affinity and anti-affinity rules as explained in subsection 4.3.4. The resource usage patterns can change as the workload changes (also known as workload shift), so if a strongly negative or positive correlation change by more than 20% (configurable) in a subsequent 5 minutes window of time, *SHOWAR* revokes the affinity (or anti-affinity) rule for that pair of microservices.

***SHOWAR*'s overhead.** Note that *SHOWAR* is built as a controller for the Kubernetes which is highly pluggable for autoscalers and other types of controllers [104]. In addition, *SHOWAR* uses the commonly used Kubernetes monitoring agents (e.g. Prometheus [103]) and one custom eBPF metric monitoring agent. As such, compared to the default Kubernetes autoscalers, *SHOWAR* does not introduce any additional overhead. Furthermore, the autoscalers are scheduled on the controller node and do not share resources with the application Pods which are scheduled on the worker nodes.

4.5 Evaluation

4.5.1 Experimental Setup

Applications. We evaluated *SHOWAR* using 3 interactive microservice applications: a) *Social Network* from DeathStarBench [65], an application consisting 36 microservices in which users can follow others, compose posts, and read and interact with others' posts; b) *Train-Ticket* [105], an application consisting 41 microservices which allows its users to reserve online tickets and make payments; and c) Google Cloud Platform's *Online Boutique* [106], consisting of 10 microservices in which users can purchase online items through their online cart and make payments. Through our experimental evaluation, we observe that the results are consistent across all 3 application benchmarks, here we only report the results for the *Social Network* application. We set the target value for *runq latency* to $15ms$ which is $2.5x$ the Linux kernel *sysctl_sched_latency* [107] scheduler parameter.

Cluster Setup. All of our evaluations are performed on Amazon Web Services (AWS) Cloud in *us-east-1* region. We use $m5.xlarge$ VM instances each with 4 vCPU, 16 GB of memory and $\$0.192/hr$ price, running Ubuntu 18.04 LTS configured for supporting running eBPF programs. Unless otherwise mentioned, our cluster consists of 25 VM instances.

Workload and Load Generation. We use Wikipedia access traces [108] as our primary workload. It's a real-world trace of users interacting with the Wikipedia website consisting of traffic patterns including periods of Poisson arrival times, short-term burstiness, and diurnal level-shifts. Since the microservices that we are evaluating are user-facing applications, the workload has to reflect realistic user behavior. As such, the Wikipedia access trace is a good fit for our evaluation. We use locust [109] as our workload generator in a distributed fashion. The locust clients reside on different VM instances than the main cluster that is hosting the application.

Baselines. We compare *SHOWAR*'s performance with two main baselines: a) Kubernetes default autoscalers [64] and b) Google Autopilot [63]. We implemented a version of Google Autopilot moving-window vertical autoscaler as described in [63] – its ML-based version is not used as it has not yet been open-sourced and not enough information has been disclosed to re-implement it.

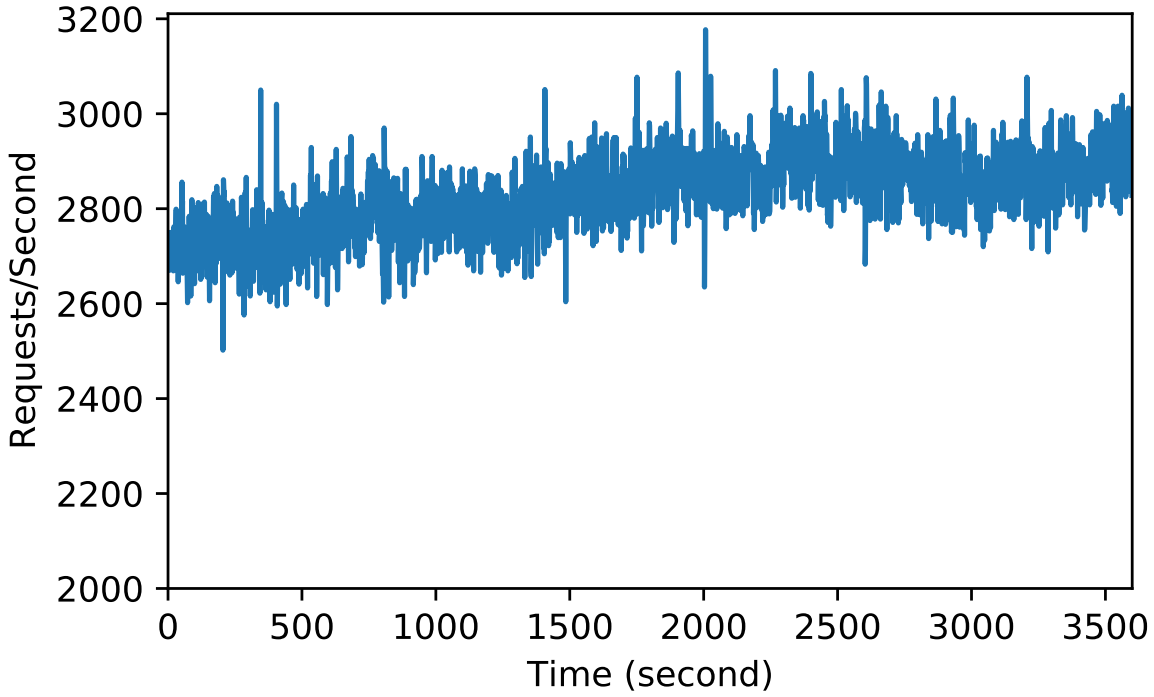

Figure 4.6. A one-hour long workload from Wikipedia access trace.

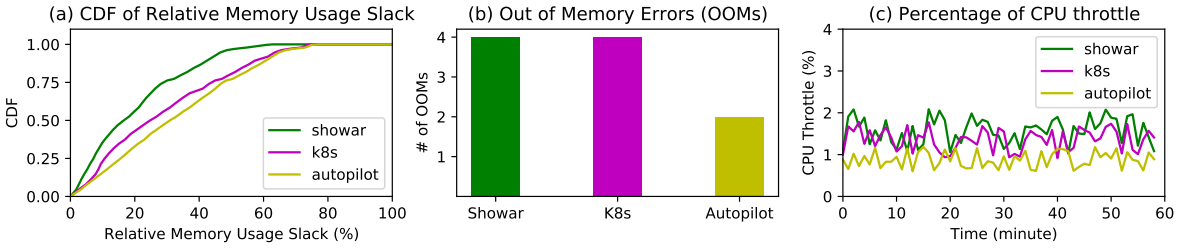

Figure 4.7. Vertical Autoscaling: (a) CDF of relative memory usage slack. (b) Number of Out of Memory (OOM) Errors. (c) Average CPU throttling across all the microservices.

4.5.2 Vertical Autoscaling

We first evaluate the effectiveness of *SHOWAR*'s vertical autoscaler (horizontal autoscalers are disabled) in reducing the relative memory slack. Our resource of interest here is memory because an insufficient allocation of memory for a service can result in out-of-memory errors which affect service availability, but CPU can be throttled and keeps the service available at the cost of degraded performance. We use a one-hour long workload from the Wikipedia access trace shown in Figure 4.6 for our evaluation. We log the memory limit set by the vertical autoscaler for each microservice as well as the microservice's actual usage every 5 minutes to calculate its memory usage slack (i.e.,

slack = limit - usage). Figure 4.7*a* depicts the cumulative distribution function (CDF) of relative memory usage slack (i.e. *slack/limit*) across all the microservices and their replicas in *Social Network application*. As can be seen, by embracing the variance of past resource usage (using three-sigma rule), *SHOWAR*'s vertical autoscaler is able to improve the memory usage slack compared to Autopilot's and Kubernetes' vertical autoscaler which use max (maximum) and P90 $\times$ 1.15 (15% more than the 90$^{\text{th}}$ percentile) of past usage respectively. In particular, for 95% of the service instances, the relative memory usage slack is less than 46% compared to 63% and 66% for Kubernetes and Autopilot respectively. This 20% savings in memory usage slack can be utilized for scheduling more instances of services or using less VM resources in the cluster which will obviously reduce costs (see subsection 4.5.5). We also observe that Kubernetes outperforms Autopilot as it has a more aggressive approach in setting the limits (using P95 $\times$ 1.15 of past usage compared to the max).

While low memory or CPU usage slack can result in efficient and cost-effective resource allocation, it can however result in higher rates of OOMs or throttled CPU and hence degradation in service performance. Figure 4.7*b* shows the number of OOMs over the course of the experiment. As can be seen, while *SHOWAR* has comparable number of OOMs compared to Kubernetes, their aggressive approaches in memory scaling result in more OOMs compared to the Autopilot. In Figure 4.7*c* we depict the average CPU throttling (result of tight CPU slack) across the microservices during the course of the experiment. When the CPU usage of a Pod exceeds its allocated CPU resources, the container runtime (using *cgroups*) throttles the CPU share of the Pod. As can be seen, *SHOWAR* has a CPU throttling comparable to the baselines, because of the high fluctuation (variance) in the CPU usage of the microservices.

As can be seen in Figure 4.7, there is a natural trade-off between resource efficiency (i.e. lower slack) and stability. *SHOWAR* and Kubernetes result in better resource efficiency while resulting in higher number of OOMs (and throttled CPU), while Autopilot results in higher slack and less number of OOMs. Depending on the objectives, one can tune *SHOWAR* and Kubernetes to achieve higher stability at the cost of higher resource usage slack. For example, in *SHOWAR*, instead of the 3σ term, one can use $k\sigma$ where $k > 3$ to allocate more resources for individual Pods and mitigate OOMs and CPU throttling.

4.5.3 Horizontal Autoscaling

Here, using the same workload from Figure 4.6 we evaluate the effectiveness of *SHOWAR*'s horizontal autoscaler (vertical autoscalers are disabled) in determining the right number of replicas for microservices. We compare both *SHOWAR*'s *One for Each* and *One for All* designs with Autopilot and Kubernetes horizontal autoscalers. Recall from subsection 4.2.4 that both Autopilot and Kubernetes use the same approach in horizontal autoscaling. We set the target CPU utilization for Autopilot and Kubernetes to 65% as it is commonly recommended.

Figure 4.8 depicts the cumulative distribution function of the number of replicas of microservices in *Social Network* application over the course of the experiment. We observe that both *SHOWAR*'s horizontal autoscalers outperform the Autopilot and Kubernetes horizontal autoscalers by allocating fewer replicas for the majority of microservices, which in turn can result in more efficient resource allocation and cost savings (see subsection 4.5.5). By having a tailored controller for each microservice, *SHOWAR*'s *One for Each* design also outperforms its *One for All*. This is because in the *One for All* design, a single controller tries to scale the microservices using a single target *runq latency* value and an averaged *runq latency* measurement across all the microservices which results in unnecessary scaling of microservices that do not have, high *runq latency*. In addition we observe that, as expected, Autopilot and Kubernetes have almost identical horizontal autoscaling decisions because they use the same method for horizontal autoscaling using the same target CPU utilization. The slight difference between the two comes from the variations in CPU utilization measurements during the experiments.

To reiterate, the effectiveness of *SHOWAR* is due to a) a stateful controller for the autoscaler and b) a better representative metric (i.e. *runq latency* instead of CPU utilization) for autoscaling decisions. The effect of being stateful and having memory of the past autoscaling actions can be seen in Figure 4.8 where *SHOWAR* allocates a smaller number of replicas for the majority of the microservices as compared to the baselines which are stateless, memoryless and reactive. More specifically, 95% of microservices have less than 9 replicas using *SHOWAR* compared to 15 replicas using Kubernetes and Autopilot. In addition, we see the effect of using more meaningful and representative metric in autoscaling decisions for individual microservices. In particular, during our evaluation, we observed that both Kubernetes and Autopilot typically set 16 replicas for *nginx* (a front-end microservice) primarily because of its high CPU utilization. However, as seen in Figure 4.4, a high CPU utilization does not always correspond to highly improved microservice performance. In contrast, *SHOWAR* sets only 10 replicas for

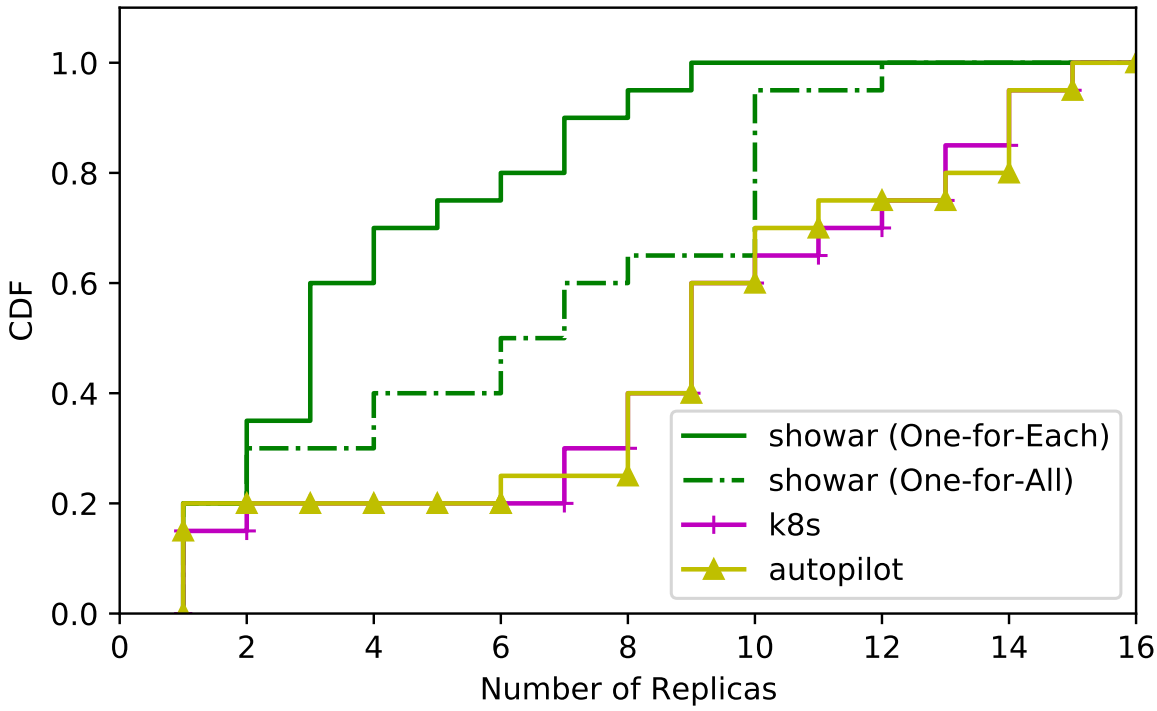

Figure 4.8. Horizontal Autoscaling: CDF of number of replicas across microservices.

this microservice. On the other hand, for the *User* microservice that several other microservices depend on it, both Kubernetes and Autopilot typically set only 3 replicas for it. In contrast, *SHOWAR* typically sets 6 replicas for this microservice.

4.5.4 The Effect of Affinity and Anti-Affinity Rules

Here, we evaluate the effect of Pod affinity and anti-affinity rules generated by *SHOWAR* using the correlation of CPU, memory, and network I/O usage between different microservices. We use the workload from Figure 4.6 and disable both vertical and horizontal autoscalers to see how our the generated affinity and anti-affinity rules can affect the Kubernetes' scheduler decisions on microservices and its effect on the latencies of users' requests compared to a situation where the scheduler does not use any affinity and anti-affinity rules. Due to space limits, we don't show the correlation data between the different microservices over the course of the experiments. Figure 4.9 depicts the CDF of the end to end user request's latencies. As it can be seen, by providing scheduling hints (using affinity and anti-affinity) for the scheduler, *SHOWAR* is able to improve the P99th latency that the users experience. In particular, using the affinity and anti-affinity rules generated by *SHOWAR*, the P99th of request latency is 6600 milliseconds compared

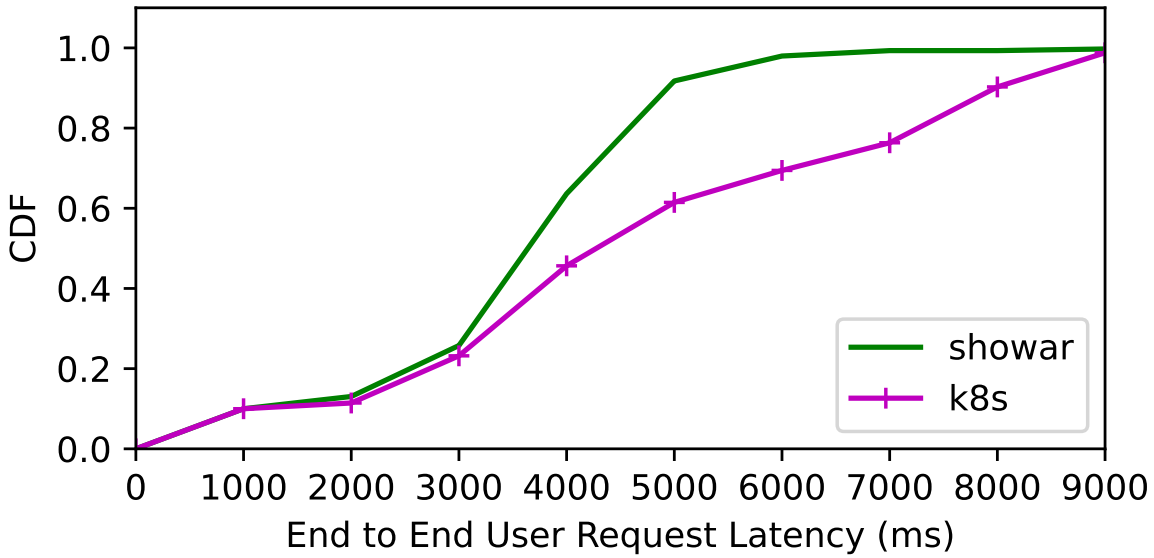

Figure 4.9. Affinity Rule Generator: CDF of user-experienced end-to-end P99 latency in presence of *SHOWAR* affinity rule generator for CPU, Memory, and Network I/O compared to Kubernetes default scheduler.

to 9000 milliseconds using Kubernetes default scheduling decisions.

4.5.5 End-to-End performance

While we evaluated each component of *SHOWAR* individually in the past three subsections, here we enable all the three components to work in tandem and perform an end-to-end evaluation. We use a 24-hour long workload from the Wikipedia access trace shown in Figure 4.10 and hence each experiment lasts for 24 hours to capture all the patterns in the (real-world) workload. To fit the workload, we increased the size of our cluster to 30 VM instances. Our results show that *SHOWAR* improves the resource allocation and utilization while maintaining a comparable performance compared to the baselines.

Figure 4.11a depicts the CDF of the end to end request latency experienced by the user during the 24 hours of the experiment. As it can be seen, the end to end performance using *SHOWAR* is comparable to the baselines and using its affinity and anti-affinity rule generator as well as its dependency-aware horizontal autoscaling, *SHOWAR* is able to improve the P99th latency by more than 20% compared to the Autopilot and Kubernetes. Both Autopilot and Kubernetes show similar performance in P99th latency, however, because of allocating more memory for the replicas, Autopilot generally outperforms the

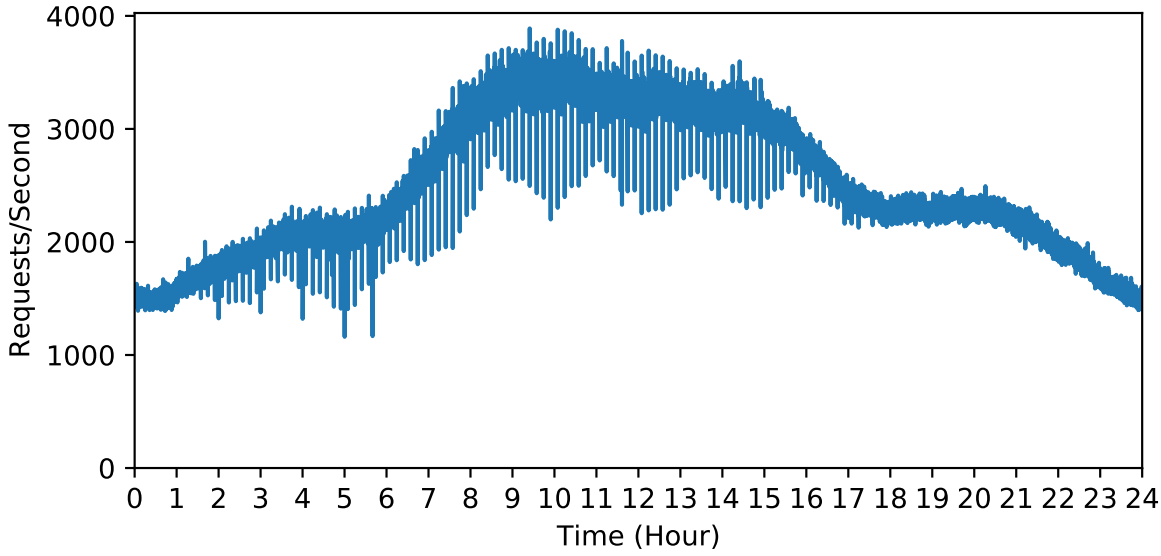

Figure 4.10. A 24-hours long workload from Wikipedia access trace.

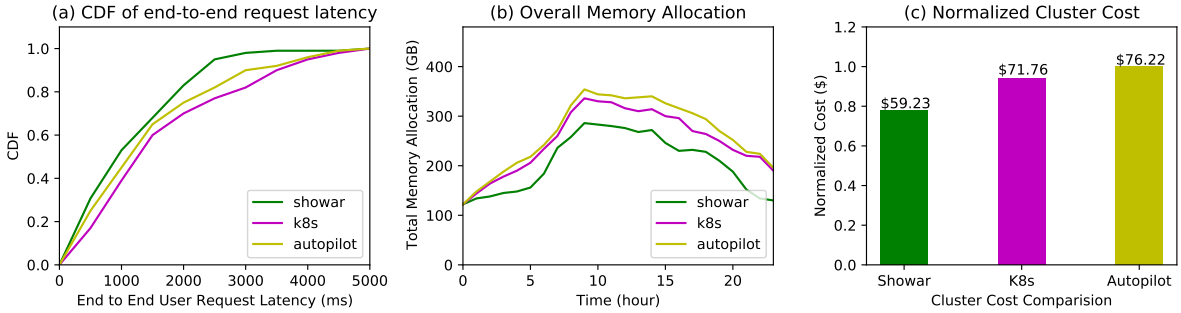

Figure 4.11. End-to-End Performance: (a) CDF of End-to-End Request Latency. (b) Total Cluster Memory Allocation. (c) Normalized Cluster Cost

Kubernetes at lower tails.

Figure 4.11b shows the total memory allocation (i.e. sum of memory limits set for the microservices replicas) in the cluster during the course of the experiments. Compared to the baselines, *SHOWAR* allocates less memory for the microservices replicas on average. In particular, on average, *SHOWAR* allocated 205 GB, while Autopilot and Kubernetes allocated 264 GB and 249 GB respectively. As it was seen in subsection 4.3.1 and subsection 4.3.2, it is mainly because *SHOWAR*'s vertical autoscaler achieves lower memory usage slack and also its horizontal autoscaler sets a lower number of replicas for the microservices. As such, the total memory allocation using *SHOWAR*is less than the baselines.

Finally, in Figure 4.11c we show the normalized cluster cost for each experiment. We

normalize the average memory allocation to the memory size of one virtual machine in the cluster (i.e. 16 GB for $m5.xlarge$ instances) and multiply it by the cost of one virtual machine (i.e. \$0.192/$hour$) in 24 hours. This is because, usually the VM's price on public clouds is a linear function of memory size [110]. As it can be seen, compared to the Autopilot and Kubernetes, *SHOWAR* improves the total cluster cost by 22% and 17% respectively. The improvements come from the fact that *SHOWAR*'s vertical and horizontal autoscalers allocate less amount of compute resources at a comparable performance compared to the baselines.

4.6 Limitations and Future Work

Generally, we designed *SHOWAR* to be computationally light weight and adaptable, in contrast with "black box" approaches that use machine learning that need training and fail to cope with workload shift, e.g., [63, 79, 91].

Nevertheless, a major limitation of *SHOWAR* currently is that it is reactive to the resource usages of the microservices. As a result, a proper avenue to explore is to equip *SHOWAR* with near-term workload and resource usage prediction, e.g., [111]. Combined with it's current design, predicting the near future workload can improve the *SHOWAR* 's resource allocation and prevent performance degradation due to inadequate autoscaling actions.

Another limitation that current *SHOWAR*'s design has is that it only focuses on microservices autoscaling and assumes a fixed-sized cluster. It's important to address scenarios where the total amount of resources that the application autoscaler requests is more than the total available cluster resources. While cluster autoscaling is orthogonal to application autoscaling, they need to work together to achieve both overall efficiency in resource allocation and the application's performance requirements. As such, a communication and coordination between the two autoscalers is required to add more resources to the cluster. In future work, we plan to improve *SHOWAR*'s autoscalers to work with existing cluster autoscalers [112].

SHOWAR is designed as an autoscaler for Kubernetes. As such, it can be used for autoscaling the workloads that are supported by Kubernetes such as user-facing microservices and long-running batch jobs. However, while Kubernetes supports short-running jobs such as serverless functions [113], autoscalers such as *SHOWAR* may not be usable for this type of workload. One reason is that vertical autoscaling is not applicable because the size of containers for serverless functions are predefined.

SHOWAR's horizontal autoscaler may face additional complexity, e.g., keeping track of the number of "dormant" serverless functions (which can be warm started) and the time until each of them "expires" (and so would require a cold start delay). We leave exploring control-theoretic approaches for horizontal scaling of serverless functions to future work.

Finally, we plan to improve the *SHOWAR*'s affinity and anti-affinity rule generators. Currently we determine affinity pairwise between microservices using simple empirical resource-utilization correlation coefficients. We can in the future explore, for example, the impact on affinities of other statistics such as cross-correlations between different types of resources, and explore different types of scheduling mechanisms that can exploit such "raw" statistical information directly toward more efficient resource utilization [114].

4.7 Conclusion

In summary, we propose *SHOWAR* a framework consisting of a vertical autoscaler, a horizontal autoscaler and a scheduling affinity rule generator for the microservices. *SHOWAR*'s vertical autoscaler embraces the empirical variance in the historical resource usage to find the optimal size and reduce the resource usage slack (the difference between allocated resource and actual resource usage). For horizontal scaling *SHOWAR* uses ideas from control theory along with kernel level performance metrics (i.e. eBPF *runq latency*) to perform accurate horizontal scaling actions. In particular uses proportional–integral–derivative controller (PID controller) as a stateful controller to control the number of replicas for each microservices. The vertical and horizontal autoscalers in *SHOWAR* work in tandem to improve the resource utilization while maintaining a good performance. Additionally, once the size for each microservices is found, *SHOWAR* bridges the gap between optimal resource allocation and scheduling by generating affinity hints for the task scheduler to further improve performance. Our empirical experiments using a variety of microservice applications and real-world workloads show that, compared to state of the art autoscaling systems, on average *SHOWAR* improves the resource allocation by up to 22% (and hence saving in costs) while improving the 99th percentile end-to-end user request latency by 20%.

Chapter 5
Attack-Resilient Microservices

5.1 Introduction

Generally, Distributed Denial of Service (DDoS) attacks, particularly widely publicized volumetric ones, continue to grow in scale [115] and cost [116]. In response to these kind of attacks, many solutions have been proposed both in academia and industry. Solutions for volumetric attacks such as Akamai Prolexic and AWS Cloudflare do not address low-volume attacks targeting the application-layer (asymmetric DoS attacks) which exploit application-layer vulnerabilities to exhaust computing resources.

Examples of these low-volume application-layer attacks include Slowloris [117] and BlackNurse [118, 119]. Slowloris employs numerous partially opened HTTP requests, thereby exhausting a server's concurrent-connections store, and so thereafter denying connections to legitimate requesting clients. BlackNurse is a ping-flood attack variant using (unsolicited) ICMP Type 3 (destination unreachable) Code 3 (port unreachable) reply packets whose processing overloads CPU resources of common server firewalls. Other types of application-level attacks include regular expression DoS (ReDoS) attack [120] where the attacker tries to exploit the regular-expression parsing libraries by passing bad regular expressions causing the program to enter an infinite loop and hence exhaust CPU and memory resources [121]. In the Billion Laughs (XML bomb) attack [122], the attacker passes a small-size nested XML file to cause the XML-file parser to run out of memory. A recent example of this attack exploited a vulnerability in kubernetes' YAML file parser which enables an attacker to bring down a whole cluster using a single request containing a bad YAML file [123]. For another example, the Event-handler Poisoning (EHP) attack [124], the attacker causes the event handling thread to become so busy that other requests don't get a chance to be serviced, hence greatly increasing their response times. Recently, a suite of eight DoS attacks targeting HTTP/2 servers was published by

Netflix [125]; these attacks involve variations of known exploits. Though NGINX has released patches for most of the attacks targeting HTTP/2 mentioned above, new ones will likely arise.

Even more subtle attacks involve “heavy hitter" clients consistently issuing abnormally heavy workloads, where the individual workloads in isolation are innocuous.

Microservices are one of the targets of the application-layer attacks. Here, the attacker can target one specific service to exhaust resources available to it and hence that service is unavailable to other services and legitimate users [123, 126].

In response to the application-layer attacks against microservices, this work describes an **unsupervised**, **non-intrusive** and **application agnostic** approach that will be able to combat new attacks by directly monitoring the resource usages of the services under attack and isolating the attacker(s) so it cannot affect the availability of the service for the legitimate users.

Our approach leverages the microservice architecture to effectively detect and defend the attack. Microserivce-based applications are deployed on a multi-node cluster which is managed by a cluster manager and container orchestrator such as kubernetes (cf. Figure 5.1). Different services are loosely coupled to each other. Each service consists of one or more containers (“pod" in kubernetes terminology) that run the service. For higher availability and scalability, there are more than one service replicas. Since each service is separated from other services, we can detect and defend the attack against individual services (and their replicas) and hence prevent availability or performance degradation of the other service replicas or the entire application (consisting of all of the services, of course).

The basic idea of our approach is to monitor the resource usage of each service (pod) in last (configurable) T seconds and predict its usage in next T seconds. By using the knowledge from the historical data, we identify whether the usage is abnormal (in respect to known past behavior) or not. If so, the services is deemed to be under attack. (Thus, our approach is unsupervised because we do not rely on any model of the new attack behavior for purposes of anomaly detection.) We then quarantine the service on a reserved quarantine node in the cluster and perform fissioning to better isolate and identify the attacker (reduce false positives in detection). Because our approach directly detects anomalously high resource utilization (the goal of all such low-volumetric application-layer attacks), it is both non-intrusive and application agnostic (and unsupervised).

To show the effectiveness of our approach, we built a prototype (see 5.4) on kubernetes, a widely used and the state-of-the-art cluster manager and container-orchestration

platform for microservices. Through experimental evaluation of our prototype we show its effectiveness, e.g., in reducing the latency by 3× for legitimate users of an application that is under the attack compared to a situation where there is no attack detection and defense tool deployed on the cluster.

In summary, we make the following contributions:

- We propose a method to effectively detect application-layer attacks against microservices. The method is agnostic to attacks (unsupervised detection), nominal application/workload type, and service type.
- We propose a service fissioning mechanism to effectively mitigate the effect of attack on legitimate users by isolating the attacker.
- We built a prototype on the kubernetes cluster manager. Performance results using our prototype show the effectiveness of our proposed methods for attack detection and response.

The rest of this chapter is organized as follows: In Section 5.2, we discuss related prior work on defense against application-layer DoS attacks. In Section 5.3, we motivate and formulate our problem. In Section 5.4, We explain our design and prototype implementation on kubernetes. In Section 5.5, we give some preliminary experimental results using our prototype on AWS. Finally, we conclude in Section 5.6 with a discussion of future work that we plan to extend our prototype.

5.2 Related Work

DoS attack detection and defense has been studied extensively in the past. In this section we only focus on prior work on application-layer DoS attacks.

Finelame [127] uses (*supervised*) K-means clustering to characterize resource utilization behavior. In order to monitor the resource utilizations, they use eBPF [128] to get fine-grain monitoring at run time (OS calls). Though now eBPF is supported by cluster managers, it opens-up attack surfaces if it's not developed and deployed carefully and hence deployment in production is typically avoided. They also require programmers annotation of the application source-code regarding required resources to insert the probes to collect the utilizations statistics, which is not always possible for a deployed application in a production environment. In contrast, we use a completely **non-application-intrusive** and unsupervised approach for a multi-node cluster (VMs or containers/pods

running on them) and do not rely on information from tools such as eBPF. Rather, we use the container-level statistics which are available on any cluster manager, which does not need programmers' involvement. Also, Finelame is limited to only the detection phase. That is, the system only detects the attacker requests and it does not do defense beyond that. In our proposed work, however, we also address attack response. Finelame's scope is also limited to a traditional single-node deployed application, while the scope of our work targets **multi-node clusters** where a **distributed microservice** application may be deployed.

Other recent efforts on application-layer DoS attack detection and defense include Rampart [129] which is focused on DoS attacks against PHP applications and the target resource is CPU. While effective for CPU targeted attacks, Rampart is not able to detect and respond to attacks such as Billion Laughs which the target resource is memory rather than CPU. In contrast, our approach is application agnostic, i.e., is not limited to PHP applications, and is also able to detect and respond to attacks against multiple IT resources (herein, jointly CPU and memory is illustrated).

Finally, our preliminary work [130] was a supervised detection framework which focused on the theoretical performance of multiple fissioning steps. We use the insights from that work to design and implement our unsupervised service fissioning approach in this work.

5.3 Problem Formulation

In this section we present our assumptions and problem formulation. Our approach to attack response (service fissioning) naturally allows for remediation of false positives. In the following, we do not consider prior *known* attacks for purposes of detection of new ones, but such behavior can easily be included in our detection framework. Also, we assume standby pods for attack response on the (small) quarantine node (VM).

First, we assume that the users each provide a workload consisting of a stream of requests.[1] While all the requests submitted to a service are all assumed to be of the same type, but for a given set of resources, the execution times of the requests may vary depending on their "size". The size of the requests could in some cases be captured simply by the size of the input dataset to be processed, or the size of the content to be retrieved. Though we assume the workload (and hence the service) is generic, we also assume that

[1]This includes the possibility that "users" are defined as segments of IP address space (or IP + port-number to account for NAT). That is, a "user" is the superposition of all clients from its address segment.

requests are continually streamed and hence very repetitive and recurring [131]. As a result, it is reasonable to conclude that we can well characterize *nominal* individual streaming workloads in terms of the cloud resource utilizations and needs in order to meet the Service-Level Objectives (SLOs) of individual requests as a function of their size. An individual request could correspond to a different client associated with the user.

To differentiate between overloads arising from high demands and overload arising from application-layer attacks, user demand can be controlled by token bucket mechanisms in terms of request arrivals, but can only be controlled in terms of request sizes if the sizes are well characterized (so that request sizes map clearly to execution times given a set of available resources). For more complex workloads, notwithstanding such controls, unexpectedly high demand may result. However, in a security setting and in presence of an application-layer attack, malicious requests or malware may be piggybacked on requests, in particular zero days which are not detected by defenses such as firewalls. Thus, we motivate the need for run-time monitoring of *utilized* resources to detect overloaded services and individual users that are heavy hitters or attackers. Since the attacks under consideration wish to exhaust such resources, detection based on utilized resource statistics will be robust to polymorphism or metamorphism of malicious requests. Detection is based on "null" models of known (generally including both known nominal and known attack) user behavior. Null models may be reinforced using run-time statistics that have been correctly "labelled" by a security administrator.

Another basic assumption of our system is that user sessions are *not* nominally placed in *statically resource-provisioned* pods[2] within servers so as to promote efficient utilization of the servers' resources (by statistical multiplexing) and reduce overhead.

Our approach to defense is based on overload detection at the pod (containers) level. Though each service might consist of multiple pods, when a service is under attack, not all the pods are involved so it suffices to only quarantine the attacked pod rather than the entire service (i.e. all the pods supporting the service). For simplicity, in this work, we assume that a service consists of one pod and our ideas are applicable to services with multiple pods as well. For purposes of overload detection, user workload can be measured in terms of plural resources jointly utilized, e.g., CPU, memory, network IO, disk IO, and capacity for open network connections. Congesting one or more of these resources may be the aim of the DDoS attacker(s), where it's possible that just one such attacker can take down a pod.

[2]Because we will be taking the context of kubernetes, we will use the term pod instead of container.

5.4 Defense Design and Implementation

In this section, we explain our design and implementation of prototype on Kubernetes, a state-of-the-art container orchestrator and cluster manager. Our defense design consists of two parts. First the detection phase where we aim to detect an application-layer attack, and second the response phase where we aim to defend the attack either by preventing the attacker to send further requests or isolating the attacker to mitigate the influence of resource exhaustion on the legitimate users.

5.4.1 Attack Detection Design

It's well-known that many workloads that cloud-deployed applications receive over the time follow a pattern that is recurring and similar which can be inferred using historical data (i.e. what has been seen in past days, weeks, or month) [131]. We leverage this opportunity to collect resource (e.g. CPU and memory) utilizations of the deployed application at different load levels for legitimate users over time. By analyzing the historical data we calculate the average and variance (and covariance) of both CPU and Memory utilization at different load levels.

The basic idea is to monitor the resource usage of each service (pod) in last (small and configurable) T seconds and predict its usage in next T seconds. By using the knowledge from the historical data (i.e. offline data from offline profiling), we identify if the usage is abnormal (in respect to known past behavior) or not. In particular, at run-time, we use first-order auto-regressive to predict the CPU and Memory utilization of the pod at current load level. Suppose that the CPU utilization and load at time t (from last T seconds) are denoted as $C(t)$ and $L(t)$ respectively. As explained above, for different load levels L, the mean μ and variance v are known and we use the normalized CPU utilization $\hat{C}(t) = C(t)/L(t)$ at load level $L(t)$. We use a first-order auto-regressive $\xi(t)$ with a fade factor α such that $\xi(t) = \alpha\hat{C}(t) + (1-\alpha)\xi(t-1)$ to predict the CPU utilization of the container. Let F be the cumulative distribution function of 2 parameter Gamma distribution (i.e., two degrees of freedom) with mean μ and variance v that are measured and mentioned above. Note that these quantities are also get slowly updated over time in a reinforcement learning way. We check if the p-value $1 - F(\xi(t)) > \varepsilon$ for small positive threshold $0 < \varepsilon \ll 1$, then deem the pod attacked (overloaded). We use the same process for Memory utilization as well. If at least one of the resources (e.g.

CPU or Memory) is overloaded we deem the pod as attacked.[3]

5.4.2 Attack-Response Design

In this section we explain our attack-response mechanism (i.e. fissioning). Note that the goal is to identify the attacker user and isolate it so it won't have an influence on the legitimate users. Once a replica pod is detected as attacked, we deschedule the pod and launch and schedule two replica pods (i.e. fission) of the service that was running on the attacked pod on the quarantine node (see Section 5.4.4) in the cluster. We then redirect all the users of that replica pod to the new service replicas on the quarantine node and assign them evenly to the two replica pods. Note that by quarantining the attacked replica pod and its users, we completely mitigate the effect of the attack on the other replica pods that are sharing the same VM. We then repeat the same detection process (as described in section 5.4.1). If after a time period T, a quarantined pod is not deemed as attacked, we reschedule the pod and redirect its users back to the normal (not quarantined) replica pods on the normal nodes in the cluster. However, if a pod is deemed as attacked, we repeat the same process and launch one more pod (i.e. fission) and assign half of its users to the new pod and the same process goes on so that finally the attacker user is identified and all the nominal users are moved back to the normal nodes.

Once the attacker is identified, a "business" decision can be made regarding how to treat the attacker. For example, one can block that user and drop any request that the attacker sends. In order to reduce the effect of false positives, we take a more conservative approach and instead of dropping all the requests from attackers, we maintain the pods that are serving them on the quarantine node but strictly limit their resource quotas so they won't exhaust the quarantine node's resources (i.e., there is no inter-pod statistical multiplexing on the quarantine node).

Note that the two initial pods can be increased as well in order to to reduce the number of steps to identify the attacker. We perform a sensitivity analysis in our evaluation (see 5.5).

[3]Note that, generally, the CPU and memory covariance could also be measured and, to reduce false positives, a p-value based on the joint CPU-memory utilization could be computed. Reducing false positives in detection means that the attack-response phase will need to "rehabilitate" fewer quarantined legitimate users.

5.4.3 Fissioning and Quarantine

To fission the attacked service replica and its users (as described above), we reserve one node in the cluster as a *quarantine* node (VM) so that we can schedule a quarantined pod replica of the service on this node. To do so, we leverage the Kubernetes feature known as **taint and toleration** [132]. This feature allows us to taint one of the nodes in the cluster as *quarantine* node. Once a node in the cluster has a taint, the Kubernetes scheduler won't be able to schedule any pod on that node unless the pod has a *tolerance* of that taint. When we want to launch a quarantined replica of the attacked service, we add the *quarantine* tolerance to the pod so it will be able to be scheduled on the quarantine node. To prevent scheduling the quarantined pods on the normal nodes, we leverage the Kubernetes **node affinity** feature [133]. This enables us to assign the normal pods to the normal nodes in the cluster. Using these two features, we are able to to do both fissioning and quarantine.

5.4.4 Implementation

In this section, we described the detailed implementation of our prototype. Our prototype consists of modules which run as pods on Kubernetes and are implemented using GoLang as it is the Kubernetes' native language. Figure 5.1 depicts a high-level overview of our prototype and its main components. In the following, we describe each component and how it is used for attack detection and response phases.

5.4.4.1 Monitor:

The *monitor* module runs as a pod on the master node and periodically collects the system statistics and resource utilizations of all the pods. Whenever the *detection* module (discussed below) asks for the statistics of a pod, it responds by providing the mean and the variance of the utilization of each resource over the past period of (configurable) time T.

5.4.4.2 Detection:

The *detection* module stores the mean and variance of utilizations for (rough) load levels in a hash tables for each service. It then periodically polls the resource utilizations of each pod from the *monitor* module in a round-robin fashion. As described in sec 5.4.1,

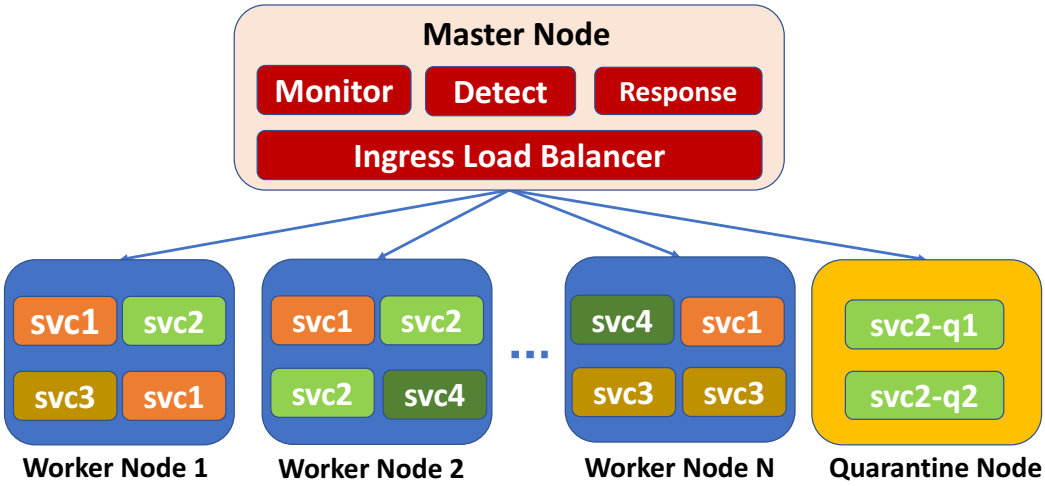

Figure 5.1. Overview of a deployed microservice on Kubernetes along with our prototype (service2 is attacked and hence quarantined)

it calculates the p-value for the service and if that pod is deemed as attacked (p-value is too small), it invokes the *response* module).

5.4.4.3 Response:

The *response* module receives defense requests from the *detection* module. The request consists of the service name and the name (or ID) of the pod replica that is running that service. Once a defense request is received, as described in Section 5.4.2, the *response* module takes the following steps to perform the defense:

- It deschedules the attacked pod.
- It launches two (or more) replicas of that service (named svc_name_q1, svc_name_q2, etc.) and schedules them on the *quarnatine* node.
- It redirects the users of the attacked pods to new quarantined pods by updating the routing tables in the *ingress* controller (discussed below) so any further request from those users will be routed to the quarantined pods.

After these initial steps are taken, the last two steps are repeated until the attacker is identified. Once the attacker is identified, the resource quota of the pod that is serving

the attacker are limited in order to not exhaust the resources on the quarantine node. As an example, once we have two quaranatined pods for a service named svc_name_q1 and svc_name_q2, and the detection module finds that the pod svc_name_q1 is not under attack, it deschedules it and returns back its users to the normal pods by updating the ingress controller routing tables. And if the pod svc_name_q2 is under attack it launch one (or more than one) pod named svc_name_q3 and updates the ingress controller routing table to evenly distribute the pod svc_name_q2 users among the both pods to further identify the attacker in the next iteration.

5.4.4.4 Ingress Controller:

In Kubernetes, the ingress controller acts as an internal load balancer to (evenly) distribute the requests among the target service replicas (i.e. pods).[4] We leverage the ingress controller to assign the users to the normal pods (reside on normal nodes) and extend it to have a routing table to route the users of the quarantined pods to the quarantine node. As mentioned above, the routing table of the ingress controller is updated each time that a quarantine pod is scheduled on the quarantine node or a pod is removed from that node and its users need to be returned back to the normal pods/nodes.

5.5 Experimental Results

In this section we demonstrate some of our preliminary results of our prototype. To gather the per load-level statistics, we run our workload with legitimate users (i.e. no attacker) at different load-levels against our deployed application on a kubernetes cluster. As explained before, the *detection* module later uses these statistics to detect whether an attack is happening or not.

5.5.1 Experimental Setup

Our cluster on AWS EC2 consists of four $m5.large$ instances each with 2 vCPUs and 8 GB of memory. One of the instances is used for the master node where our prototype module pods (i.e. *monitor*, *detection* and *response*) along with other kubernetes pods such as kubernetes scheduler are deployed. The other three instances are our worker nodes where the application pods and also the ingress controller are deployed. Note that,

[4]Generally, the ingress controller could also group into pods the requests that have similar loads (for example, same input data sizes) to facilitate statistical characterization used for attack detection.

as mentioned before, we *taint* one of the worker nodes as a quarantine node so by default no application pods are scheduled on it and this node is only used for quarantined pods (see Section 5.4.4).

Application: We use a computer vision application that performs facial landmark detection on an input image. We deploy the application as a service with 4 replicas so each of our two normal nodes will have two of the replicas scheduled on them.

Attack Scenario: The attack consists of malicious requests that attacker sends in order to exhaust the CPU or memory resources. For our application, the malicious requests can be of two types: *a*) A request with a file pretending to be an image file but contains code injected into the application to exhaust memory and/or use-up CPU cycles, *b*) A request with an standard-sized image file that contains no faces but too many face-like objects resulting in anomalously long processing times.

Baselines: We compare the performance (i.e. the total latency or execution/response time that legitimate users experience in presence of attack) of our prototype compared to a baseline where there is no detection and response mechanism in place. Because we always take one of the nodes in the cluster as a quarantine node, there are $n-1$ nodes (where n is the total number of nodes, e.g. $n = 3$ in our reported experimental results herein) that serve the legitimate nominal pods, we use two baselines. First, a baseline with $n-1$ nodes where the number of worker nodes is same as ours. Second, to be fair in terms of total cost of worker nodes we compare to a baseline with n worker nodes serving legitimate users.

5.5.2 Experiment Scenario

We deploy our application on the cluster described above. Twelve distributed users, one of which is the attacker, start sending requests to the cluster each with mean rate of 100 requests per second for a duration of 600 seconds (i.e. 10 minutes). The scheduler assigns 4 users to each replica in the cluster. After 300 seconds, the attacker starts sending malicious requests described above. The attack continues for 300 seconds. The parameters for *detection* module are as follow. We empirically find that the best fade factor for the first-order auto-regressive estimator for both CPU and memory utilization is $\alpha = 0.15$ hence we stick with that throughout the experiments. The value for T the window time in which the *detection* module polls statistics to detect the attack is chosen such that the it spans at most the serving time for two requests, therefore we set this value to $T = 3$ *seconds*. By analysing the data from our offline profiling, we set the two parameters for the Gamma distribution at the heart of the *detection* module to

$shape = 390$ and $rate = 5.2$ for CPU and $shape = 10560$ and $rate = 6.6$ for memory. Finally, the p-value to deem a pod attacked is set to 0.15.[5]

Figure 5.2 depicts the average end to end latency[6] that the 11 non-attacker users experience during the experiment. We compare the performance of our prototype against two baselines (see above). 4 seconds after the attack starts, the *detection* module detects the attack and invokes the *response* module. The *response* module then schedules two replicas of the attacked pod and quarantines them on the quarantine node and redirects its 4 users to the quarantine node and each quarantined pod serves two of the users. In next 4 seconds the *detection* module detects the attacked quarantined pod and invokes the *response* module again. The *response* module then moves back the healthy pod to the normal nodes and makes one more replica of the attacked pod and each pod will serve one of the remained two users of which one of them is the attacker. In next 3 seconds, the *detection* module detects the attacked pod and invokes the *response* module for the last time in this scenario. The *response* module then moves back the normal/not-attacking user to the normal nodes, limits the resources of the attacked pod which is serving the attacker, and keeps it scheduled on the quarantine node for the rest of the experiment. In total, it takes less than 14 seconds for the *detection* and *response* modules to identify the attacker and isolate it. As can be seen in Figure 5.2, this results in about 3× less latency for legitimate users compared to the baseline with 3 nodes, since fewer users are directly affected by the attack. This performance is reduced for the baseline with 2 nodes where less resources are available and more users are affected by the attack.

5.5.3 Scaled-Up Cluster Evaluation Results

We also ran the same experiments explained above on a scaled-up cluster with 10 nodes (still only 1 quarantine node), 60 users and 4 attackers. Same as before, two Pods are scheduled on each worker node and each Pod is serving 4 users. Figure 5.3 depicts the result for this scaled-up experiment. As it can be seen from this figure, the main difference is that since there are more attackers (and more attacked Pods) and also more legitimate users are fissioned, it takes about 10s longer to recover the nominal latency for the legitimate users. This is mainly because more legitimate users are on the quarantine node at the same time.

[5]Generally for unsupervised detection, this p-value threshold can be set to minimize false positives.
[6]The time at which user sends the request until the response is received

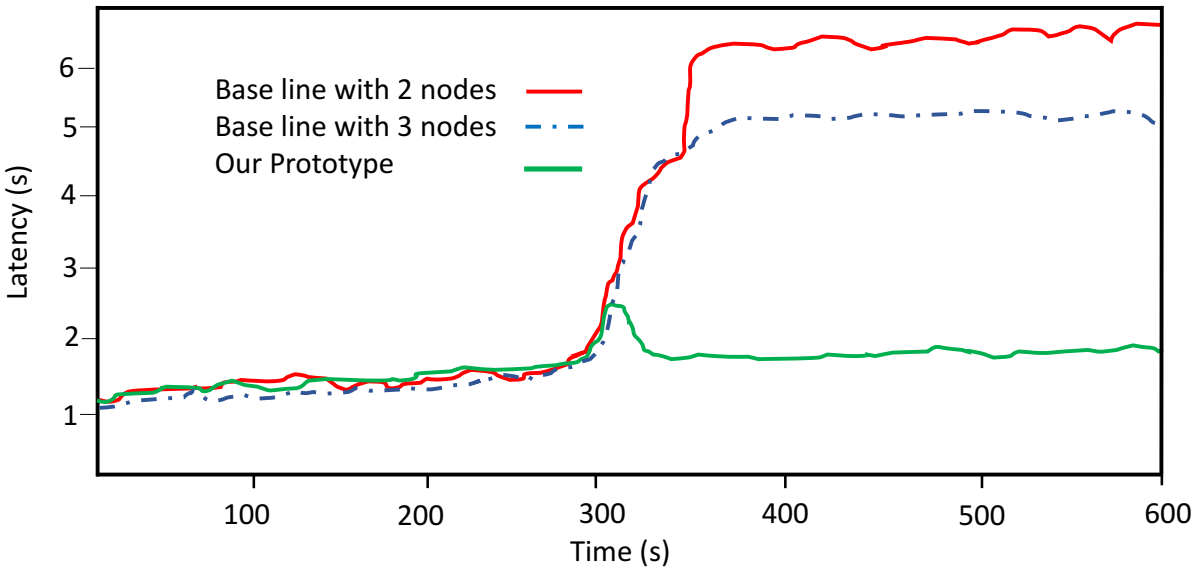

Figure 5.2. The average latency that legitimate users experience in presence of the attacker compared to in presence of our detection and response mechanism

5.5.4 Sensitivity Analysis

We also perform a sensitivity analysis for the effect of number of quarantined replica pods on the time it takes to identify the attacker. Since each time the *detection* module invokes the *response* module, it performs binary fissioning, the number of quarantine pod replicas in each step affects the time it takes to identify the attacker, because it reduces the number of steps needed to divide the users of the attacked pod to finally identify the attacker. On the other hand, the time to identify the attacker will also depend on the number of other legitimate users along with the attacker that are served by the attacked pod. Figure 5.4 depicts the effect of number of quarantine pod replicas on the time to identify the attacker for different number of users (denoted as N) assigned to a pod.

As it can be seen in this figure, as the number of users being served by an attacked pod increases, it takes longer to identify the attacker(s). Again, this is because of the performed binary fissioning which requires more steps as the number of users are increased.

As the number of quarantine pod replicas increases, the time to identify the attacker(s) decreases. This is because it basically reduces the number of steps needed for *response* module to identify the attacker. Hence one might think that it is a reasonable direction to choose a higher number of quarantine pod replicas to reduce the identification time. However, having too many quarantine pod replicas on one single quarantine node might

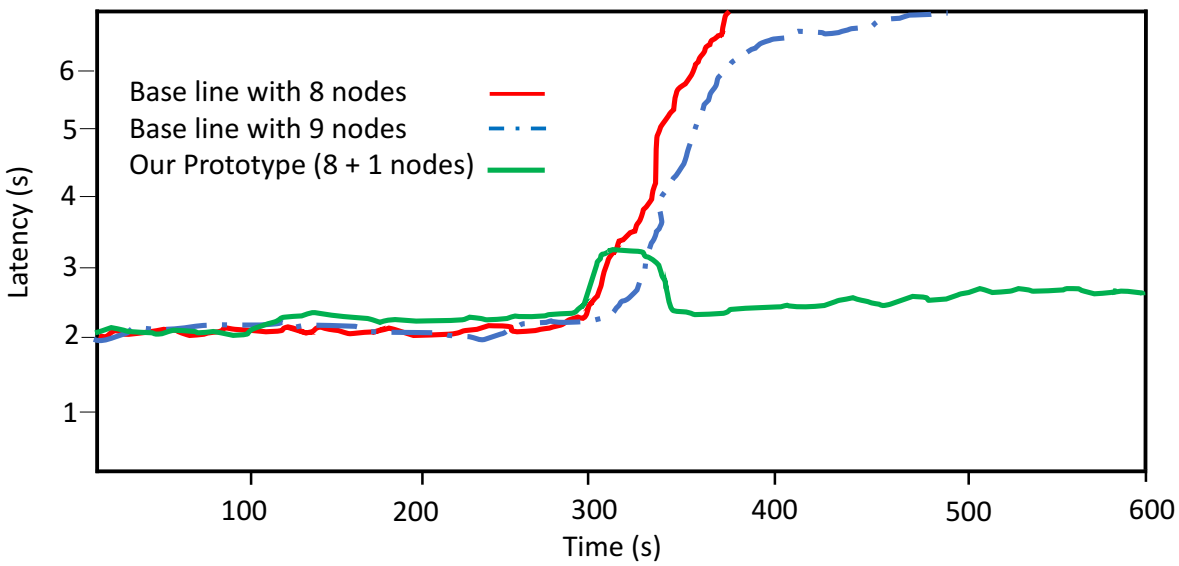

Figure 5.3. The average latency that legitimate users experience in presence of the attacker compared to in presence of our detection and response mechanism in scaled up experiment

Number of Attackers	4	8	12	16
Time to Recover Latency (seconds)	28	30	30	32

Table 5.1. Time to recover latency for legitimate users in the scaled-up cluster with 9 worker nodes and 64 users with varying number of attackers.

lead to resource contention among the pods which can affect the latency of legitimate users on the quarantine node as a result. This specifically arises when there are multiple attacks on different services, so we need to keep the number of quarantine pod replicas limited to prevent from resource contention.

Finally, in the scaled-up experiment, when we vary the number of attackers (i.e 4 attackers, then 8, 12, and 16 attackers), the results are not substantially different from what we observe from Figure 5.3. This is because the monitoring, detection, and defense modules work on all the scheduled pods on the cluster in parallel, therefore once an attacked pod with more than one attacker is quarantined, all the attackers will be fissioned and isolated in parallel. The time it takes to recover the latency for the legitimate users is presented in Table 5.1.

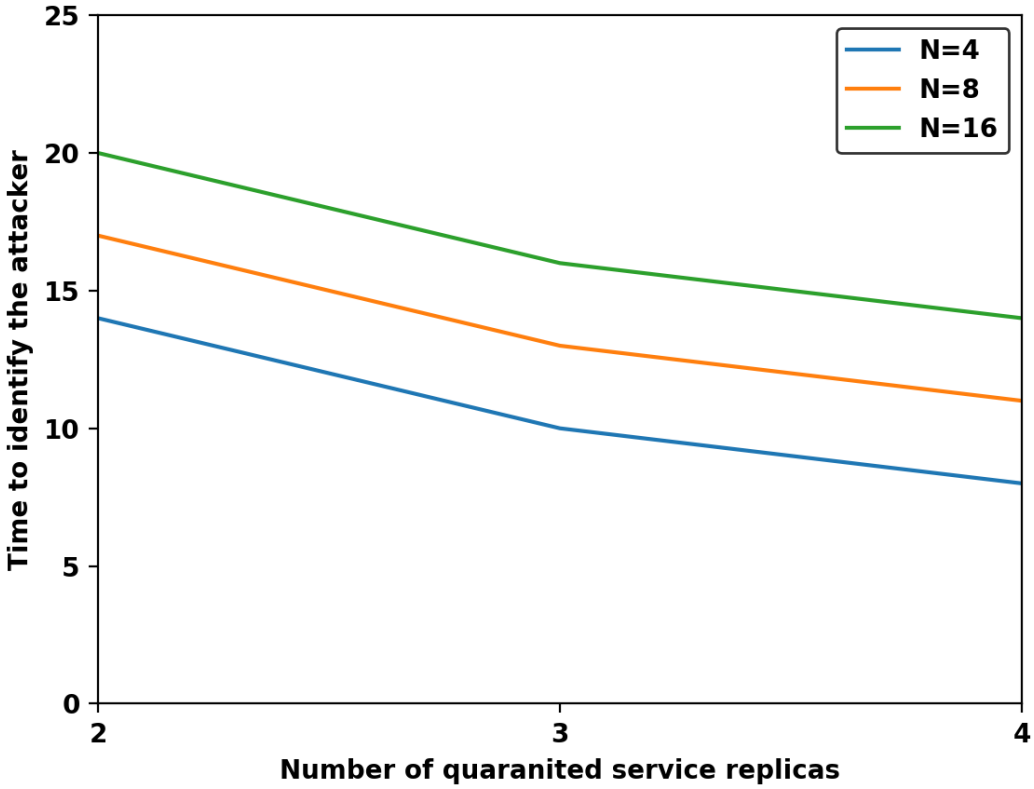

Figure 5.4. The effect of number of quarantine pod replicas on the time to identify the attacker for different number of users assigned to a pod

5.6 Discussion of Future Work

In future work, we plan to study the performance of our prototype and improve it for more complicated applications consisting of a greater number of different interdependent microservice types. In such applications, because of dependency that exists between the services, when one service is under the attack, other dependent services will be affected by the attack indirectly. We will extend our work to further explore the dependencies of the microservices in order to protect the un-attacked services from availability and performance degradation arising from attacked services that they depend on.

In this work, we evaluated our prototype with an attack similar to the billion laughs attack. We will also evaluate our prototype with more attack scenarios as discussed in Section 1.

We will also explore the use of a more sophisticated null model, e.g., multiple-

component Gaussian Mixture Models (GMMs) [134], including components corresponding to *known* attacks. Such models will *jointly* consider utilizations of different cloud resource types, i.e., their covariances, when computing p-values.

Finally, to reduce costs associated with the defense subject to latency constraints, we plan to explore the use of "warm start" cloud functions (e.g., AWS Lambda) [135] for timely and cost-effective attack response, instead of VM-based pods, for quarantine.

Chapter 6
Multi-Cloud Serverless Model

6.1 Introduction

Serverless computing is one of the fastest growing cloud paradigms. It aims to decouple infrastructure management from application development. Under the serverless paradigm, the application developer does not worry about provisioning and up-/down-scaling resources, which can be particularly tricky in the shadow of varying demand. Instead, the provider conducts automatic and scalable provisioning. This, complemented with the pay-as-you-go model and scale-down-to-zero capabilities have made serverless one of the most exciting cloud models for developers. Today, all major cloud providers have serverless offerings, mainly in the form of Function-as-a-Service (FaaS): AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions. Serverless is still growing rapidly, where new system features are frequently introduced and various aspects of pricing and quality of service (QoS) are constantly re-visited.

In this paper, we present and argue for the concept of Virtual Serverless Provider (VSP), a third-party entity that aggregates serverless offerings from multiple providers and seamlessly delivers a superior unified serverless to developers. In doing so, it prevents vendor lock-in problems and exploits competition between providers to achieve a confluence of cost, performance, and reliability goals. Specifically, this paper presents the merits of having VSPs take advantage of different pricing schemes as well as variable performance and to achieve enhanced cost, performance, and scalability.

Certain characteristics of serverless systems and ongoing technology trends make the VSP model viable. Unlike conventional cloud federations of VMs, serverless functions are much lighter and faster to deploy, eliminating the pain of slow migrations. Additionally, there are already a number of open-source provider-agnostic serverless frameworks and tools that pave the way towards building efficient and cost-effective multi-cloud serverless

aggregation. Finally, a number of recent technical advances and ongoing research areas will improve the performance predictability of today's highly variable serverless systems [136]. Increased predictability allows the VSP to better optimize choices.

Designing and implementing VSP's opens new research directions. One such avenue is developing domain-specific VSP's (e.g., for ML [137], etc.) to enhance the optimality of the multi-cloud deployment. This can potentially lead to new cloud market structures.

6.2 Merits

6.2.1 Harnessing Cost and Service Variances

Table 6.1 summarizes the pricing elements for three popular serverless providers. As seen, the final cost of a function (i.e., excluding storage or data transfer) depends on the number of invocation requests as well as execution. The execution cost is a combination of memory usage and execution time. AWS Lambda and Azure Functions offer the same limits on their free tiers: the first one million requests and 400 GB-seconds in a month are free. Beyond that, AWS Lambda charges slightly less than Azure Functions for additional requests and additional GB-s. However, the way the GB-second consumption is calculated varies between these two providers. Azure Functions comes with the overhead of charging for at least 100ms of execution, whereas Lambda is limited by static memory allocation and charging for memory even when the function is not using it. This makes running a sub-hundred-millisecond function with consistent memory usage cheaper on Lambda, and running a two-hundred-millisecond function with bursty memory usage cheaper on Azure Functions. There exist many more subtle differences in serverless pricing, especially when the entire ecosystem including storage and third-party services is considered. However, this simple example should suffice in elaborating that the cost-optimally of each serverless provider depends on the workload. The VSP can exploit this to minimize developers' overall costs while maintaining acceptable performance.

In addition to using provider differences to reduce cost, the VSP can use it for performance optimizations. An example is how different providers deploy various function keep-alive policies, considering that function containers/VMs cannot be kept alive (resident in memory) indefinitely [138, 139]. The VSP can pick the one that suits the application the best. For instance, if invocations are infrequent and too irregular and bound to get cold starts anyway, the VSP would rather map it to the FaaS provider with the cheapest offering, regardless of performance. On the other hand, an application

	Cost						
	Request Cost		Function Execution Cost				
Provider	Free	Add. Cost	Free (GB-s)	Add. Cost (per GB-s)	Min. Duration	Round-up Resol.	Mem. Metering
AWS Lambda	1 M	\$0.2/M	400,000	$\$1.67 \times 10^{-5}$	1 ms	1 ms	Static
Azure Functions	1 M	\$0.256/M	400,000	$\$2.1 \times 10^{-5}$	100 ms	1 ms	Dynamic
GCP Functions	2 M	\$0.4/M	400,000*	$\$2.5 \times 10^{-5}$*	100 ms	100 ms	Static

Table 6.1. The pricing schemes of AWS Lambda, Azure Functions and GCP Cloud Functions consumption plan as of May 28, 2021. (M: Million, GB-s: GB-seconds). *: GCP Cloud Functions charges additional $\$10^{-4}$ per GHZ-seconds with 200,000 free GHZ-seconds.

with a predictable invocation pattern can benefit from Azure Functions' adaptive Hybrid Histogram policy [138] and thus experience fewer cold starts.

Pushing this angle further, the VSP can also use the real-time performance monitoring data to adaptively distribute its function requests away from those providers experiencing a slowdown. This strategy is supported by empirical evidence that the performance variation for different providers is statistically independent. Wang et al. monitored the cold start latency of AWS, Google, and Azure for over a week and showed that their performance degradation is uncorrelated [139]. Performance variation is not limited to different providers and can exist across various regions of the same provider. Indeed in a recent study, Ginzburg et al. [136] reported significant performance variations in two AWS Lambda regions: an 11% difference in end-to-end performance as well as a 12-hour lag in daily performance degradation hour. In Section 6.5 we demonstrate the feasibility and performance benefits of such adaptive scheduling across serverless platforms.

6.2.2 Fusing the Benefits of Providers

Scalability is a central promise of serverless computing and stems from the fact that developers are not responsible for resource provisioning. It is achieved by state-of-the-art auto-scalers, relying on function images being lightweight, and usually stateless functions. Despite all of these, this scalability is not infinite. Previous characterization studies on production serverless systems have observed different scalability rates and patterns for various providers [140]. Lloyd et al. observed that going beyond fifty concurrent requests significantly increased the cold-run execution time of functions served by Azure Functions [141]. Parallel deployment on multiple serverless providers allows the VSP to increase the scalability limits, both in terms of ramp-up speed as well as sustained throughput. This parallel deployment requires low-latency load balancing at the VSP,

which is viable, as the VSP is not performing high latency tasks such as language runtime or application-specific initializations [142]. We demonstrate this later in Section 6.5.

In addition to scalability benefits, using multiple providers for each application creates a larger virtual monthly free-tier by combining free-tier limits from multiple providers. This potential should not be overlooked, as recent production serverless traces released by Azure [138] revealed that the majority of serverless applications are invoked infrequently (*"81% of the applications are invoked once per minute or less on average"* [138]). This means that while a small percentage of applications that are heavily invoked would probably see no cost-saving from free tier extension, a significant share of medium class applications would enjoy it.

6.2.3 Data-Aware Deployment

In a multi-cloud setting, it is possible that the business data and compute resources are distributed among different cloud providers due to business decisions or other technical or economic incentives. In such cases, putting computation closer to the data is the desired approach [143]. A business might not be able to migrate parts of data out of a provider or region to comply with data governance and protection laws such as GDPR. A VSP would ease deployment of such applications by scheduling and deploying functions such that 1) they are as close as possible to the data that they consume, 2) the data transfer between regions is minimized to reduce data transfer (and processing) costs, and 3) data protection laws are complied with.

6.3 Viability

We next consider the *viability* of building a VSP. To do so, we compare serverless to conventional cloud systems already federated. We also enumerate a number of new systems and technology trends that either have paved or will pave the way for VSPs.

6.3.1 Fast Dispatch

Conventional federated cloud systems are built on top of slow-booting VMs, which requires caching VM images or keeping some backup VMs alive in order to ensure smooth transitions between providers. Building a federation of serverless offerings, however, is easier to handle as the cold boot of functions is much faster than that of VMs. Initiating a medium-size VM on AWS EC2 with 2 vCPUs, 8 GB of memory, and 8 GB of SDD

storage (100/3000 IOPS) took us ~250 seconds [1]. On the other hand, initiating Node.js 8 or Python 3.6 functions on AWS Lambda took around ~1 seconds, which is 250 times faster. These VM initiation measurements were conducted without transferring any VM or container images; if those transfers were included, the gap in initiation times would have been even more dramatic due to the significantly larger size of VM images. Thus, VSP users would likely experience significantly lower delays than in traditional cloud federations, making them a viable option for application developers.

6.3.2 Provider-Agnostic APIs and Bridges

Vendor lock-in hinders the viability of VSPs: Serverless offerings are often designed to be compatible with other cloud services within the cloud provider. For example, they might have native integration with other services such as event sources, logging and metrics services, queue services, and other specific services on the cloud provider they belong to. Because of this service-level vendor lock-in problem, serverless developers cannot always benefit from services offered by other cloud providers.

Fortunately, there are several tools and frameworks which prevent API lock-in at the API-level [144]. Examples include the Serverless Framework [145], Gloo [146], PyWren [147], Fn Project [148], and Nimbella [149]. For instance, Gloo is an API gateway and controller specifically designed to support hybrid applications and clouds.

In a multi-cloud setting, it is also vital to allow functions to consume events from different event sources on multiple clouds. New systems have been introduced to support cross-cloud event bridging. Unlike AWS's EventBridge [150], TriggerMesh's EveryBridge [151] can consume event data from various sources to trigger functions on any public or private cloud. Open source and cloud-native initiatives such as Knative [152] can also be used. In particular, Knative Eventing [153] can be used to implement cloud-agnostic event sources and event consumers.

Regarding the service-level vendor lock-in problem, there remain opportunities to develop more tools and platforms to support seamless cross-cloud bridging in order to enable serverless functions on one cloud to utilize services on another cloud. Collectively, these tools and platforms circumvent the locking effect from provider-specific APIs and services and pave the way towards production class VSPs.

[1] This measurement was taken using a Ubuntu Server 18.04 LTS image in the North Virginia region.

6.3.3 Performance Predictability

Increased performance predictability of serverless systems would facilitate building VSP's. Currently, one major source of latency variation in serverless environments is cold start overhead. There have been many advances on this topic in the past few years. These include techniques that reduce cold start latency, including lightweight virtualization [154, 155], sandboxing [156], snapshotting [157, 158], ahead-of-time allocation of network interfaces [159], and language runtime optimizations [160]; and techniques that reduce the number of cold starts through adaptive lifetime management of functions [138, 161]. Given the ongoing research in this angle, we anticipate cold start effects to be mitigated to a great degree. Data caching is another practical solution for performance predictability and cost reduction [162, 163].

6.4 Proposed Architecture

Building and deploying a VSP requires careful design of an architecture that can ensure seamless transitions between providers and exploit variations in pricing and performance. Figure 6.1 shows a high-level overview of our proposed architecture. Our proposed architecture for the VSP consists of eight modules: a utility-driven scheduler, controller, event bridge, performance monitor, pre-loader, local cache, billing, and authentication. We describe the roles of each of these modules below.

Utility-driven Scheduler: This module uses performance metrics and cost information to find the right provider(s) to maximize user utility. Utility optimization policies could include cost minimization, performance maximization, or arbitrary combinations of the two objectives. The scheduler also considers hard constraints such as compliance, resource allocation limits, etc. to eliminate unacceptable providers. To account for potential variations in performance and to avoid overloading a single provider (and potentially degrading its performance), the scheduler may identify multiple providers that provide near-optimal performance. Since function performance can vary not only between providers but also between different regions of a single provider, the granularity of cloud environment options that we consider is a single region of a single provider.

Controller: The controller maps requests of each application to its set of providers according to the scheduler's utility optimization results, acting as a lightweight load balancer across the providers identified by the scheduler. For instance, if the performance at one provider suddenly degrades, the controller can quickly shift to another provider.

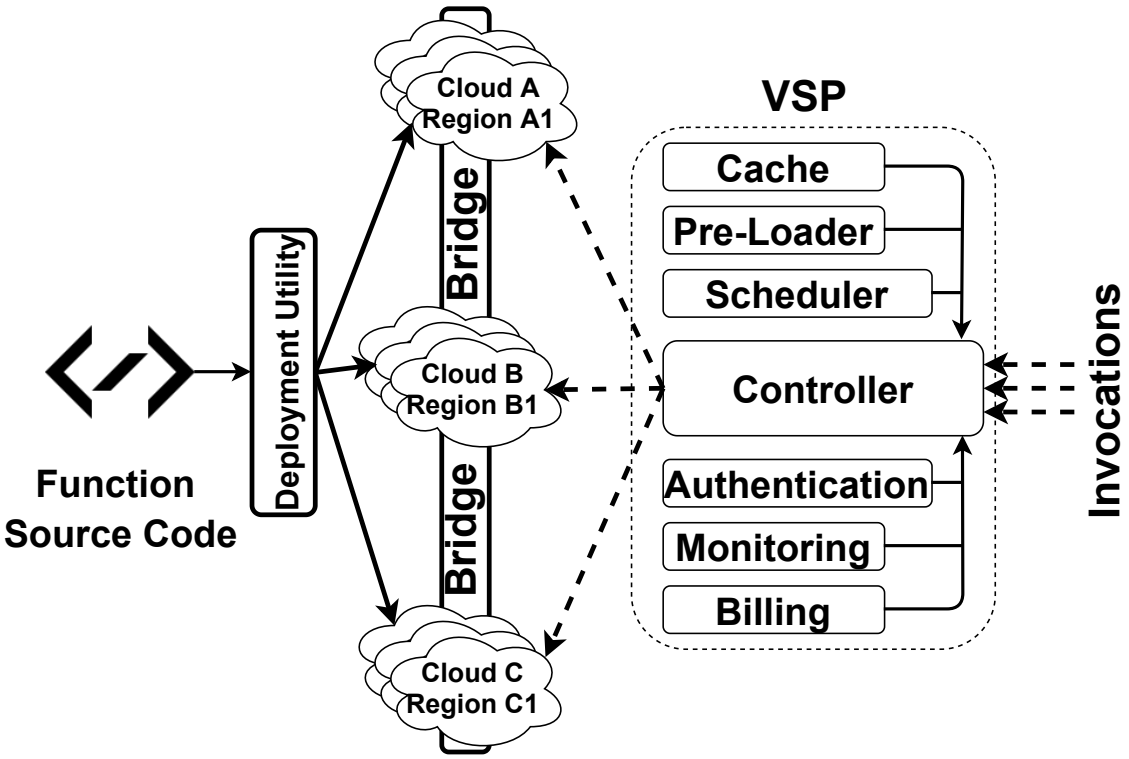

Figure 6.1. The high-level overview of the proposed VSP.

Event Bridge: Once the scheduler schedules different functions on different cloud environments, it is possible that a function on cloud A, needs to consume events from cloud B. This module as discussed before enables cross-cloud event sourcing and consumption.

Performance Monitor: Since function performance can vary over time, the set of optimal providers may vary as well. This module logs and tracks performance metrics such as latency and execution times of functions running on different providers. If a major deviation from the performance history is observed, this module can trigger the utility-driven scheduler to update the optimized mapping of functions/applications to providers.

Pre-loader: In case the scheduler identifies new providers (or new regions within the same provider) as part of the set of optimal providers, this module starts initializing and invoking the application functions in the new provider/region. It then notifies the controller module to update the set of available regions, and the controller can begin to utilize the new provider. Providing this "warm start" will reduce perceived latency from switching between providers.

Cache: Maintaining a local cache at the VSP enables a number of optimizations. For pure and memoizable functions, this eliminates sending repetitive invocations to providers. For functions with annotated data intent [164], the VSP can use the cached

data to accelerate data movement across providers to keep compute and data co-located. For generic functions without annotations or hints, recent serverless data cache designs such as OFC [162] and Faa$T [163] can be extended to operate at the VSP-level. The main challenges in designing the VSP's cache are delivering consistency guarantees and prioritizing what to cache and for what duration,

The VSP finally needs **billing** and **authentication** modules to facilitate developer payments and authentication between users and FaaS providers. The billing service keeps track of the number of requests from each user, the execution, and the corresponding providers; the VSP can then bill developers directly for both its own aggregation services and the serverless computing resources that they use at each provider. The authentication module should bridge the end-to-end authentication between users and FaaS providers.

In terms of developer experience, deploying a serverless application is similar to deploying to the cloud environment that the developer is currently using. In addition to the required deployment configuration, the developer will provide a list of cloud environments that her application can be deployed on. A deployment utility is used to deploy the application on a VSP backed by multiple cloud environment options. Note that the modules of VSP described above themselves are deployed on one (or more than one) cloud environment.

6.5 Preliminary Results

To further showcase the merits mentioned above, we implemented the prototype of a VSP. Using our preliminary results we demonstrate how the VSP allows maintaining the QoS when a provider slows down and how it reduces the costs by dynamically prioritizing cheaper providers.

6.5.1 Experimental Setup

The VSP controller is implemented using GoLang programming language. We deploy the controller on a *m4.2xlarge* AWS VM instance with 4 vCPU and 32 GB of memory. We use an open-loop load generator as our client and deploy it on a separate $m4.2xlarge$ instance. Our evaluation benchmarks cover a wide range of applications and consist of these five functions:

- *NLP*: A function that scrapes a random article from Wikipedia and builds a bag-of-words model from it.

- *Image Resizing*: A function that gets an image from an S3 bucket and resizes it to 3 target sizes and a thumbnail.
- *Video Encoding*: A function that gets a video file from an S3 bucket and converts it to GIF format using the *ffmpeg* utility.
- *ALU*: A CPU-intensive ALU function from the ServerlessBench [165] that launches a random number of threads to perform arithmetic calculations in a loop with a random number of iterations.
- *Sleepy*: A function that sleeps for one second, then sends a request to a back-end web service and it returns once the response is received. The web service receives a random number from the functions and performs an addition to it and returns.

Unless otherwise mentioned, all the functions are configured to have 512 MB of memory. We use AWS Lambda and Google Cloud Functions as the underlying providers of the VSP.

6.5.2 Results

Increasing the concurrency limit. In order to provide highly available managed services, cloud providers set limits on individual tenant's usage. In particular, for FaaS offerings, they set limits on the number of concurrent instances of the function serving invocations. Additional invocations beyond the concurrency limit will fail with throttling errors. A VSP can prevent such failures and thus improve the quality of service by avoiding scheduling further invocations to the cloud provider that has reached its concurrency limit.

To demonstrate this, we deployed our benchmarks on both providers to find the maximum throughput at which we achieve 100% success. Lambda has a default concurrency limit of 1,000. The concurrency of Google Cloud Functions is configurable and we set it to 1,000 for a fair comparison. Figure 6.2 depicts the maximum sustained throughput for each benchmark using AWS Lambda, GCP Functions, and the multi-cloud VSP setting. We did not include the Sleepy function as it is not a representative workload for throughput. For *Image Resizing* and *Video Encoding*, the GCP-only setting has significantly lower rate as the data resides on an AWS S3 bucket. We included this case intentionally as this might happen for a VSP too. The VSP uses *Least Outstanding Invocations* (*LOI*) as its load balancing policy to distribute invocations among the providers. As seen, the

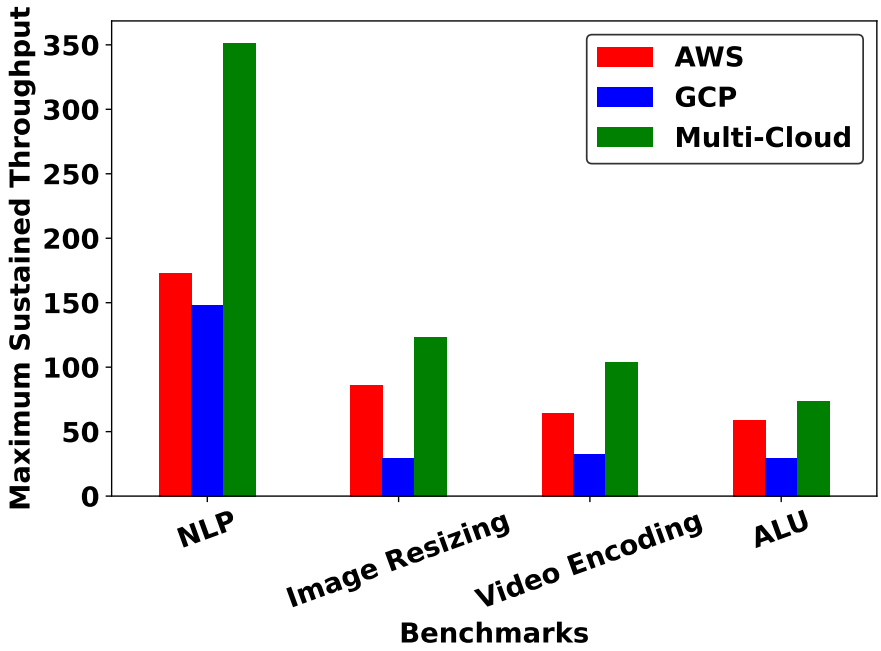

Figure 6.2. Given the same concurrency limit (1,000), the maximum sustained throughput is different for AWS and GCP. The multi-cloud VSP seamlessly increases this throughput.

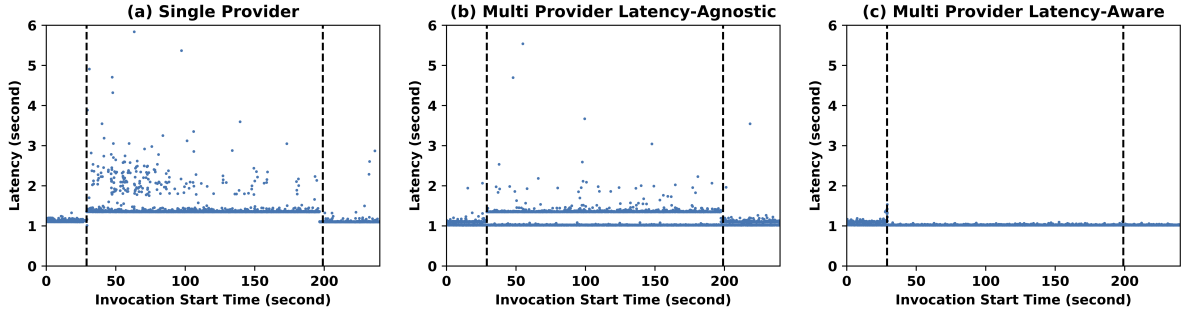

Figure 6.3. Performance of various VSP scheduling strategies under injected latency anomaly in the 30s-200s window. A latency-aware approach allows diverting invocations from slow provider and guarantees meeting the SLO.

VSP achieves throughputs close to the sum of the two cloud providers'. This is due to its low added latency – an average of 150μs in our measurements. In particular, compared to the AWS-only setting, the VSP improves the quality of service by 1.2x (for *ALU*) to 2x (for *NLP*). Compared to GCP only setting, the VSP improves the quality of service by 2.5x (for *ALU*) to 4.2x (for *Image Resizing*).

Latency-aware deployment. As discussed in Section 6.2.1, VSP's can exploit the variable performance of different cloud providers to reduce latency and cost (through execution time reduction). In particular, by monitoring the performance of each provider, a VSP can identify latency SLO violations and avoid scheduling the invocations to the problematic provider to mitigate SLO violations. To demonstrate this, we deploy the

Sleepy function and the back-end web services on both AWS and GCP cloud and set the latency SLO to 1.2 seconds. We run a fixed load of 30 requests per second for 4 minutes. To identify the SLO violation, the VSP maintains the Exponentially Weighted Moving Average (EWMA) of latency values for each cloud provider. The VSP de-prioritizes scheduling to the cloud provider(s) with EWMA latency violating the SLO. To emulate performance degradation at a provider, we inject a synthetic anomaly [166] at time $t = 30s$ to add an additional *250ms* latency to the GCP function invocations. The anomaly lasts for 170 seconds, and after that, the web service continues to perform normally.

Figure 6.3 shows the end-to-end latency of invocations. We compare the performance of three settings for the VSP: single provider (GCP), two-provider latency-agnostic VSP with round-robin load balancing, and two-provider setup with latency-aware scheduling. As seen, once the anomaly starts, the invocation latency for the single provider and the latency-agnostic settings increases and lasts during the entire 170-second window. Note that as the back-end web service gets overloaded by the constant invocation rate, latency variance also increases. The latency-aware VSP adapts to the situation in roughly two seconds (given the EWMA averaging, it is not instantaneous) and diverts invocations to the healthy provider (AWS Lambda here).

Cost improvements. In the pay-as-you-go pricing model of serverless, applications are charged for their true execution. Thus, a higher execution time for non-application-related reasons such as high context switches in overloaded servers [167] would entail a higher cost for developers [168, 169].

In the last experiment, we added the *250ms* extra latency by increasing execution times. In reality, latency variances can stem from non-execution-related sources, such as a network slow-down or an overloaded API gateway. However, for brevity, we use the same simple experiment to show the impact of cost-aware scheduling when execution times are monitored by the VSP. Figure 6.4 shows the distribution of invocation cost and latency values for the experiments shown above in Figure 6.3. As seen, using a multi-cloud cost-/latency-aware VSP reduces the cost as well as the latency by diverting invocations from the slow provider during the slow-down window. In particular, under the cost-aware VSP, 99% of the requests cost less than $\$1.02 * 10^{-5}$ compared to $\$1.4 * 10^{-5}$ and $\$2.25 * 10^{-5}$under the cost-agnostic VSP and the single provider deployment, respectively. This translates to 27% and 54% cost improvements compared to the cost-agnostic VSP and the single provider deployment, respectively.

Note that the cost numbers in this experiment only reflect the execution cost that is

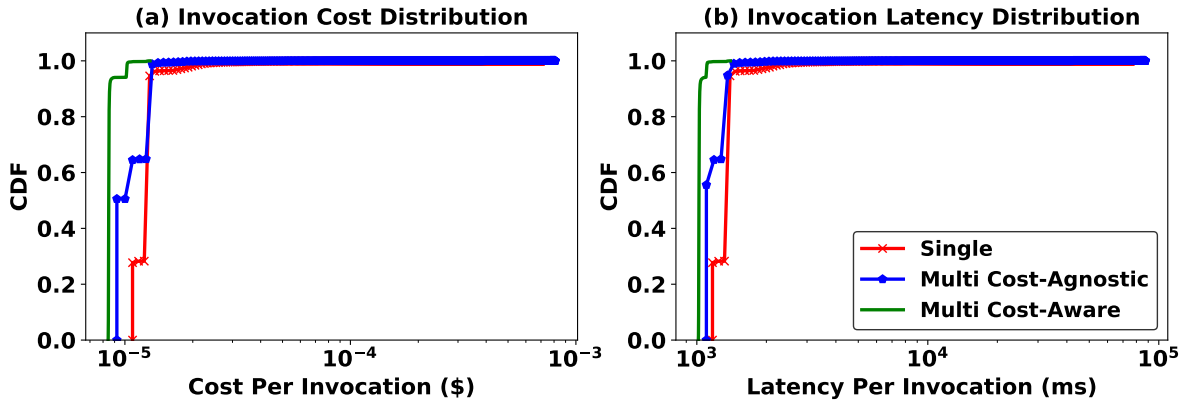

Figure 6.4. Cost and latency of invocations distribution.

the dominating cost factor for serverless systems. However, depending on the application, the total service cost can include other components such as storage cost. Specifically, in situations where data needs to be replicated across cloud platforms storage costs can become more significant. The cost calculations and comparison in such scenarios are left as future work.

6.6 Discussion

Controversial points: As we discuss in Section 6.7, federated architectures and algorithms have been proposed for cloud services for many years. However, these architectures are not widely deployed today, in part due to vendor lock-in effects. Serverless computing is less susceptible to many obstacles to realizing federated clouds (see Section 6.3), but it remains to be seen whether others would prove fatal to building VSPs. The economic viability of such virtual providers is also a concern; the VSP's scheduler would need to be reconfigured whenever an individual cloud provider significantly changed its serverless pricing or execution logic, which could prove infeasible in practice. Additionally, one may argue that due to economic incentives, cloud providers may try to sabotage the VSPs. On the contrary, we believe that the VSP can improve the adoption of serverless computing and hence expanding the market for cloud providers. Note that, in our proposed model, the VSP runs third-party code within its own purchased resources and is thus the sole customer dealing with each provider.

Open challenges: Several research challenges remain before production-scale VSPs can be realized. Since a (if not the) major use case of cloud computing is data analytics, many functions invoked by serverless applications might run on data stored at the FaaS provider. Thus, a viable VSP will require effective mechanisms for maintaining

consistent data storage across multiple providers. Developing algorithms to optimally exploit temporal cost and performance variations at the different providers is another open area of research; it is not clear how the scheduler should determine the optimal set of FaaS providers, or how the presence of a VSP would affect FaaS providers' pricing and performance policies.

Points of failure: Maintaining consistent data storage across multiple providers will likely incur additional costs, which may cancel out any savings from utilizing multiple providers. Individual cloud providers might also attempt to block VSP's from accessing serverless functions in order to preserve the competitive advantages of vendor lock-in.

VSP operation costs: The reported cost numbers in this paper do not include the operation of the VSP itself. We demonstrate the total savings from aggregating serverless providers. Based on those savings, the VSP, which is a separate entity from the providers, developers, and end-users, decides the profitability of the adoption.

6.7 Related Work

Researchers have studied various aspects of serverless computing systems in the past few years. Those span over scheduling and resource management, isolation and virtualiztion, novel applications and services, performance analysis of public offerings or open-source serverless platforms, and economics of serverless architecture within a single provider. However, to the best of our knowledge, the research community has not explored the multi-cloud serverless angle extensively yet.

Lithops [170] uses Python's multi-processing library to enable running Python programs on multiple serverless providers. It enables the developers to write programs at once and leverage the scalability of multi-cloud deployment. Lowgo [171] is a tracing system for serverless applications deployed on multiple cloud environments and can be used to record the events and dependencies between to ease the correctness and performance debugging. Aske et al. [172] proposed deploying serverless applications on the resource-constrained edge platforms along with the cloud providers. It is limited to scheduling the application on one environment at a time based on latency SLO. In contrast, we propose simultaneous utilization of multiple providers adaptively based on a variety of factors including workload patterns, variable performance, concurrency limits, costs, latency SLOs, data locality, and security. Furthermore, our design aims to improves the total costs by exploiting different providers and multiplexing them. Using simulations, other works [173, 174] have investigated the cost-performance trade-off for executing

the serverless functions in environments with computation resource heterogeneity and disparity. They introduce new optimization objectives the results of which can be used by the VSP's utility scheduler for better function placement in a multi-cloud setting.

Cloud computing users have faced challenges like vendor lock-in and differences in pricing even before the advent of serverless computing. To overcome these challenges, several works have proposed *federating* or interconnecting cloud services [175]. Such federations allow cloud providers to pool their resources so that users can build and access services that operate across multiple providers [176]. Since users can freely move between providers, the providers are forced to compete to offer better, less expensive services, thus benefiting users. Indeed, prior work has examined cloud providers' incentives for joining such a federation [177]. A concurrent work recently described the merits and viability of aggregating cloud computing platforms [178]. In this paper, we explore aggregating serverless offerings, its technical challenges, and conduct preliminary experiments with a prototype. Other work has explored building systems for integrating cloud storage services [179, 180], which, much like our proposed VSP, allows users to take advantage of differentiated pricing at different storage services. Similarly, other work has studied the aggregation of content delivery network (CDN) providers [181–184]. In doing so, distributing an object across multiple CDN providers is somewhat similar to our setup. Building such a virtual provider for serverless computing, however, raises new challenges: in particular, the decision of which provider to use at which time must account for not only storage costs but also execution costs and temporal variations in the performance of the serverless functions. Recently, [185] showed the feasibility of the serverless aggregation idea using a small local cluster. However, we are advocating for large-scale public cloud federation.

Another line of research aims to build algorithms to dynamically utilize multiple cloud services or providers in order to minimize cost. Much of this work has focused on utilizing Amazon EC2's spot and burstable offerings, which offer discounted services at lower availability [4, 186, 187]. These works generally attempt to minimize user costs while participating in multiple markets, with the hope that at least one market will have available resources that the user can utilize at any given time. Thus, their focus is on managing availability by exploiting the dynamics of the spot market. Other work [188] has examined the viability of a virtual provider that aggregates user jobs at multiple cloud providers in order to save cost. However, the temporal variation in serverless computing performance, as well as the relative complexity of its pricing policies, likely require the VSP to develop new scheduling algorithms for mapping functions to providers.

Chapter 7
Concluding Remarks and Future Work

7.1 Conclusion

As the internet services are becoming more popular, the business owners need more IT resources in order to supply the demand from their users. However, the cost of maintaining and operating the IT resources is also increasing. In order to focus on the business, rather than managing the private IT resources, businesses (aka tenants) have started to move to the public cloud [1]. The public cloud providers provide virtually unlimited amount of resource in variety of types including Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS) (i.e. serverless computing), and Function as a Service (FaaS). Tenants might choose one or more than one type of resource offerings to operate their services on the public cloud. The success of the tenants business is profoundly impacted by how efficient their services are. We argue that the efficiency itself can be decoupled into three main aspects: cost, performance, and security. In this thesis, we proposed, developed, and implemented mechanisms and systems to achieve cost and performance efficiency while operating on the public cloud and mechanisms to develop attack-resilient cloud-deployed services which are subject to application-layer DDoS attacks.

In the first part (chapter 3) of this thesis we described our system *BurScale*. *BurScale* is an autoscale system that exploits the cheap price of burstable instances in order to minimize the cost of resource provisioning in the public cloud. BurScale uses the results from queueing theory known as “square root staffing rule" to decouple the cluster of virtual machines into two parts: regular VM instances and burstable VM instances. By combining burstable and regular instances, BurScale is able to save in cost by 50% for

both stateless and stateful applications.

In the second part of this thesis (chapter 4) we presented *SHOWAR* to improve the efficiency of deploying microservices on public cloud through right-sizing and efficient scheduling of the containers. *SHOWAR* consists of three major components: a vertical autoscaler, a horizontal autoscaler, and a scheduling affinity (and anti-affinity) rule generator. For vertical autoscaling, *SHOWAR* utilizes the empirical variance in the resource usage of containers to determine the size (e.g. number of CPUs and Memory size) of each container. *SHOWAR* uses results from control theory for horizontal autoscaling of microservices. Finally, using the resource usage correlation between different microservices, it generates affinity (and anti-affinity) rules for the scheduler to better schedule the microservices. *SHOWAR* is able to save in cost by 22% compared to the state of the art autoscalers for microservices.

In the third part of this thesis (chapter 5), we presented our mechanisms and methods for detecting and defending application-layer DDoS attacks on microservices. We leverage the capabilities of Kubernetes, the state of the art container orchestrator tool, to detect and defend the attacks against cloud-deployed microservices. Our experimental evaluations show that our mechanisms can efficiently detect and isolate the attacks and reduce the impact of attack on the legitimate users by 3×.

Finally, in the last part (chapter 6), we advocate for a multi-cloud serverless model where this model aggregates multiple cloud providers services to achieve the best cost and performance for the FaaS workloads using a virtual serverless provider (VSP). We first discuss the merits of such a model and then present the viability of a multi-cloud serverless platform that seeks for cost and performance efficiency. Our results from evaluating an initial prototype of a VSP show that VSPs can potentially save more than 50% in deploying FaaS workloads.

7.2 Future Work

Here, we highlight a few challenging future research directions.

7.2.1 More Cost-Effective Resource Management

In chapter 3 we presented *BurScale* as a cost-effective cluster autoscaler. In addition, in chapter 4 we presented *SHOWAR* as cost-effective and performant application autoscalers for the microservices deployed on a cluster virtual machines on public cloud. However,

one of the major limitations in evaluating *SHOWAR* was that we assumed a fixed-size cluster. As such, an interesting future work is making the application autoscaler and the cluster autoscaler work together to benefit from both autoscalers and save further in costs.

Another line of future work for improving the cost-efficacy of autoscaling is using machine learning (ML) techniques to accurately predict the incoming workload. An accurate prediction of the workload can help both the cluster and the application autoscaler to proactively provision resources with out performance penalties.

7.2.2 Multi-Cloud Serverless and Sky Computing

In chapter 6 we presented our vision for multi-cloud serverless computing through an initial prototype of virtual serverless provider (VSP). Exploring many aspects of a VSP is left as future work. In particular, implementing a full-fledged VSP with multi-provider and multi-tenant support is a major future work that needs to be done. In addition to implementations, other aspects of the VSP can be studied in the future: How a VSP can ensure security of tenant's workload and data? How a VSP can comply with data protection laws such as GDPR? Finally, studying new techniques and optimizations for provider selection is an interesting future work.

A new trend in multi-cloud computing has emerged in the form of Sky computing (i.e. beyond cloud computing, see [178]) that envisions to unify the public cloud offerings. In doing so, new interfaces and abstractions will be developed to ease service deployment on multiple cloud providers. A direction for future work is exploring how a VSP can fit in the bigger picture of Sky computing.

Bibliography

[1] "Spotify Moves to GCP," https://cloud.google.com/blog/products/gcp/spotify-chooses-google-cloud-platform-to-power-data-infrastructure, accessed: 2020-04-02.

[2] Atikoglu, B., Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny (2012) "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, ACM, pp. 53–64.

[3] "Spotify GCP Costs," https://davidmytton.blog/how-much-is-spotify-paying-google-cloud/, accessed: 2020-04-02.

[4] Baarzi, A. F., T. Zhu, and B. Urgaonkar (2019) "BurScale: Using burstable instances for cost-effective autoscaling in the public cloud," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 126–138.

[5] López, P. G., M. Sánchez-Artigas, G. París, D. B. Pons, Á. R. Ollobarren, and D. A. Pinto (2018) "Comparison of FaaS orchestration systems," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, pp. 148–153.

[6] "Kubernetes," https://kubernetes.io/, accessed: 2020-04-02.

[7] "Google Cloud Engine Micro Instances," https://cloud.google.com/compute/docs/machine-types#sharedcore, accessed: 2018-01-13.

[8] "Amazon AWS T2 Series," https://aws.amazon.com/ec2/instance-types/t2/, accessed: 2017-12-30.

[9] "Amazon AWS T3 Series," https://aws.amazon.com/ec2/instance-types/t3/, accessed: 2018-08-24.

[10] "Microsoft Azure B Series," https://docs.microsoft.com/en-us/azure/virtual-machines/windows/b-series-burstable, accessed: 2018-01-13.

[11] "Memcached," https://memcached.org/, accessed: 2018-06-03.

[12] Berger, D. S., B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter (2018) "RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, USENIX Association, Carlsbad, CA, pp. 195–212. URL `https://www.usenix.org/conference/osdi18/presentation/berger`

[13] Urdaneta, G., G. Pierre, and M. van Steen (2009) "Wikipedia Workload Analysis for Decentralized Hosting," *Elsevier Computer Networks*, **53**(11), pp. 1830–1845, `http://www.globule.org/publi/WWADH_comnet2009.html`.

[14] Gong, Z., X. Gu, and J. Wilkes (2010) "PRESS: PRedictive Elastic ReSource Scaling for cloud systems." *CNSM*, **10**, pp. 9–16.

[15] Jiang, J., J. Lu, G. Zhang, and G. Long (2013) "Optimal cloud resource auto-scaling for web applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, IEEE, pp. 58–65.

[16] Gandhi, A., M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch (2012) "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Transactions on Computer Systems (TOCS)*, **30**(4), p. 14.

[17] Cruz, F., F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça (2013) "Met: workload aware elasticity for nosql," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ACM, pp. 183–196.

[18] Padala, P., K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant (2009) "Automated control of multiple virtualized resources," in *Proceedings of the 4th ACM European conference on Computer systems*, ACM, pp. 13–26.

[19] Urgaonkar, B., G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi (2005) "An analytical model for multi-tier internet services and its applications," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, ACM, pp. 291–302.

[20] Nelson, R. D. and T. K. Philips (1989) *An approximation to the response time for shortest queue routing*, vol. 17, ACM.

[21] Lin, H.-C. and C. S. Raghavendra (1992) "An analysis of the join the shortest queue (JSQ) policy," in *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, IEEE, pp. 362–366.

[22] Harchol-Balter, M. (2013) *Performance modeling and design of computer systems: queueing theory in action*, Cambridge University Press.

[23] Hafeez, U. U., M. Wajahat, and A. Gandhi (2018) "ElMem: Towards an Elastic Memcached System," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, pp. 278–289.

[24] Zhu, T., A. Gandhi, M. Harchol-Balter, and M. A. Kozuch (2012) "Saving Cash by Using Less Cache," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, USENIX Association, Berkeley, CA, USA, pp. 3–3.
URL http://dl.acm.org/citation.cfm?id=2342763.2342766

[25] "mcrouter prefix routing," https://github.com/facebook/mcrouter/wiki/Routing-Prefix, accessed: 2019-01-03.

[26] Gandhi, A., P. Dube, A. Karve, A. Kochut, and L. Zhang (2018) "Model-driven optimal resource scaling in cloud," *Software & Systems Modeling*, **17**(2), pp. 509–526.

[27] Gandhi, A., T. Zhu, M. Harchol-Balter, and M. A. Kozuch (2012) "SOFTScale: stealing opportunistically for transient scaling," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, Springer, pp. 142–163.

[28] "Netflix Project Nimble," https://bit.ly/3114KoF, accessed: 2019-04-02.

[29] "AWS CloudWatch," https://aws.amazon.com/cloudwatch/, accessed: 2019-01-02.

[30] "Wikimedia Software," https://www.mediawiki.org/wiki/MediaWiki, accessed: 2018-05-17.

[31] "Wikimedia Database Dump," https://archive.org/details/enwiki-20180320, accessed: 2018-05-17.

[32] van Baaren, E.-J. (2009) "Wikibench: A distributed, wikipedia based web application benchmark," *Master's thesis, VU University Amsterdam*.

[33] Roy, N., A. Dubey, and A. Gokhale (2011) "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, IEEE, pp. 500–507.

[34] "Amazon AWS AutoScaling Service," https://aws.amazon.com/autoscaling/, accessed: 2017-10-02.

[35] "Auto Scaling on the Google Cloud Platform," https://cloud.google.com/compute/docs/autoscaler/, accessed: 2018-01-16.

[36] "Auto Scaling on the Microsoft Azure," https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-how-to-scale-portal, accessed: 2018-01-22.

[37] "Spotify Big Table Autoscaler," https://github.com/spotify/bigtable-autoscaler, accessed: 2019-01-02.

[38] Wang, C., B. Urgaonkar, N. Nasiriani, and G. Kesidis (2017) "Using Burstable Instances in the Public Cloud: Why, When and How?" *Proc. ACM Meas. Anal. Comput. Syst.*, **1**(1), pp. 11:1–11:28.
URL http://doi.acm.org/10.1145/3084448

[39] Jiang, Y., M. Shahrad, D. Wentzlaff, D. H. Tsang, and C. Joe-Wong (2019) "Burstable Instances for Clouds: Performance Modeling, Equilibrium Analysis, and Revenue Maximization," in *Proc. IEEE INFOCOM*.

[40] Wang, C., B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang (2017) "Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud," in *Proceedings of the Twelfth European Conference on Computer Systems*, ACM, pp. 620–634.

[41] Harlap, A., A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons (2018) "Tributary: spot-dancing for elastic services with latency SLOs," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, USENIX Association.

[42] Ali-Eldin, A., J. Westin, B. Wang, P. Sharma, and P. Shenoy (2019) "SpotWeb: Running Latency-sensitive Distributed Web Services on Transient Cloud Servers," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, ACM.

[43] Wang, W., B. Li, and B. Liang (2013) "To Reserve or Not to Reserve: Optimal Online Multi-Instance Acquisition in IaaS Clouds." in *ICAC*, pp. 13–22.

[44] Sharma, U., P. Shenoy, S. Sahu, and A. Shaikh (2011) "A cost-aware elasticity provisioning system for the cloud," in *2011 31st International Conference on Distributed Computing Systems*, IEEE, pp. 559–570.

[45] Novak, J. H., S. K. Kasera, and R. Stutsman (2019) "Cloud functions for fast and robust resource auto-scaling," in *2019 11th International Conference on Communication Systems & Networks (COMSNETS)*, IEEE, pp. 133–140.

[46] Wang, L., M. Li, Y. Zhang, T. Ristenpart, and M. Swift (2018) "Peeking Behind the Curtains of Serverless Platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, USENIX Association, Boston, MA, pp. 133–146.
URL https://www.usenix.org/conference/atc18/presentation/wang-liang

[47] Jonas, E., J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. (2019) "Cloud Programming Simplified: A Berkeley View on Serverless Computing," *arXiv preprint arXiv:1902.03383*.

[48] Hellerstein, J. M., J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu (2018) "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*.

[49] Burns, B. (2018) *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*, " O'Reilly Media, Inc.".

[50] (April, 05, 2021), "Microsoft Microservices Architecture Guide," https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.

[51] (April, 21, 2021), "Microservices Architecture on Google App Engine," https://cloud.google.com/appengine/docs/standard/python/microservices-on-app-engine.

[52] (April, 01, 2021), "Adopting microservices at Netflix," https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[53] (April, 10, 2021), "Adapt or Die: A microservices story at Google, December," https://status.cloud.google.com/incident/zall/20013.

[54] Wolff, E. (2016) *Microservices: flexible software architecture*, Addison-Wesley Professional.

[55] Zhou, H., M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang (2018) "Overload control for scaling wechat microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 149–161.

[56] Parker, A., D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs (2020) *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*, O'Reilly Media.

[57] Reiss, C., A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch (2012) "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the third ACM symposium on cloud computing*, pp. 1–13.

[58] Lu, C., K. Ye, G. Xu, C.-Z. Xu, and T. Bai (2017) "Imbalance in the cloud: An analysis on alibaba cluster trace," in *2017 IEEE International Conference on Big Data (Big Data)*, IEEE, pp. 2884–2892.

[59] (April, 12, 2021), "Google Cloud Services Outage Due to Insufficient Resource Quotas," https://status.cloud.google.com/incident/zall/20013.

[60] (April, 02, 2021), "Kubernetes," https://kubernetes.io/.

[61] Hellerstein, J. et al. (2004) *Feedback Control for Computing Systems*, IEEE Press / Wiley.

[62] Janert, P. K. (2013) *Feedback control for computer systems: introducing control theory to enterprise programmers*, " O'Reilly Media, Inc.".

[63] Rzadca, K., P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes (April 2020) "Autopilot: Workload autoscaling at Google," in *Proc. ACM EuroSys.*

[64] (March, 22, 2021), "Kubernetes Autoscalers," https://github.com/kubernetes/autoscaler.

[65] Gan, Y., Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. (2019) "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–18.

[66] (April, 02, 2021), "Docker Swarm," https://docs.docker.com/engine/swarm/.

[67] Verma, A., L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes (2015) "Large-scale cluster management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, pp. 1–17.

[68] (April, 07, 2021), "eBPF," https://ebpf.io/.

[69] (April, 07, 2021), "bpftrace," https://bpftrace.org/.

[70] (April, 07, 2021), "Facebook Katran," https://github.com/facebookincubator/katran.

[71] (April, 07, 2021), "The Cilium Project," https://cilium.io/.

[72] (April, 07, 2021), "The Falco Project," https://falco.org/.

[73] (April, 14, 2021), "Autoscaling in Google Cloud Platform," https://cloud.google.com/compute/docs/load-balancing-and-autoscaling.

[74] (April, 14, 2021), "Autoscaling in Amazon Web Sevices Cloud," https://aws.amazon.com/autoscaling/.

[75] (April, 14, 2021), "Autoscaling in Microsoft Azure Cloud," https://azure.microsoft.com/en-us/features/autoscale/.

[76] Lorido-Botran, T., J. Miguel-Alonso, and J. A. Lozano (2014) "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, **12**(4), pp. 559–592.

[77] Kalavri, V., J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe (2018) "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 783–798.

[78] Gias, A. U., G. Casale, and M. Woodside (2019) "ATOM: Model-driven autoscaling for microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, pp. 1994–2004.

[79] Qiu, H., S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer (2020) "FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association.

[80] Gan, Y., M. Liang, S. Dev, D. Lo, and C. Delimitrou (2021) "Sage: Practical & Scalable ML-Driven Performance Debugging in Microservices," .

[81] Kannan, R. S., L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang (2019) "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–16.

[82] Sriraman, A., A. Dhanotia, and T. F. Wenisch (2019) "Softsku: Optimizing server architectures for microservice diversity at scale," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 513–526.

[83] Zhang, Y., W. Hua, Z. Zhou, E. Suh, and C. Delimitrou (2021) "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices," .

[84] MacCarthaig, C. (Sept. 1, 2019), "PID Loops and the Art of Keeping Systems Stable," https://www.youtube.com/watch?v=3AxSwCC7I4s.

[85] Park, S.-M. and M. Humphrey (2009) "Self-tuning virtual machines for predictable escience," in *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, IEEE, pp. 356–363.

[86] Lim, H., S. Babu, and J. Chase (2010) "Automated control for elastic storage. In: Proceeding of the 7th international conference on Autonomic computing-ICAC'10," .

[87] Gandhi, A., Y. Chen, D. Gmach, M. Arlitt, and M. Marwah (2011) "Minimizing data center SLA violations and power consumption via hybrid resource provisioning," in *2011 International Green Computing Conference and Workshops*, IEEE, pp. 1–8.

[88] Krioukov, A., P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. H. Katz (2010) "Napsac: Design and implementation of a power-proportional web cluster," in *Proceedings of the first ACM SIGCOMM workshop on Green networking*, pp. 15–22.

[89] (Feb. 16, 2021), "Kubernetes vertical pod autoscaler," https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler.

[90] (Feb. 16, 2021), "Kubernetes horizontal pod autoscaler," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[91] Gan, Y., Y. Zhang, K. Hu, Y. He, M. Pancholi, D. Cheng, and C. Delimitrou (2019) "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices," in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[92] Levin, J. and T. A. Benson "ViperProbe: Rethinking Microservice Observability with eBPF," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*.

[93] (Feb. 18, 2021), "CPU Utilization is Wrong," http://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html.

[94] Gotin, M., F. Lösch, R. Heinrich, and R. Reussner (2018) "Investigating performance metrics for scaling microservices in cloudiot-environments," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 157–167.

[95] Dean, J. and L. A. Barroso (2013) "The tail at scale," *Communications of the ACM*.

[96] Ousterhout, K., P. Wendell, M. Zaharia, and I. Stoica (2013) "Sparrow: distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*.

[97] Lo, D., L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis (2015) "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*.

[98] Barroso, L. A. and U. Hölzle (2009) "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, **4**(1), pp. 1–108.

[99] Janert, P. (Oct. 2013) *Feedback Control for Computer Systems*, O'Reilly.

[100] (Feb. 28, 2021), "eBP runq latency metric," http://www.brendangregg.com/blog/2016-10-08/linux-bcc-runqlat.html.

[101] (Feb. 16, 2021), "GCP kubernetes best practices," https://cloud.google.com/blog/products/containers-kubernetes/kubernetes-best-practices-resource-requests-and-limits.

[102] (September, 22, 2021), "Pearson correlation coefficient," https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[103] (Feb. 28, 2021), "Prometheus Monitoring Tool," https://prometheus.io/.

[104] (September, 15, 2021), "Kubernetes Controller Pattern," https://kubernetes.io/docs/concepts/architecture/controller/.

[105] Zhou, X., X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding (2018) "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*.

[106] (March, 28, 2021), "Google Cloud Platform's Demo Microservice," https://github.com/GoogleCloudPlatform/microservices-demo.

[107] (September, 15, 2021), "Linux Kernel Scheduler sysctl_sched_latency," https://elixir.bootlin.com/linux/v4.6/source/kernel/sched/fair.c#L50.

[108] Urdaneta, G., G. Pierre, and M. van Steen (2009) "Wikipedia Workload Analysis for Decentralized Hosting," *Elsevier Computer Networks*, **53**(11), pp. 1830–1845.

[109] (March, 20, 2021), "Locust workload generator," https://locust.io/.

[110] (April, 22, 2021), "AWS EC2 On-demand Pricing," https://aws.amazon.com/ec2/pricing/on-demand/.

[111] (April, 22, 2021), "Netflix's Predictive Auto Scaling Engine," https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270.

[112] (April, 12, 2021), "Kubernetes Cluster Autoscaler," https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler.

[113] (September, 15, 2021), "Kubeless," https://kubeless.io/.

[114] (April, 25, 2021), "Scheduling Extensions in Kubernetes," https://github.com/akanso/extending-kube-scheduler.

[115] (Feb. 11, 2019), "Help Net Security. Average DDoS attack volumes grew by 194% in 12 months," https://www.helpnetsecurity.com/2019/02/11/ddos-attack-volumes-grew-by-194-in-12-months/.

[116] Osborne, C. (May 2, 2017), "The average DDoS attack cost for businesses rises to over $2.5 million," https://www.zdnet.com/article/the-average-ddos-attack-cost-for-businesses-rises-to-over-2-5m/.

[117] "http-slowloris," https://nmap.org/nsedoc/scripts/http-slowloris.html.

[118] Khandelwal, S. (Nov. 13, 2016), "Even A Single Computer Can Take Down Big Servers Using BlackNurse Attack," http://thehackernews.com/2016/11/dos-attack-server-firewall.html.

[119] Hansson, L., P. Hogh, B. Bachmann, K. Jorgensen, and D. Rand (2016), "The BlackNurse Attack," http://soc.tdc.dk/blacknurse/blacknurse.pdf.

[120] Staicu, C.-A. and M. Pradel (2018) "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers," in *27th USENIX Security Symposium (USENIX Security 18)*, USENIX Association, Baltimore, MD, pp. 361–376.
URL `https://www.usenix.org/conference/usenixsecurity18/presentation/staicu`

[121] "CloudFlare regex outage," https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[122] "CVE-2003-1564," https://nvd.nist.gov/vuln/detail/CVE-2003-1564.

[123] Wallen, J. (9 Oct 2019), "Kubernetes 'Billion Laughs' Vulnerability Is No Laughing Matter," https://thenewstack.io/kubernetes-billion-laughs-vulnerability-is-no-laughing-matter/.

[124] Davis, J. C., E. R. Williamson, and D. Lee (2018) "A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning," in *27th USENIX Security Symposium (USENIX Security 18)*, USENIX Association, Baltimore, MD, pp. 343–359.
URL `https://www.usenix.org/conference/usenixsecurity18/presentation/davis`

[125] (2019-08-13), "HTTP/2 Denial of Service Advisory," https://github.com/Netflix/security-bulletins/blob/master/advisories/third-party/2019-002.md.

[126] "Netflix DDoS for Microservices," https://www.infoq.com/news/2017/07/application-ddos-microservices.

[127] Demoulin, H., I. Pedisich, N. Vasilakis, V. Liu, B. Loo, and L. Phan (July 2019) "Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FINELAME," in *Proc. USENIX ATC*.

[128] "eBPF," https://github.com/iovisor/bcc.

[129] Davis, J. C., E. R. Williamson, and D. Lee (2018) "A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning," in *27th USENIX Security Symposium (USENIX Security 18)*, USENIX Association, Baltimore, MD, pp. 343–359.
URL `https://www.usenix.org/conference/usenixsecurity18/presentation/davis`

[130] Shan, Y., G. Kesidis, D. Fleck, and A. Stavrou (Oct. 2017) "Preliminary Study of Fission Defenses against Low-Volume DoS Attacks on Proxied Multiserver System," in *Proc. Malware Conference (MALCON)*, Peurto Rico.

[131] Jindal, A., H. Patel, A. Roy, S. Qiao, Z. Yin, R. Sen, and S. Krishnan (2019) "Peregrine: Workload Optimization for Cloud Query Engines," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, Association for Computing Machinery, New York, NY, USA, p. 416–427.
URL `https://doi.org/10.1145/3357223.3362726`

[132] "Kube8s taint and toleration," https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/.

[133] "Kube8s node affinity," https://kubernetes.io/docs/concepts/configuration/assign-pod-node/.

[134] Miller, D., Z. Qiu, and G. Kesidis (Sept. 2018) "Parsimonious Cluster-based Anomaly Detection (PCAD)," in *Proc. IEEE MLSP*, Aalborg, Denmark.

[135] Wang, L., M. Li, Y. Zhang, T. Ristenpart, and M. Swift (2018) "Peeking Behind the Curtains of Serverless Platforms," in *Proc. USENIX ATC*, Boston.

[136] Ginzburg, S. and M. J. Freedman (2020) "Serverless Isn't Server-Less: Measuring and Exploiting Resource Variability on Cloud FaaS Platforms," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pp. 43–48.

[137] Carreira, J., P. Fonseca, A. Tumanov, A. Zhang, and R. Katz (2019) "Cirrus: A serverless framework for end-to-end ML workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 13–24.

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

[139] Wang, L., M. Li, Y. Zhang, T. Ristenpart, and M. Swift (2018) "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 133–146.

[140] McGrath, G. and P. R. Brenner (2017) "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, IEEE, pp. 405–410.

[141] Lloyd, W., S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara (2018) "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," IEEE IC2E.

[142] Tariq, A., A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka (2020) "Sequoia: Enabling Quality-of-Service in Serverless Computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, p. 311–327.

[143] Zhang, T., D. Xie, F. Li, and R. Stutsman (2019) "Narrowing the gap between serverless and its state with storage functions," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–12.

[144] Spillner, J. (2017) "Practical tooling for serverless computing," in *Proceedings of the 10th International Conference on Utility and Cloud Computing*, pp. 185–186.

[145] "The Serverless Framework," `https://serverless.com`, last accessed on 5/28/2021.

[146] "What is Gloo Edge," `https://docs.solo.io/gloo-edge/latest/`, last accessed on 5/28/2021.

[147] Jonas, E., Q. Pu, S. Venkataraman, I. Stoica, and B. Recht (2017) "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 445–451.

[148] "Fn Project," `https://fnproject.io/`, last accessed on 5/28/2021.

[149] "Nimbella," `https://nimbella.com/platform`, last accessed on 5/28/2021.

[150] "Amazon EventBridge," `https://aws.amazon.com/eventbridge/`, last accessed on 9/28/2020.

[151] "TriggerMesh EveryBridge," `https://triggermesh.com/cloud_native_integration_platform/everybridge/`, last accessed on 9/28/2020.

[152] "Knative," `https://knative.dev/`, last accessed on 5/28/2021.

[153] "Knative Eventing," `https://knative.dev/docs/eventing/`, last accessed on 5/28/2021.

[154] Agache, A., M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa (2020) "Firecracker: Lightweight virtualization for serverless applications," in *17th Usenix symposium on networked systems design and implementation (NSDI 20)*, pp. 419–434.

[155] Thalheim, J., P. Bhatotia, P. Fonseca, and B. Kasikci (2018) "Cntr: Lightweight OS Containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 199–212.

[156] Akkus, I. E., R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt (2018) "SAND: Towards High-Performance Serverless Computing," in *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pp. 923–935.

[157] Cadden, J., T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo (2020) "SEUSS: skip redundant paths to make serverless fast," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–15.

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

[159] Mohan, A., H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov (2019) "Agile cold starts for scalable serverless," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[160] Carreira, J., S. Kohli, R. Bruno, and P. Fonseca (2021) "From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era," HotOS.

[161] Fuerst, A. and P. Sharma (2021) "FaasCache: keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 386–400.

[162] Mvondo, D., M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, and A. Tchana (2021) "OFC: an opportunistic caching system for FaaS platforms," in *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 228–244.

[163] Romero, F., G. I. Chaudhry, Í. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini (2021) "Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, ACM.

[164] Tang, Y. and J. Yang (2020) "Lambdata: Optimizing Serverless Computing by Making Data Intents Explicit," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, IEEE, pp. 294–303.

[165] Yu, T., Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen (2020) "Characterizing Serverless Platforms with ServerlessBench," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '20, ACM.

[166] Qiu, H., S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer (2020) "FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, pp. 805–825.

[167] Shahrad, M., J. Balkind, and D. Wentzlaff (2019) "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1063–1075.

[168] "AWS Lambda Pricing," `https://aws.amazon.com/lambda/pricing/`, last accessed on 5/28/2021.

[169] "GCP Cloud Function Pricing," `https://cloud.google.com/functions/pricing`, last accessed on 5/28/2021.

[170] Sampé, J., P. Garcia-Lopez, M. Sánchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona (2020) "Toward Multicloud Access Transparency in Serverless Computing," *IEEE Software*, **38**(1), pp. 68–74.

[171] Lin, W.-T., C. Krintz, and R. Wolski (2018) "Tracing function dependencies across clouds," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, pp. 253–260.

[172] Aske, A. and X. Zhao (2018) "Supporting multi-provider serverless computing on the edge," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, pp. 1–6.

[173] Kumar, R., M. Baughman, R. Chard, Z. Li, Y. Babuji, I. Foster, and K. Chard (2021) "Coding the computing continuum: Fluid function execution in heterogeneous computing environments," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, pp. 66–75.

[174] Baughman, M., R. Kumar, I. Foster, and K. Chard (2020) "Expanding Cost-Aware Function Execution with Multidimensional Notions of Cost," in *Proceedings of the 1st Workshop on High Performance Serverless Computing*, pp. 9–12.

[175] Kaur, K., D. S. Sharma, and D. K. S. Kahlon (2017) "Interoperability and portability approaches in inter-connected clouds: A review," *ACM Computing Surveys (CSUR)*, **50**(4), pp. 1–40.

[176] Rochwerger, B., D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, et al. (2009) "The reservoir model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, **53**(4), pp. 4–1.

[177] Mashayekhy, L., M. M. Nejad, and D. Grosu (2014) "Cloud federations in the sky: Formation game and mechanism," *IEEE Transactions on Cloud Computing*, **3**(1), pp. 14–27.

[178] Stoica, I. and S. Shenker (2021) "From cloud computing to sky computing," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 26–32.

[179] Wu, Z., M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha (2013) "SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 292–308.

[180] Bessani, A., M. Correia, B. Quaresma, F. André, and P. Sousa (2013) "DepSky: dependable and secure storage in a cloud-of-clouds," *ACM TOS*.

[181] Liu, H. H., Y. Wang, Y. R. Yang, H. Wang, and C. Tian (2012) "Optimizing cost and performance for content multihoming," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 371–382.

[182] Mukerjee, M. K., I. N. Bozkurt, B. Maggs, S. Seshan, and H. Zhang (2016) "The impact of brokers on the future of content delivery," in *proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pp. 127–133.

[183] Mukerjee, M. K., I. N. Bozkurt, D. Ray, B. M. Maggs, S. Seshan, and H. Zhang (2017) "Redesigning CDN-broker interactions for improved content delivery," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pp. 68–80.

[184] Hohlfeld, O., J. Rüth, K. Wolsing, and T. Zimmermann (2018) "Characterizing a meta-CDN," in *International Conference on Passive and Active Network Measurement*, Springer, pp. 114–128.

[185] Vasconcelos, A., L. Vieira, I. Batista, R. Silva, and F. Brasileiro (2019), "DistributedFaaS: Execution of containerized serverless applications in multi-cloud infrastructures," .

[186] Zhang, Y., A. Ghosh, and V. Aggarwal (2018) "Optimized Portfolio Contracts for Bidding the Cloud," *IEEE Transactions on Services Computing*.

[187] Park, H., G. R. Ganger, and G. Amvrosiadis (2020) "More IOPS for Less: Exploiting Burstable Storage in Public Clouds," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[188] Zheng, L., C. Joe-Wong, C. G. Brinton, C. W. Tan, S. Ha, and M. Chiang (2016) "On the viability of a cloud virtual service provider," *ACM SIGMETRICS Performance Evaluation Review*, **44**(1), pp. 235–248.

Vita

Ataollah Fatahi Baarzi

Education

Penn State University — State College, PA
- *Ph.D. in Computer Science and Engineering* — *Aug. 2017 – Dec 2021*

Sharif University of Technology — Tehran, Iran
- *Bachelor of Science in Computer Science and Engineering* — *Aug. 2012 – July. 2017*

Experiences

LinkedIn — Sunnyvale, CA
- *Systems and Infrastructure Engineering Intern* — *June 2021 - Sept. 2021*
  - Worked on runtime optimizations for ML model inference.

Microsoft — Sunnyvale, CA
- *Research Intern* — *May. 2020 - Aug. 2020*
  - Worked on differential data provenance in relational databases.

HashiCorp — San Francisco, CA
- *Research Intern* — *June. 2019 - Sept. 2019*
  - Worked on Consul scalability and overload controlling.

Penn State University — State College, PA
- *Graduate Research Assistant* — *Aug. 2017 - Present*
  - Working on improving the efficiency of cloud and distributed systems

Institute For Research in Fundamental Sciences (IPM) — Tehran, Iran
- *Undergraduate Research Assistant* — *Feb. 2016 - May 2017*
  - Worked on realistic SSD simulation and workload replay.

Publications

- **Ata Fatahi**, George Kesidis, "*SHOWAR*: Right-Sizing And Efficient Scheduling of Microservices," 12th ACM Symposium on Cloud Computing (**SoCC '21**)
- **Ata Fatahi**, George Kesidis, Carlee Joe-Wong, Mohammad Shahrad, " On Merits and Viability of Multi-Cloud Serverless Model," 12th ACM Symposium on Cloud Computing (**SoCC '21**)
- Aman Jain, **Ata Fatahi**, Nader Alfares, George Kesidis, Bhuvan Urgaonkar, Mahmut Kandemir, "SplitServe: Efficiently Splitting Apache Spark Jobs Across FaaS and IaaS," ACM **Middlware '20**.
- **Ata Fatahi**, George Kesidis, Daniel Fleck, Angelos Stavrou, "Microservices Made Attack-Resilient Using Unsupervised Service Fissioning," 13th European Workshop On Systems Security (**EuroSec '20**).
- **Ata Fatahi**, Timothy Zhu, Bhuvan Urgaonkar, "BurScale: Using Burstable Instances for Cost-Effective Autoscaling in the Public Cloud," 10th ACM Symposium on Cloud Computing (**SoCC '19**).
- Aman Jain, **Ata Fatahi**, Nader Alfares, George Kesidis, Bhuvan Urgaonkar, Mahmut Kandemir, "SplitServe: Efficiently Splitting Complex Workloads Across FaaS and IaaS," 10th ACM Symposium on Cloud Computing (**SoCC '19**).

Teaching Experiences

- Teaching Assistant, Penn State University
  - CMPSC 311, Systems Programming, Fall 2020
  - CMPSC 473, Operating Systems, Spring 2020
  - CMPSC 473, Operating Systems, Fall 2018
  - CMPSC 473, Operating Systems, Fall 2017