EFFICIENT RESOURCE MANAGEMENT OF CLOUD NATIVE SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Yanqi Zhang

May 2023

EFFICIENT RESOURCE MANAGEMENT OF CLOUD NATIVE SYSTEMS

Yanqi Zhang, Ph.D.

Cornell University 2023

Cloud native architecture has been a prevailing trend and is widely adopted by major online service providers including Netflix, Uber and WeChat. It enables applications to be structured as loosely-coupled distributed systems that can be developed and managed independently, and provide different programming models, namely microservice and serverless, to accommodate different user requirements. Specifically, microservices are a group of small services that collectively perform as a complete application. Each microservice implements a web server that handles specific business logic, and is usually packaged in a container that encapsulates its own runtime and dependencies. Microservice containers typically live for a long time and scale up or down to cope with load fluctuations as per user-specified policies. Serverless provides a further simplified approach to application development and deployment. It allows users to upload their application code as functions, without the need for explicit provisioning or management of containers, through an event-driven interface. Serverless containers are typically short-living 'one-off' containers handling a single request at a time. The billing of serverless is fine-grained and users only pay for the resources consumed by actual function execution.

Despite the popularity of cloud native systems, managing their resources efficiently is challenging. Cloud native applications consist of many component services with diverse resource requirements, posing a greater challenge compared to traditional monolithic applications. Furthermore, the backpressure effect caused by inter-service connections also complicates resource management. Lastly, although cloud-native relives users

from the burden of infrastructure management, cloud providers still need to provision and pay for the infrastructure to host cloud native applications, which incurs high cost.

This dissertation aims to tackle the challenge of efficient resource management for cloud-native systems and proposes three resource managers. First, we present **Sinan**, a machine learning (ML)-driven and service level agreement (SLA)-aware resource manager for microservices. Sinan uses a set of validated ML models to learn the per-service resource requirements , taking into account the effects of inter-service dependencies. Sinan's ML models predict the end-to-end latency of a given resource allocation, and the resource manager then chooses the optimal resource allocation that preserves the SLAs, based on the predictions. Sinan highlights the importance of a balanced training dataset that includes an equal share of SLA violations and satisfactions, for the effectiveness of ML models. Additionally, Sinan demonstrates that the system is flawed if the training dataset is dominated by either SLA satisfaction or violation. In order to obtain a balanced training dataset, Sinan explores different resource allocations with an algorithm inspired by multi-arm bandit (MAP).

Although Sinan outperforms traditional approaches such as autoscaling, it requires a lengthy exploration process and triggers a large number of SLA violations, hindering its practicality. Furthermore, the ML models are on the critical path of resource management decisions, limiting the speed and scalability of the system. To address these limitations, we further propose **Ursa**, a lightweight and scalable resource management framework for microservices. By investigating the backpressure-free conditions, Ursa allocates resources within the space that each service can be considered independent for the puropose of resource allocation. Ursa then uses an analytical model that decomposes the end-to-end latency into per-service latency, and maps per-service latency to individually checkable resource allocation threshold. To speed up the exploration process, Ursa explores as many independent microservices as possible across different request paths,

and swiftly stops exploration in case of SLA violations.

Finally, in order to reduce the infrastructure provisioning cost of cloud-native systems, we propose to leverage harvested resources in datacenter, which cloud providers provide at a massive discount. Orthogonal to the first two parts of the thesis which aim to reduce operation cost by providing the minimum amount of resources that do not compromise performance, this part aims to achieve cost reduction by using cheaper but less reliable resources. We use serverless as the target workload, and propose to run serverless platforms on low-priority Harvest VMs that grow and shrink to harvest all the unallocated CPU cores in their host servers. We quantify the challenges of running serverless on harvest VMs by characterizing the serverless workloads and Harvest VMs in production. We propose a series of policies that uses a mix of Harvest and regular VMs with different tradeoffs between reliability and efficiency, and design a serverless load balancer that is aware of VM evictions and resource variations in Harvest VMs. Our results show that adopting harvested resources improves efficiency and reduces cost significantly, and request failure rate caused by Harvest VM evictions is marginal.

BIOGRAPHICAL SKETCH

Yanqi Zhang received his master degree in Electrical and Computer Engineering from University of Wisconsin, Madison, where he worked with Prof. David Wood and Prof. Mikko Lipasti on computer architecture. He then joined the School of Electrical and Computer Engineering (ECE) at Cornell University as a Ph.D. student, where he has been supervised by Prof. Christina Delimitrou at the Computer Systems Laboratory. During his doctorate studies, Yanqi worked on various research problems including benchmarking, simulation, resource management and performance optimization of cloud native systems. He interned twice at Microsoft Research, in Fall 2019 and Summer 2022, where he was supervised by Dr. Sameh Elinkety, and worked closely with Dr. Ricardo Bianchini, Dr. Rodrigo Fonseca and Dr. Íñigo Goiri.

For my family and everyone who has faith in me.

ACKNOWLEDGEMENTS

As I reflect on my PhD journey, I find that I have been fortunate to have had the support and assistance of many intelligent and kind people, to whom I am deeply grateful.

I would like to convey my gratitude to all my committee members. I would like to thank my advisor Prof. Christina Delimitrou for her support and advice throughout my PhD journey, especially during the COVID pandemic which had me stationed in China for almost two years. Furthermore, I would like to thank her for introducing me to all kinds of opportunities in both academia and industry, and for allowing me complete freedom to steer the course of my research and to study topics that interest me. I would also like to thank my other two committee members from Cornell, Prof. Edward Suh and Prof. Lorenzo Alvisi, for their valuable feedbacks on my A exam. I also wish to extend my appreciation to Dr. Sameh Elinkety, my mentor at Microsoft Research, with whom I collaborated on a majority of the projects presented in this thesis. Dr. Elinkety's vast knowledge and unwavering patience as a researcher have always provided me with profound insights, and I am immensely grateful for the experience.

I would like to thank my collaborators at Microsoft Research, including Dr. Sameh Elinkety, Dr. Ricardo Bianchini, Dr. Rodrigo Fonseca and Dr. Íñigo Goiri. My internships at Microsoft Research proved to be a most valuable and rewarding experience during my doctoral studies. These bright minds imparted upon me the essence of computer systems research in industry, especially in terms of identifying high-impact research problems and pursuing simplicity and efficiency in problem-solving, setting a benchmark for good computer systems research.

I would like to thank my collaborators at Cornell, including Dr. Weizhe Hua, Zhuangzhuang Zhou, and Dr. Yu Gan. Weizhe is a motivated researcher who is always full of brilliant research ideas, from whom I learnt a lot about machine learning. Zhuangzhuang is always a smart collaborator with whom I have worked together on

multiple projects spanning both microservices and serverless. Yu and I were part of the first group of students to work with Prof. Delimitrou, and together, we have contributed to some of the lab's infrastructure projects. I would also like to extend my appreciation to all the members of the SAIL group, including Mingyu Liang, Nikita Lazarev, and Dr. Shuang Chen, among others, for our inspiring research discussions. Furthermore, I would like to express my gratitude to all my friends at CSL, for the wonderful time we have shared together, particularly on the badminton courts. I would also like to extend my thanks to friends outside Cornell for sharing a lot of pleasure in daily life.

Foremost, I would like to express my utmost gratitude to my parents, Jianping Zhang and Xuliang Zhang for their invaluable and unwavering support throughout my life, without which I would not be the person I am today.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vii
List of Tables . . . x
List of Figures . . . xi

1 Introduction 1
1.1 Background . . . 1
1.2 Related Work . . . 3
1.3 Contributions . . . 5
1.4 Thesis Organization . . . 7

2 Sinan: ML-Based and SLA-Aware Resource Management for Cloud Microservices 9
2.1 Introduction . . . 9
2.2 Overview . . . 11
2.2.1 Problem Statement . . . 11
2.2.2 Motivating Applications . . . 12
2.2.3 Management Challenges & the Need for ML . . . 14
2.2.4 Proposed Approach . . . 17
2.3 Machine Learning Models . . . 18
2.3.1 Latency Predictor . . . 20
2.3.2 Violation Predictor . . . 23
2.4 System Design . . . 25
2.4.1 System Architecture . . . 25
2.4.2 Resource Allocation Space Exploration . . . 26
2.4.3 Online Scheduler . . . 31
2.5 Evaluation . . . 33
2.5.1 Methodology . . . 33
2.5.2 Sinan's Accuracy and Speed . . . 34
2.5.3 Performance and Resource Efficiency . . . 36
2.5.4 Incremental Retraining . . . 39
2.5.5 Sinan's Scalability . . . 41
2.5.6 Explainable ML . . . 42
2.6 Conclusion . . . 45

3 Ursa: Lightweight Resource Management for Cloud-Native Microservices 46
3.1 Introduction . . . 46
3.2 Backpressure Effect . . . 49
3.3 Performance model . . . 53
3.4 Proof of Theorem 1 . . . 58

3.5 Allocation Space Exploration . . . 59
3.6 Design and Implementation . . . 62
3.7 Benchmarks . . . 65
3.8 Evaluation . . . 67
3.8.1 Experimental Setup . . . 68
3.8.2 Competing Approaches . . . 68
3.8.3 Online Exploration Overhead . . . 69
3.8.4 Model Accuracy . . . 71
3.8.5 Performance Comparison . . . 72
3.8.6 Control Plane Latency . . . 76
3.8.7 Adapting to Service Changes . . . 79
3.8.8 Summary . . . 79
3.9 Conclusion . . . 80

4 Faster and Cheaper Serverless Computing on Harvested Resources 82
4.1 Introduction . . . 82
4.2 Background and Related Work . . . 84
4.3 Characterization . . . 87
4.3.1 Harvest VMs . . . 88
4.3.2 Serverless Functions . . . 90
4.3.3 Implications . . . 93
4.4 Handling Evictions . . . 95
4.4.1 Methodology . . . 95
4.4.2 Combining Regular and Harvest VMs . . . 98
4.4.3 Running on Harvest VMs . . . 100
4.4.4 VM Migration/Snapshotting . . . 101
4.4.5 Conclusion . . . 101
4.5 Handling Resource Variability . . . 102
4.5.1 Join-the-Shortest-Queue (JSQ) . . . 102
4.5.2 Min-Worker-Set (MWS) . . . 103
4.6 Implementation . . . 105
4.6.1 OpenWhisk Architecture . . . 106
4.6.2 Harvest VM-Aware Load Balancing . . . 107
4.7 Evaluation . . . 109
4.7.1 Experiment Setup . . . 109
4.7.2 Impact of Load Balancing . . . 110
4.7.3 Impact of Resource Variability . . . 112
4.7.4 Cost vs Performance . . . 115
4.7.5 Harvest VMs vs Spot VMs . . . 116
4.7.6 Running on Real Harvest VMs . . . 119
4.7.7 Summary . . . 122
4.8 Conclusion . . . 123

5 Conclusions and Future Work **124**
5.1 Summary and Contributions 124
5.2 Open Problems 125
5.3 Future Work 127

Bibliography **129**

LIST OF TABLES

2.1 ML model parameters. 24
2.2 Resource allocation actions in Sinan. 32
2.3 RMSE, model size, and performance for three NNs — Batch size is 2048. Initial learning rates for MLP, LSTM, and CNN are 0.0001, 0.0005, and 0.001, respectively. All models are trained with a single NVidia Titan Xp. 34
2.4 The accuracy, number of trees, and total training time of Boosted Trees using a single NVidia Titan Xp. 35
2.5 Top-5 most critical tiers and resources for QoS with/without log synchronization in Social Network — SGrf and WUsr are social graph and write user, respectively. 44

3.1 Notations in the MIP. 56
3.2 SLAs of the social network. 66
3.3 SLAs of the media service. 67
3.4 SLAs of the video processing pipeline. 67
3.5 Online exploration overheads. 71
3.6 Control plane latency (ms). 78

4.1 Details on the two FaaS traces used in the chapter. 90
4.2 The examined serverless functions from FunctionBench [82] and their description. 110
4.3 Number of Harvest VMs with the same budget, based on the discount level. 115
4.4 Characteristics of the Harvest VMs, regular VMs, and Spot VMs used in the experiment in §4.7.6. 119
4.5 Latency reduction at multiple percentiles of Harvest and Spot VM clusters over regular VM clusters. 120

LIST OF FIGURES

2.1 Hotel reservation microservice architecture [66]. Client requests first reach a front-end webserver, and, depending on the type of requests, are then directed to logic tiers implementing functionality for searching hotels, completing hotel reservations, and getting recommendations on available hotels. At the right-most of the figure, the requests reach the back-end databases, implemented both with in-memory caching tiers (memcached), and persistent databases (MongoDB). 13

2.2 Social Network microservice architecture [66]. Client requests first reach Nginx, which works as frontend http servers. Then, depending on the type of user request, a number of logic, mid-tiers will be invoked to create a post, read a user's timeline and to follow/unfollow users. At the right-most of the figure, the requests reach the back-end databases, implemented both with in-memory caching tiers (memcached and Redis), and persistent databases (MongoDB). 14

2.3 The figure showcases the delayed queueing effect in microservices; QoS violations that are not detected eagerly (blue line), become unavoidable (red), even if later action is taken. 16

2.4 Multi-task NN overpredicts Social Network latency, due to the semantic gap between the QoS violation probability, a value between 0 and 1, and the latency, a value that is not strictly bounded. 19

2.5 Sinan's hybrid model, consisting of a CNN and a Boosted Trees (BT) model. The CNN extracts the latent variable (L_f) and predicts the end-to-end latency (y_L). The BT take the latent variable and proposed resource allocation, and predicts the probability of a QoS violation (p_V). . 22

2.6 Scale function $\phi(\cdot)$ with different k. 24

2.7 Sinan's system architecture. As user requests are being received, Sinan collects resource and performance metrics through Docker and Jaeger, inputs the collected metrics to the ML models, and uses the models' output to accordingly allocate resources for each tier. Allocation decisions are re-evaluated periodically online. 25

2.8 Training dataset latency distribution and ML training vs. validation error with respect to dataset latency range. The training dataset includes an approximately balanced set of samples between those that preserve and those that violate QoS. If the training dataset does not include any samples that violate QoS (500ms), both the CNN and XGBoost experience serious overfitting, greatly mispredicting latencies and QoS violations. 29

2.9 Comparison of predicted and true latency with (a) autoscaling and (b) random data collection schemes. When using autoscaling, the model significantly underestimates latency due to insufficient training samples of QoS violations, and causes large spikes in tail latency, forcing the scheduler to use all available resources to prevent further violations. On the other hand, when the model is trained using random profiling, it constantly overestimates latency and prohibits any resource reduction, leading to resource overprovisioning. 30
2.10 The mean and max CPU allocation, and the probability of meeting QoS for Sinan, Autoscaling, and PowerChief. 37
2.11 (Top) RPS, latency, and allocated resources per tier with Sinan for Social Network with 250 users. (Bottom) RPC, latency, and allocated resources per tier with diurnal load. For both scenarios, Sinan's predicted latency closely follows the end-to-end measured latency, avoiding QoS violations and excessive overprovisioning, while allocated resources per tier take into account the impact of microservice dependencies on end-to-end performance. 38
2.12 Training & validation RMSE of Fine-tunned CNNs with different amounts of samples. 40
2.13 Comparison of the average CPU allocation of four request mixes for Social Network on GCE. 42
2.14 99^{th} percentile latency distribution for four workload types of Social Network on GCE, managed by Sinan. 42
2.15 Tail latency for the Social Network application when Redis's logging is enabled (red) and disabled (blue). Sinan identified Redis as the source of unpredictable performance, and additionally determined the resources that were being saturated, pointing to the issue being in Redis's logging functionality. Disabling logging significanly improved performance, which is also reflected in that tier's importance, as far as meeting QoS is concerned, being reduced. 44

3.1 Inter-service communication methods. 48
3.2 Backpressure effect in a service chain. 49
3.3 Backpressure profiling engine architecture. 50
3.4 Profiling CPU threshold for no backpressure. 51
3.5 System Architecture of Ursa. 63
3.6 Estimated vs. measured latency for social network. 73
3.7 Estimated vs. measured latency for the video processing pipeline. . . . 74
3.8 SLA violation rate. 75
3.9 Average CPU allocation. 77
3.10 Ursa's CPU allocation under diurnal load. 78
3.11 99^{th} latency distribution of object-detect. 80

4.1 Distribution of the Harvest VM lifetime [36]. 88

4.2 Intervals between Harvest VM CPU changes. 89
4.3 Distribution of Harvest VM CPU change sizes and correlation of change sizes and change interval. 90
4.4 CDFs of the average and top percentiles of the invocation durations per application in the F_{Large} trace. 92
4.5 Invocation durations per app for F_{Large} and F_{Small}. 93
4.6 Durations of all invocations in the F_{Small} trace. 94
4.7 Durations of long applications invocations. 94
4.8 Harvest VM creations and eviction patterns. 97
4.9 Inter-arrival times for short vs. long apps. 99
4.10 Fraction of Harvest VM capacity versus acceptable percentile of per-app long invocations. 100
4.11 Architecture of our resource-variation-aware load balancing solution on OpenWhisk. The dotted lines show our modifications and components not present in vanilla OpenWhisk. 106
4.12 P99 latency across load balancing algorithms. 111
4.13 Cold start rate of MWS vs. JSQ. 112
4.14 Low percentile latency of MWS vs. JSQ. 113
4.15 Performance of harvest clusters in normal case (“Normal”), under frequent and significant CPU changes (“Active”), and of the “Dedicated” cluster. 114
4.16 Cold start rate against load for fixed budget. 115
4.17 Regular vs Harvest VMs with same budget. 116
4.18 Harvest VMs vs Spot VMs. Hx refers to Harvest VMs with base size of x CPUs, and Sx refers to Spot VMs with x CPUs. 118
4.19 Invocations in the combined function trace. 120
4.20 CPU number and cluster CPU utilization for Harvest VMs (upper left), regular VMs (upper right), and Spot VMs with 4 CPUs (lower left) and 48 CPUs (lower right). 121
4.21 Response latency comparing MWS on harvested resources to vanilla OpenWhisk running on dedicated resources. 122

CHAPTER 1

INTRODUCTION

1.1 Background

Cloud computing has become a prevailing trend, offering various benefits such as scalability to large systems, fine-grained elasticity, rapid time to market and data loss prevention [76, 64, 128, 24, 21]. In line with this trend, cloud native systems are also gaining popularity, as it enables applications to be built as loosely coupled distributed systems that are elastic, scalable and observable. Cloud native systems leverage the compute resources provided by Infrastructure as a Service (IaaS), and use containers [7, 98, 30, 25, 29, 124, 60] and lightweight virtual machines (VMs) [33, 95] as the smallest unit to package user applications. The virtualized sandboxes are typically managed by orchestration frameworks [13, 28] (sometimes with the help of service mesh [27]) which handle tasks including service discovery, load balancing, automatic scaling, resource bin packing, service self-healing, and automatic rollout/rollback. Moreover, the ecosystem also provides a range of frameworks to enable communication, data storage, monitoring and tracing of cloud native systems [3, 18, 16, 31, 17, 47, 22, 105]. These frameworks relive users from the burden of infrastructure development, allowing them to focus on rapid development and iteration of business logic. Cloud native systems provide different programming models, namely microservice and serverless, to accommodate different user requirements.

Microservices are small, loosely-coupled services that collectively perform as a complete application, and are widely adopted by major online service providers such as Twitter, Netflix and WeChat [6, 143, 131]. Each microservice implements a web server that handles specific business logic, and is usually packaged in a container that

encapsulates its own runtime and dependencies [23]. These microservice containers typically live for a long time and scale in or out automatically according to user specified policy to cope with load fluctuations. Communication between microservices are implemented via protocols including REST APIs, remote procedure calls (RPCs) and message queues [5, 10, 18, 3]. Compared to monolithic applications, microservices offer several benefits. Microservices enable flexible and rapid development by allowing each microservice to be developed and deployed independently, with its own programming language, framework, and dependencies. This flexibility shortens time-to-market and facilitates fast iteration, allowing users to make small and incremental updates to their business logic without the need for redeployment of the entire system. In addition, microservices enable fine-grained management, allowing each microservice to be configured, deployed, and scaled independently to meet its specific resource requirements. This results in lower operational costs, as users can add resources only to bottleneck services when faced with load spikes, rather than scaling the entire system as in monolithic systems.

Serverless provides a further simplified approach to application development and deployment. Serverless platforms enable users to upload their application codes as functions, without the need for explicit provisioning or management of containers and virtual machines. This is achieved through an event-driven interface, where user functions are executed in response to specific events, such as an HTTP request or a timer alarm. To accommodate the event-driven nature of serverless, the underlying platform dynamically allocates containers to execute user functions. Unlike microservice containers that usually are long-living and process multiple requests in parallel, serverless container are typically short-living 'one-off' containers handling a single request at a time. The billing of serverless is fine-grained and users are only billed for the resources consumed by actual function execution. The simplicity offered by serverless comes at the cost of

performance and efficiency. The event-driven framework incurs overhead in communication, as well as a container cold start, where the code and runtime of the function must be brought into memory from persistent storage. Despite these trade-offs, the simplicity of serverless makes it a compelling option for cloud programming from the user's perspective.

Despite their popularity, cloud native systems introduce new challenges for resource management. Cloud native systems usually need to satisfy SLA constraints defined in terms of end-to-end tail latency, which is unstable and hard to estimate. Compared to traditional monolithic systems, cloud native systems consist of many small component services that have diverse resource requirements. In addition, the backpressure effect, or dependencies between services caused by inter-service connections can lead to exacerbated queueing effects and potential cascading SLA violations, further complicating resource management [66, 143]. Lastly, although cloud native systems relieves users from the burden of infrastructure management, cloud providers still need to provision and pay for the infrastructures, which usually incurs high cost.

1.2 Related Work

We now review related work on cloud native system including both microservice and serverless, and resource management of cloud systems in general.

Microservices. The emergence of microservices has prompted efforts to benchmark and characterize them. Representative benchmarks include DeathstarBench [66] and ticket reservation [144], which implement several end-to-end user-facing applications. These benchmarks use RPC as the only inter-service communication method, and mostly perform lightweight text processing in the business logic. More recently,

Luo et al. [92] characterize microservices running on AliCloud and show that message queues are common in practice, accounting for 23% of all communication methods, and the performance of microservices is most sensitive to CPU interference. Related work [143, 117, 94, 122] also shows that cloud-native applications implement a variety of business logic, including ML workloads, image and video processing, etc. In terms of resource management for microservices, Wechat [143] manages microservices with overload control, by matching the throughput of the upstream and downstream services; PowerChief [141] dynamically power boosts bottleneck services in multi-phase applications, and Suresh et al. [129] leverage overload control and adopt deadline-based scheduling to improve tail latency in multi-tier workloads. Finally, Sriraman et al. [127] present an autotuning framework for microservice concurrency, and show the impact of threading decisions on application performance and responsiveness.

Serverless. Serverless computing has been the subject of extensive research aimed at expanding its range of applicable applications and enhancing its infrastructure. This research covers a wide range of areas, including: (a) scheduling policies for making serverless platforms cost-effective and performant [122, 78]; (b) performance-aware and cost-effective storage [83, 84, 103, 117]; (c) secure and light-weight container infrastructure [34, 106, 102, 137, 133, 125, 33]; (d) characterization of existing serverless workloads [122]; and (e) enabling applications to run in a serverless-native manner, including data processing and analytics [75, 112], video processing [63], ML training [46], DNA sequence visualization [88] and compilation [62].

Resource management for the cloud. Improving resource efficiency in cloud platforms in general is an important research area, and recent work [44, 61, 67, 70, 74, 77, 111, 132, 55, 80, 110] focuses on cluster scheduling frameworks, such as Kubernetes [13] and Apache YARN [134]. Resource central [53] uses a set of ML models to

predict VM performance metrics, such as CPU utilization and lifetime, Autopilot [119] uses an ensemble of models to tune container configurations, Ambati et al. [36] propose providing SLAs for resource harvesting VMs, and Narayanan et al. [104] propose to efficiently solve large-scale granular resource allocation problems by randomly partitioning them to smaller problems. However, these proposals are mainly applicable to single VMs or containers, rather than microservices with directed acyclic graph (DAG) topologies.

1.3 Contributions

This dissertation focuses on improving the resource efficiency of cloud native systems.

First, we tackle the problem of allocating the optimal amount of resources under SLA constraints. Specifically, we target microservice workloads and investigate both using machine learning (ML) and analytical models. We first propose **Sinan**, **an ML-driven, SLA-aware resource manager for microservices**. Sinan leverages a set of validated ML models to automatically learn the per-service resource requirements and the impact of inter-service dependencies from data, and assign appropriate resources to each service to preserve SLAs. Sinan highlights the importance of a balanced training dataset that includes an equal share of SLA violation and SLA satisfaction, for the effectiveness of ML models. To collect the balanced training dataset, Sinan employs an action space exploration algorithm inspired by multi-arm bandits to investigate the space of possible resource allocations. After collecting the proper training dataset, Sinan trains a set of models to predict the outcome of a resource allocation, both in the near-future and in the long term. The predictions are then used by the resource manager to decide the optimal resource allocation that preserves the SLA.

Sinan demonstrates that ML models are effective in predicting performance metrics such as end-to-end tail latency and can be used to guide resource allocations. However, the exploration process for collecting the balanced training dataset is time-consuming and triggers a large number of SLA violations, making it impractical to be performed online to track changes in user behavior or to cope with frequent business logic updates. In addition, the ML models are on the critical path of every resource management decision, limiting the speed and scalability of resource management. To address these limitations, we design **Ursa**, **a lightweight resource management framework for cloud native microservices**. First, we study how latency anomalies, or backpressure, propagate through different communication methods, due to improper resource allocation. The results show that backpressure is only significant in RPCs and is most pronounced in the parent service of the bottleneck microservice. Based on these findings, we propose a method to determine the resource utilization threshold for microservices that prevents backpressure. In a backpressure-free system, we develop a performance model based on mixed-integer programming (MIP). The model decomposes end-to-end latency SLA constraints into per-service latency constraints, and maps them to resource allocation thresholds for individual services. To speed up exploration while reducing SLA violations, Ursa explores as many independent services across different request paths, and swiftly stops exploration when violations occur or the resource utilization reaches the backpressure-free thresholds. In brief review of the first two parts of the thesis, we emphasize the benefits of analytical models in comparision to deep neural networks (DNNs) concerning resource management or control problems. Despite the demonstration of DNNs' capability in resolving black-box problems, analytical models designed with domain knowledge can be more performant, efficient and explainable, while requiring considerably smaller training dataset and incurring less exploration overheads.

Second, in order to reduce the infrastructure provisioning cost of cloud native sys-

tems, we propose to leverage harvested resources in datacenters which are provided at massive discounts. Orthogonal to the first two parts of the thesis, which aim to reduce operation cost by providing the minimum amount of resources without performance degradation, this part aims to achieve cost reduction by using cheaper but less reliable resources. For this part of the thesis we target serverless workloads because serverless workloads are short running and benefit the most from harvested resources. Specifically, we propose to **run serverless platforms on Harvest VMs** [36], a type of low-priority evictable VMs that grow and shrink to harvest all the unallocated CPU cores in their host servers. To understand the impact of evictions and of the variability in harvested resources on a serverless platform, we characterize both the serverless workload and the resources available to Harvest VMs using production traces. We contrast the duration of function executions with the lifetime of Harvest VMs and the durations over which resources are available for harvesting, and the analysis suggests a good match between serverless platforms and Harvest VMs. Additionally, we study how to adapt a serverless platform to run on harvested resources. To address Harvest VM evictions, we explore the space of regular and Harvest VMs mixes, for short- and long-running functions, and quantify the trade-off between cost and reliability. We find that even when running serverless solely on Harvest VMs, evictions cause at most 0.0015% of invocations to fail. To make this practical, we design and implement a load balancer for serverless platforms that places functions in VMs according to the availability of harvested resources, while keeping the function cold start rate low.

1.4 Thesis Organization

The rest of this dissertation is organized as follows. Chapter 2 presents Sinan, an ML-driven, SLA-aware resource manager for microservices. Chpater 3 presents Ursa, a

lightweight resource management framework for cloud native microservices. Chapter 4 presents the study of running serverless workloads on Harvest VMs, which makes serverless both faster and cheaper. Finally, Chapter 5 concludes the dissertation.

CHAPTER 2

SINAN: ML-BASED AND SLA-AWARE RESOURCE MANAGEMENT FOR CLOUD MICROSERVICES

2.1 Introduction

In recent years, cloud applications have progressively shifted from *monolithic* services to graphs with hundreds of single-purpose and loosely-coupled *microservices* [129, 66, 126, 32, 20, 6]. This shift is becoming increasingly pervasive, with large cloud providers, such as Amazon, Twitter, Netflix, and eBay having already adopted this application model [32, 20, 6].

Despite several advantages, such as modular and flexible development and rapid iteration, microservices also introduce new system challenges, especially in resource management, since the complex topologies of microservice dependencies exacerbate queueing effects, and introduce cascading Quality of Service (QoS) violations that are difficult to identify and correct in a timely manner [66, 143]. Current cluster managers are designed for monolithic applications or applications consisting of a few pipelined tiers, and are not expressive enough to capture the complexity of microservices [135, 97, 90, 91, 66, 109, 121, 123, 56, 58]. Given that an increasing number of production cloud services, such as EBay, Netflix, Twitter, and Amazon, are now designed as microservices, addressing their resource management challenges is a pressing need [32, 20, 66].

We take a data-driven approach to tackle the complexity microservices introduce to resource management. Similar machine learning (ML)-driven approaches have been effective at solving resource management problems for large-scale systems in previous

work [58, 53, 119, 57]. Unfortunately, these systems are not directly applicable to microservices, as they were designed for monolithic services, and hence do not account for the impact of dependencies between microservices on end-to-end performance.

We present Sinan, a scalable and QoS-aware resource manager for interactive cloud microservices. Instead of tasking the user or cloud operator with inferring the impact of dependencies between microservices, Sinan leverages a set of validated ML models to automatically determine the impact of per-tier resource allocations on end-to-end performance, and assign appropriate resources to each tier.

Sinan first uses an efficient space exploration algorithm to examine the space of possible resource allocations, especially focusing on corner cases that introduce QoS violations. This yields a training dataset used to train two models: a Convolutional Neural Network (CNN) model for detailed short-term performance prediction, and a Boosted Trees model that evaluates the long-term performance evolution. The combination of the two models allows Sinan to both examine the near-future outcome of a resource allocation, and to account for the system's inertia in building up queues with higher accuracy than a single model examining both time windows. Sinan operates online, adjusting per-tier resources dynamically according to the service's runtime status and end-to-end QoS target. Finally, Sinan is implemented as a centralized resource manager with global visibility into the cluster and application state, and with per-node resource agents that track per-tier performance and resource utilization.

We evaluate Sinan using two end-to-end applications from DeathStarBench [66], built with interactive microservices: a social network and a hotel reservation site. We compare Sinan against both traditionally-employed empirical approaches, such as autoscaling [19], and previous research on multi-tier service scheduling based on queueing analysis, such as PowerChief [141]. We demonstrate that Sinan outperforms previous

work both in terms of performance and resource efficiency, successfully meeting QoS for both applications under diverse load patterns. On the simpler hotel reservation application, Sinan saves *25.9%* on average, and up to *46.0%* of the amount of resources used by other QoS-meeting methods. On the more complex social network service, where abstracting application complexity is more essential, Sinan saves *59.0%* of resources on average, and up to *68.1%*, essentially accommodating twice the amount of requests per second, without the need for more resources. We also validate Sinan's scalability through large-scale experiments on approximately 100 container instances on Google Compute Engine (GCE), and demonstrate that the models deployed on the local cluster can be reused on GCE with only minor adjustments instead of retraining.

Finally, we demonstrate the explainability benefits of Sinan's models, delving into the insights they can provide for the design of large-scale systems. Specifically, we use an example of Redis's log synchronization, which Sinan helped identify as the source of unpredictable performance out of tens of dependent microservices to show that the system can offer practical and insightful solutions for clusters whose scale make previous empirical approaches impractical.

2.2 Overview

2.2.1 Problem Statement

Sinan aims to manage resources for complex, interactive microservices with tail latency QoS constraints in a scalable and resource-efficient manner. Graphs of dependent microservices typically include tens to hundreds of tiers, each with different resource requirements, scaled out and replicated for performance and reliability. Section 2.2.2

describes some motivating examples of such services with diverse functionality used in this work; other similar examples can be found in [32, 20, 6, 126].

Most cluster managers focus on CPU and memory management [119, 53, 135]. Microservices are by design mostly stateless, hence their performance is defined by their CPU allocation. Given this, Sinan primarily focuses on allocating CPU resources to each tier [66], both at sub-core and multi-core granularity, leveraging Linux cgroups through the Docker API [7]. We also provision each tier with the maximum profiled memory usage to eliminate out of memory errors.

2.2.2 Motivating Applications

We use two end-to-end interactive applications from DeathStarBench [66]: a hotel reservation service, and a social network.

Hotel Reservation

The service is an online hotel reservation site, whose architecture is shown in Figure 2.1.

Functionality: The service supports searching for hotels using geolocation, placing reservations, and getting recommendations. It is implemented in Go, and tiers communicate over gRPC [10]. Data backends are implemented in `memcached` for in-memory caching, and `MongoDB`, for persistent storage. The database is populated with 80 hotels and 500 active users.

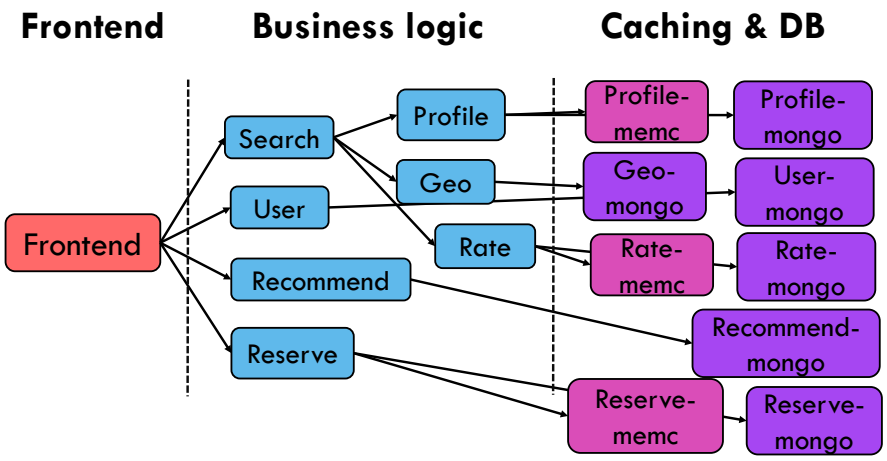


Figure 2.1: Hotel reservation microservice architecture [66]. Client requests first reach a front-end webserver, and, depending on the type of requests, are then directed to logic tiers implementing functionality for searching hotels, completing hotel reservations, and getting recommendations on available hotels. At the right-most of the figure, the requests reach the back-end databases, implemented both with in-memory caching tiers (memcached), and persistent databases (MongoDB).

Social Network

The end-to-end service implements a broadcast style social network with uni-directional follow relationships, shown in Figure 2.2. Inter-microservice messages use Apache Thrift RPCs [130].

Functionality: Users can create posts embedded with text, media, links, and tags to other users, which are then broadcasted to all their followers. The texts and images uploaded by users, specifically, go through image-filter (a CNN classifier) and text-filter services (an SVM classifier), and contents violating the service's ethics guidelines are rejected. Users can also read posts on their timelines. We use the `Reed98` [118] social friendship network to populate the user database. User activity follows the behavior of Twitter users reported in [86], and the distribution of post text length emulates Twitter's text length distribution [69].

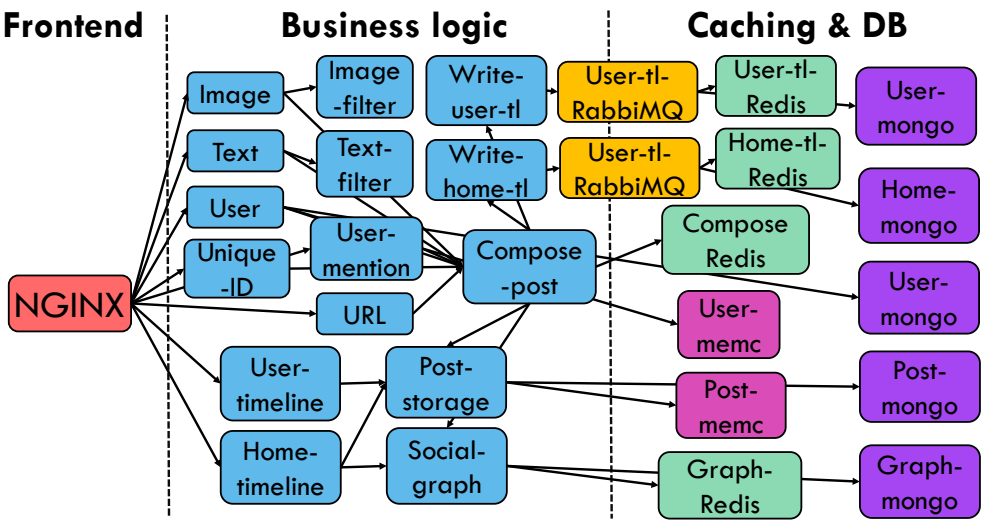


Figure 2.2: Social Network microservice architecture [66]. Client requests first reach Nginx, which works as frontend http servers. Then, depending on the type of user request, a number of logic, mid-tiers will be invoked to create a post, read a user's timeline and to follow/unfollow users. At the right-most of the figure, the requests reach the back-end databases, implemented both with in-memory caching tiers (memcached and Redis), and persistent databases (MongoDB).

2.2.3 Management Challenges & the Need for ML

Resource management in microservices faces four challenges.

1. Dependencies among tiers Resource management in microservices is additionally complicated by the fact that dependent microservices are not perfect pipelines, and hence can introduce backpressure effects that are hard to detect and prevent [66, 143]. These dependencies can be further exacerbated by the specific RPC and data store API implementation. Therefore, the resource scheduler should have a global view of the microservice graph and be able to anticipate the impact of dependencies on end-to-end performance.

2. System complexity Given that application behaviors change frequently, resource

management decisions need to happen online. This means that the resource manager must traverse a space that includes all possible resource allocations per microservice in a practical manner. Prior empirical approaches use resource utilization [19], or latency measurements [48, 58, 90] to drive allocation decisions. Queueing approaches similarly characterize the system state using queue lengths [141]. Unfortunately these approaches cannot be directly employed in complex microservices with tens of dependent tiers. First, microservice dependencies mean that resource usage across tiers is codependent, so examining fluctuations in individual tiers can attribute poor performance to the wrong tier. Similarly, although queue lengths are accurate indicators of a microservice's system state, obtaining exact queue lengths is hard. First, queues exist across the system stack from the NIC and OS, to the network stack and application. Accurately tracking queue lengths requires application changes and heavy instrumentation, which can negatively impact performance and/or is not possible in public clouds. Second, the application may include third-party software whose source code cannot be instrumented. Alternatively, expecting the user to express each tier's resource sensitivity is problematic, as users already face difficulties correctly reserving resources for simple, monolithic workloads, leading to well-documented underutilization [58, 114], and the impact of microservice dependencies is especially hard to assess, even for expert developers.

3. Delayed queueing effect Consider a queueing system with processing throughput T_o under a latency QoS target, like the one in Figure 2.3. T_o is a non-decreasing function of the amount of allocated resources R. For input load T_i, T_o should equal or slightly surpass T_i for the system to stably meet QoS, while using the minimum amount of resources R needed. Even when R is reduced, such that $T_o < T_i$, QoS will not be immediately violated, since queue accumulation takes time.

The converse is also true; by the time QoS is violated, the built-up queue takes a long

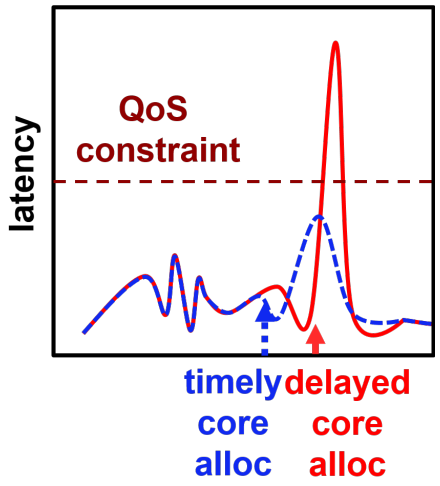


Figure 2.3: The figure showcases the delayed queueing effect in microservices; QoS violations that are not detected eagerly (blue line), become unavoidable (red), even if later action is taken.

time to drain, even if resources are upscaled immediately upon detecting the violation (red line). Multi-tier microservices are complex queueing systems with queues both across and within microservices. This delayed queueing effect highlights the need for ML to evaluate the long-term impact of resource allocations, and to proactively prevent the resource manager from reducing resources too aggressively, to avoid latency spikes with long recovery periods. To mitigate a QoS violation, the manager must increase resources proactively (blue line), otherwise the violation becomes unavoidable, even if more resources are allocated a posteriori.

4. Boundaries of resource allocation space Data collection or profiling are essential to the performance of any model. Given the large resource allocation space in microservices, it is essential for any resource manager to quickly identify the boundaries of that space that allow the service to meet its QoS, with the minimum resource amount [59], so that neither performance nor resource efficiency are sacrificed. Prior work often uses random exploration of the resource space [58, 90, 48] or uses prior system state as the

training dataset [65]. Unfortunately, while these approaches work for simpler applications, in microservices they are prone to covariant shift. Random collection blindly explores the entire space, even though many of the explored points may never occur during the system's normal operation, and may not contain any points close to the resource boundary of the service. On the contrary, data from operation logs are biased towards regions that occur frequently in practice but similarly may not include points close to the boundary, as cloud systems often overprovision resources to ensure that QoS is met. To reduce exploration overheads it is essential for a cluster manager to efficiently examine the necessary and sufficient number of points in the resource space that allow it to *just* meet QoS with the minimum resources.

2.2.4 Proposed Approach

These challenges suggest that empirical resource management, such as autoscaling [19] or queueing analysis-based approaches for multi-stage applications, such as PowerChief [141], are prone to unpredictable performance and/or resource inefficiencies. To tackle these challenges, we take a data-driven approach that abstracts away the complexity of microservices from the user, and leverages ML to identify the impact of dependencies on end-to-end performance, and make allocation decisions. We also design an efficient space exploration algorithm that explores the resource allocation space, especially boundary regions that may introduce QoS violations, for different application scenarios. Specifically, Sinan's ML models predict the end-to-end latency and the probability of a QoS violation for a resource configuration, given the system's state and history. The system uses these predictions to maximize resource efficiency, while meeting QoS.

At a high level, the workflow of Sinan is as follows: the data collection agent col-

lects training data, using a carefully-designed algorithm which addresses Challenge 4 (efficiently exploring the resource space). With the collected data, Sinan trains two ML models: a convolution neural network (CNN) model and a boosted trees (BT) model. The CNN handles Challenges 1 and 2 (dependencies between tiers and navigating the system complexity), by predicting the end-to-end tail latency in the near future. The BT model addresses Challenge 3 (delayed queueing effect), by evaluating the probability for a QoS violation further into the future, to account for the system's inertia in building up queues. At runtime, Sinan infers the instantaneous tail latency and the probability for an upcoming QoS violation, and adjusts resources accordingly to satisfy the QoS constraint. If the application or underlying system change at any point in time, Sinan retrains the corresponding models to account for the impact of these changes on end-to-end performance.

2.3 Machine Learning Models

The objective of Sinan's ML models is to accurately predict the performance of the application given a certain resource allocation. The scheduler can then query the model with possible resource allocations for each microservice, and select the one that meets QoS with the least necessary resources.

A straightforward way to achieve this is designing an ML model that predicts the immediate end-to-end tail latency as a function of resource allocations and utilization, since QoS is defined in terms of latency, and comparing the predicted latency to the measured latency during deployment is straightforward. The caveat of this approach is the delayed queueing effect described in Sec. 2.2.3, whereby the impact of an allocation decision would only show up in performance later. As a resolution, we experimented

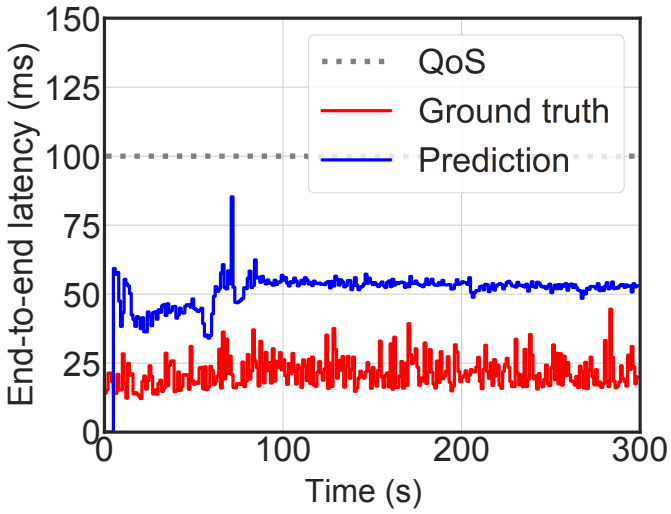


Figure 2.4: Multi-task NN overpredicts Social Network latency, due to the semantic gap between the QoS violation probability, a value between 0 and 1, and the latency, a value that is not strictly bounded.

with training a neural network (NN) to predict latency distributions over a future time window: for example, the latency for each second over the next five seconds. However, we found that the prediction accuracy rapidly decreased the further into the future the NN tried to predict, as predictions were based only on the collected current and past metrics (resource utilization and latency), which were accurate enough for immediate-future predictions, but were insufficient to capture how dependencies between microservices would cause performance to evolve later on.

Considering the difficulty of predicting latency further into the future, we set an alternative goal: predict the latency of the immediate future, such that imminent QoS violations are identified quickly, but only predict the probability of experiencing a QoS violation later on, instead of the exact latency of each decision interval. This binary classification is a much more contained problem than detailed latency prediction, and still conveys enough information to the resource manager on performance events, e.g., QoS violations, that may require immediate action in the present.

An intuitive method for this are multi-task learning NNs that predict the latency of

the next interval, and the QoS violation probability in the next few intervals. However, the multi-task NN considerably overpredicts tail latency and QoS violation probability, as shown in Figure 2.4. Note that the gap between prediction and ground truth does not indicate a constant difference, which could be easily learned by NNs with strong overfitting capabilities. We attribute the overestimation to interference caused by the semantic gap between the QoS violation probability, a value between 0 and 1, and the latency, a value that is not strictly bounded.

To address this, we designed a two-stage model: first, a CNN that predicts the end-to-end latency of the next timestep with high accuracy, and, second, a Boosted Trees (BT) model that estimates the probability for QoS violations further into the future, using the latent variable extracted by CNN. BT is generally less prone to overfitting than CNNs, since it has much fewer tunable hyperparameters than NNs; mainly the number of trees and tree depth. By using two separate models, Sinan is able to optimize each model for the respective objective, and avoid the overprediction issue of using a joint, expensive model for both tasks. We refer to the CNN model as the *short-term latency predictor*, and the BT model as the *long-term violation predictor*.

2.3.1 Latency Predictor

As discussed in Section 2.2.3, the CNN needs to account for both the dependencies across microservices, and the timeseries pattern of resource usage and application performance. Thus, both the application topology and the timeseries information are encoded in the input of the CNN. The input of the CNN includes the following three parts:

1. an "image" (3D tensor) consisting of per-tier resource utilization within a past time window. The y-axis of the "image" corresponds to different microservices,

with consecutive tiers in adjacent rows, the x-axis corresponds to the timeseries, with one timestep per column, and the z-axis (channels) corresponds to resource metrics of different tiers, including CPU usage, memory usage (resident set size and cache memory size) and network usage (number of received and sent packets), which are all retrieved from Docker's cgroup interface. Per-request tracing is not required.

2. a matrix of the end-to-end latency distribution within the past time window, and
3. the examined resource configuration for the next timestep, which is also encoded as a matrix.

In each convolutional (Conv) layer of the CNN, a convolutional kernel ($k \times k$ window) processes information of k adjacent tiers within a time window containing k timestamps. The first few Conv layers in the CNN can thus infer the dependencies of their adjacent tiers over a short time window, and later layers observe the entire graph, and learn interactions across all tiers within the entire time window of interest. The latent representations derived by the convolution layers are then post-processed together with the latency and resource configuration information, through concatenation and fully-connected (FC) layers to derive the latency predictions. In the remainder of this section, we first discuss the details of the network architecture, and then introduce a custom loss function that improves the prediction accuracy by focusing on the most important latency range.

As shown in Figure 2.5, the latency predictor takes as input the resource usage history (X_{RH}), the latency history (X_{LH}), and the resource allocation under consideration for the next timestep (X_{RC}), and predicts the end-to-end tail latencies (y_L) (95^{th} to 99^{th} percentiles) of the next timestep.

X_{RH} is a 3D tensor whose x-axis is the N tiers in the microservices graph, the y-axis

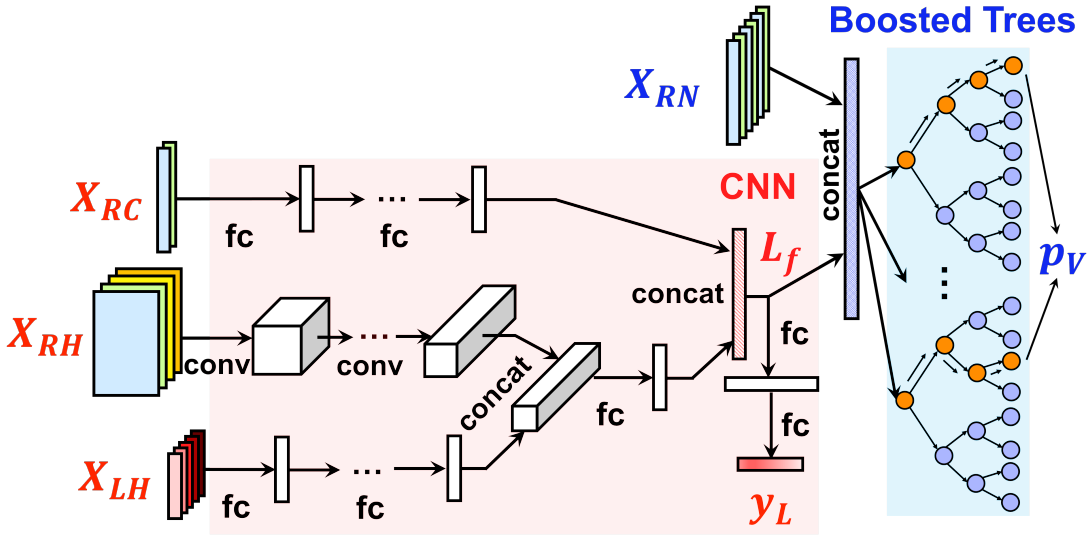


Figure 2.5: Sinan's hybrid model, consisting of a CNN and a Boosted Trees (BT) model. The CNN extracts the latent variable (L_f) and predicts the end-to-end latency (y_L). The BT take the latent variable and proposed resource allocation, and predicts the probability of a QoS violation (p_V).

is T timestamps ($T > 1$ accounts for the non-Markovian nature of microservice graph), and channels are F resource usage information related to CPU and memory. The set of necessary and sufficient resource metrics is narrowed down via feature selection. X_{RC} and X_{LH} are 2D matrices. For X_{RC}, the x-axis is the N tiers and the y-axis the CPU limit. For X_{RH}, the x-axis is T timestamps, and the y-axis are vectors of different latency percentiles (95^{th} to 99^{th}). The three inputs are individually processed with Conv and FC layers, and then concatenated to form the latent representation L_f, from which the predicted tail latencies L_f are derived with another FC layer.

The CNN minimizes the difference between predicted and actual latency, using the squared loss function below:

$$\mathcal{L}(X, \hat{y}, W) = \sum_{i}^{n} (\hat{y}_i - f_W(x_i))^2 \tag{2.1}$$

where $f_W(\cdot)$ represents the forward function of the CNN, $\hat{y}$ is the ground truth, and n is the number of training samples. Given the spiking behavior of interactive microservices

that leads to very high latency, the squared loss in Eq. 2.1 tends to overfit for training samples with large end-to-end latency, leading to latency overestimation in deployment. Since the latency predictor aims to find the best resource allocation within a tail latency QoS target, the loss should be biased towards training samples whose end-to-end latencies are $\leq QoS$. Therefore, we use a scaling function to scale both the predicted and actual end-to-end latency before applying the squared loss function. The scaling function ($\phi(\cdot)$) is:

$$\phi(x) = \begin{cases} x & x \leq t \\ t + \frac{x-t}{1+\alpha(x-t)} & x > t \end{cases} \tag{2.2}$$

where the latency range is $(0, t)$, and the hyper-parameter α can be tuned for different decay effects. Figure 2.6 shows the scaling function with $t = 100$ and $\alpha = 0.005, 0.01, 0.02$. It is worth mentioning that scaling end-to-end latencies only mitigates overfitting of the predicted latency for the next decision interval, and does not improve predictions further into the future, as described above. We implement all CNN models using MxNet [50], and trained them with Stochastic Gradient Descent (SGD).

2.3.2 Violation Predictor

The violation predictor addresses the binary classification task of predicting whether a given allocation will cause a QoS violation further in the future, to filter out undesirable actions. Ensemble methods are good candidates as they are less prone to overfitting. We use Boosted Trees [96], which realizes an accurate non-linear model by combining a series of simple regression trees. It models the target as the sum of trees, each of which maps features to a score. The final prediction is determined by accumulating scores across all trees.

Param	Definition
k	future timesteps in BT
T	past timesteps in CNN&BT
N	application tiers
M	latency percentiles
F	resource statistics
R	allocated resources

Table 2.1: ML model parameters.

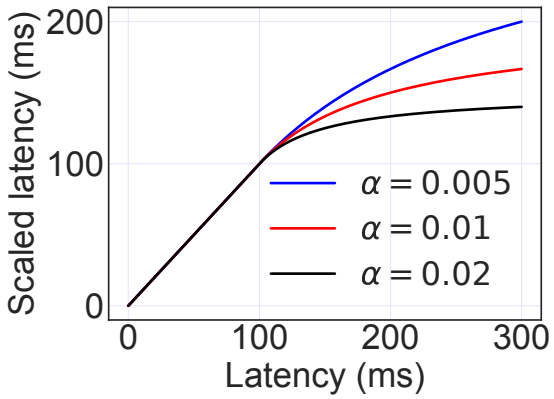


Figure 2.6: Scale function $\phi(\cdot)$ with different k.

To further reduce the computational cost and memory footprint of Boosted Trees, we reuse the compact latent variable L_f extracted from the CNN as its input. Moreover, since the latent variable L_f is significantly smaller than X_{RC}, X_{RH}, and X_{LH} in dimensionality, using L_f as the input also makes the model more resistant to overfitting.

Boosted Trees also takes resource allocations as input. During inference, we simply use the same resource configuration for the next k timesteps to predict whether it will cause a QoS violation k steps in the future. As shown in Figure 2.5, each tree leaf represents either a violation or a non-violation with a continuous score. For a given example, we sum the scores for all chosen violation (s_V) and non-violation leaves (s_V) from each tree. The output of BT is the predicted probability of QoS violation (p_V), which can be calculated as $p_V = \frac{e^{sV}}{e^{sV}+e^{sNV}}$. For the violation predictor we leverage XGBoost [49], a gradient tree boosting framework that improves scalability using sparsity-aware approx-

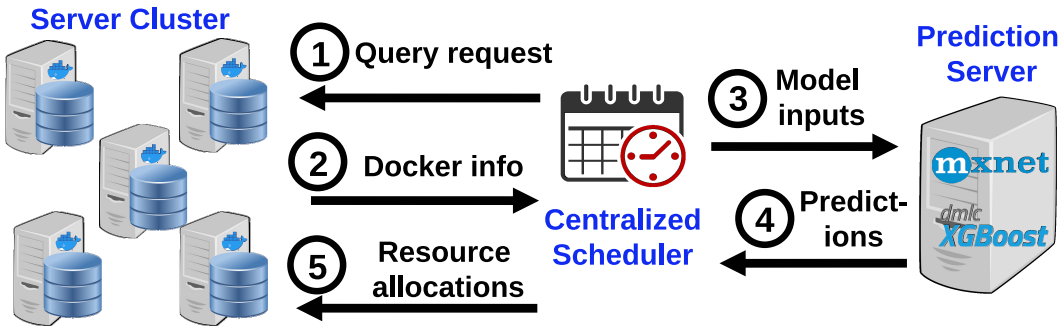


Figure 2.7: Sinan's system architecture. As user requests are being received, Sinan collects resource and performance metrics through Docker and Jaeger, inputs the collected metrics to the ML models, and uses the models' output to accordingly allocate resources for each tier. Allocation decisions are re-evaluated periodically online.

imate split finding.

We first train the CNN and then BT using the extracted latent variable from the CNN. The CNN parameters (number of layers, channels per layer, weight decay etc.) and XGBoost (max tree depth) are selected based on the validation accuracy.

2.4 System Design

We first introduce Sinan's overall architecture, and then discuss the data collection process, which is crucial to the effectiveness of the ML models, and Sinan's online scheduler.

2.4.1 System Architecture

Sinan consists of three components: a centralized scheduler, distributed operators deployed on each server/VM, and a prediction service that hosts the ML models. Fig-

ure 2.7 shows an overview of Sinan's architecture.

Sinan makes decisions periodically. In each 1s decision interval (consistent with the granularity at which QoS is defined), the centralized scheduler queries the distributed operators to obtain the CPU, memory, and network utilization of each tier in the previous interval. Resource usage is obtained from Docker's monitoring infrastructure, and only involves a few file reads, incurring negligible overheads. Aside from per-tier information, the scheduler also queries the API gateway to get user load statistics from the workload generator. The scheduler sends this data to the hybrid ML model, which is responsible for evaluating the impact of different resource allocations. Resource usage across replicas of the same tier are averaged before being used as inputs to the models. Based on the model's output, Sinan chooses an allocation vector that meets QoS using the least necessary resources, and communicates its decision to the per-node agents for enforcement.

Sinan focuses on compute resources, which are most impactful to microservice performance. Sinan explores sub-core allocations in addition to allocating multiple cores per microservice to avoid resource inefficiencies for non-resource demanding tiers, and enable denser colocation.

2.4.2 Resource Allocation Space Exploration

Representative training data is key to the accuracy of any ML model. Ideally, test data encountered during online deployment should follow the same distribution as the training dataset, so that covariate shift is avoided. Specifically for our problem, the training dataset needs to cover a sufficient spectrum of application behaviors that are likely to occur during online deployment. Because Sinan tries to meet QoS without sacrificing

resource efficiency, it must efficiently explore the boundary of the resource allocation space, where points using the minimum amount of resources under QoS reside. We design the data collection algorithm as a multi-armed bandit process [68], where each tier is an independent arm, with the goal of maximizing the knowledge of the relationship between resources and end-to-end QoS.

The data collection algorithm approximates the running state of the application with a tuple (rps, lat_{cur}, lat_{diff}), where rps is the input requests per second, lat_{cur} is the current tail latency, and lat_{diff} is the tail latency difference from the previous interval, to capture the rate of consuming or accumulating queues. Every tier is considered as an arm that can be played independently, by adjusting its allocated resources. For each tier, we approximate the mapping between its resources and the end-to-end QoS as a Bernoulli distribution, with probability p of meeting the end-to-end QoS, and we define our information gain from assigning certain amount of resources to a tier, as the expected reduction of confidence interval of p for the corresponding Bernoulli distribution. At each step for every tier, we select the operation that maximizes the information gain, as shown in Eq. 2.3, where op_T^s is an action selected for tier T at running state s, n are the samples collected for the resulting resource assignment after applying op on tier T at state s, p is the previously-estimated probability of meeting QoS, and p_+ and p_- are the newly-estimated probabilities of meeting QoS, when the new sample meets or violates QoS respectively. Each operation's score is multiplied by a predefined coefficient C_{op} to encourage meeting QoS and reducing overprovisioning.

$$\begin{aligned} op_T^s = \arg\max_{op} C_{op} \cdot (& \sqrt{\frac{p(1-p)}{n}} - p\sqrt{\frac{p_+(1-p_+)}{n+1}} \\ & -(1-p)\sqrt{\frac{p_-(1-p_-)}{n+1}}) \end{aligned} \tag{2.3}$$

By choosing operations that maximize Equation. 2.3, the data collection algorithm is

incentivized to explore the boundary points that meet QoS with the minimum resource amount, since exploring allocations that definitely meet or violate QoS (with $p = 1$ or $p = 0$) has at most 0 information gain. Instead, the algorithm prioritizes exploring resource allocations whose impact on QoS is nondeterministic, like those with $p = 0.5$. It is also worth noting that the state encoding and information gain definition are simplified approximations of the actual system, with the sole purpose of containing the exploration process in the region of interest. Eventually, we rely on ML to extract the state representation that incorporates inter-tier dependencies in the microservice graph.

To prune the action space, Sinan enforces a few rules on both data collection and online scheduling. First, the scheduler is only allowed to select out of a predefined set of operations. Specifically in our setting, the operations include reducing or increasing the CPU allocation by 0.2 up to 1.0 CPU, and increasing or reducing the total CPU allocation of a service by 10% or 30%. These ratios are selected according to the AWS step scaling tutorial [19]; as long as the granularity of CPU allocations does not change, other resource ratios also work without retraining the model. Second, an upper limit on CPU utilization is enforced on each tier, to avoid overly aggressive resource downsizing that can lead to long queues and dropped requests. Third, when end-to-end tail latency exceeds the expected value, Sinan disables resource reclamations so that the system can recover as soon as possible. A subtle difference from online deployment is that the data collection algorithm explores resource allocations in the $[0, QoS + \alpha]$ tail latency region, where α is a small value compared to QoS. The extra α allows the data collection process to explore allocations that cause slight QoS violations without the pressure of reverting to states that meet QoS immediately, such that the ML models are aware of boundary cases, and avoid them in deployment. In our setting α is 20% of QoS empirically, to adequately explore the allocation space, without causing the tail latency distribution to deviate too much from values that would be seen in deployment. Collecting data

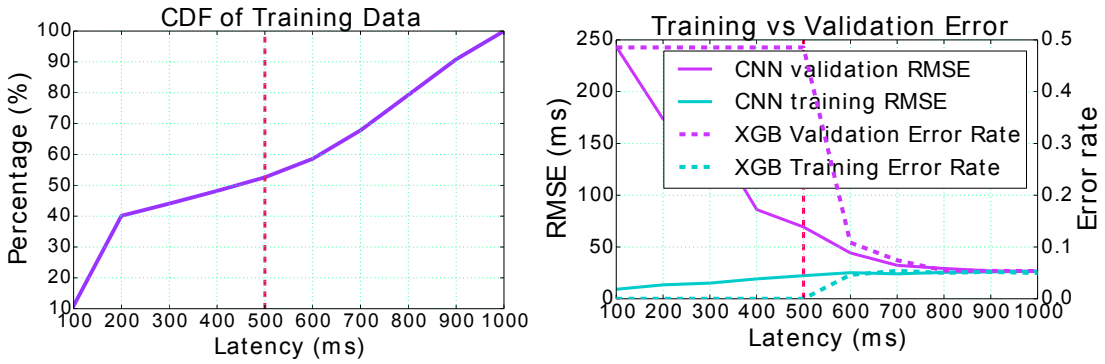
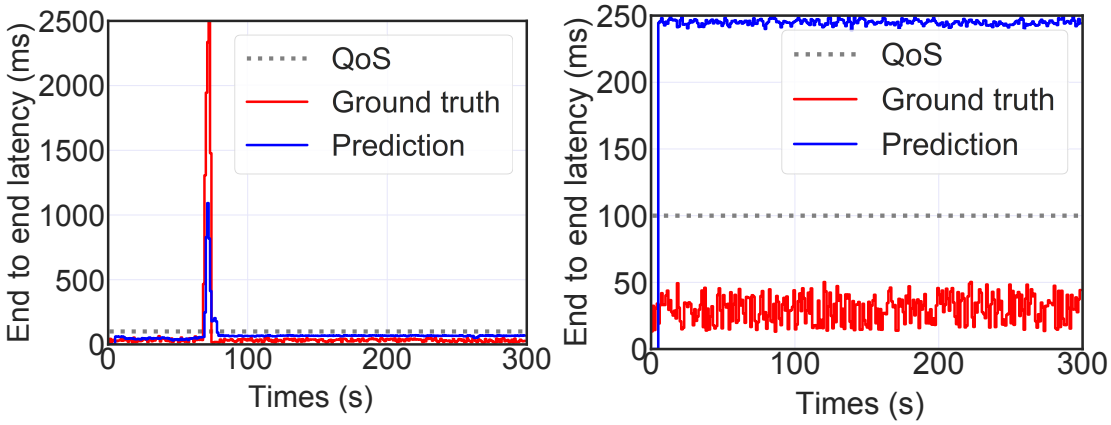


Figure 2.8: Training dataset latency distribution and ML training vs. validation error with respect to dataset latency range. The training dataset includes an approximately balanced set of samples between those that preserve and those that violate QoS. If the training dataset does not include any samples that violate QoS (500ms), both the CNN and XGBoost experience serious overfitting, greatly mispredicting latencies and QoS violations.

exclusively when the system operates nominally, or randomly exploring the allocation space does not fulfill these requirements.

Figure 2.8 shows the latency distribution in the training dataset, and how the training and validation error of the model changes with respect to the latency range observed in the training dataset, for the Social Network application. In the second figure, the x-axis is the latency of samples in the training dataset, the left y-axis is the root mean squared error RMSE of the CNN, and the right y-axis represents the classification error rate of XGBoost. Each point's y-axis value is the model's training and validation error when trained only with data whose latency is smaller than the corresponding x-value. If the training dataset does not include any samples that violate QoS (500ms), both the CNN and XGBoost experience serious overfitting, greatly mispredicting latencies and QoS violations.

Figure 2.9 shows data collected using data collection mechanisms that do not curate the dataset's distribution. Specifically, we show the prediction accuracy when the training dataset is collected when autoscaling is in place (a common resource manage-



(a) Autoscaling data collection.

(b) Random data collection.

Figure 2.9: Comparison of predicted and true latency with (a) autoscaling and (b) random data collection schemes. When using autoscaling, the model significantly underestimates latency due to insufficient training samples of QoS violations, and causes large spikes in tail latency, forcing the scheduler to use all available resources to prevent further violations. On the other hand, when the model is trained using random profiling, it constantly overestimates latency and prohibits any resource reduction, leading to resource overprovisioning.

ment scheme in most clouds), and when resource allocations are explored randomly. As expected, when using autoscaling, the model does not see enough cases that violate QoS, and hence seriously underestimates latency and causes large spikes in tail latency, forcing the scheduler to use all available resources to prevent further violations. On the other hand, when the model is trained using random profiling, it constantly overestimates latency and prohibits any resource reduction, highlighting the importance of jointly designing the data collection algorithms and the ML models.

Incremental and Transfer Learning: Incremental retraining can be applied to accommodate changes to the deployment strategy or microservice updates. In deployment, retraining can be triggered periodically in the background or when prediction accuracy drops below expected thresholds. In cases where the topology of the microservice graph is not impacted, such as hardware updates and change of public cloud provider, transfer learning techniques such as fine tune can be used to train the ML models in the back-

ground with newly collected data. If the topology is changed, the CNN needs to be modified to account for removed and newly-added tiers.

Additional resources: Sinan can be extended to other system resources. Several resources, such as network bandwidth and memory capacity act like thresholds, below which performance degrades dramatically, e.g., network bandwidth [48], or the application experiences out of memory errors, and can be managed with much simpler models, like setting fixed thresholds for memory usage, or scaling proportionally with respect to user load for network bandwidth.

2.4.3 Online Scheduler

During deployment, the scheduler evaluates resource allocations using the ML models, and selects appropriate allocations that meet the end-to-end QoS without overprovisioning.

Evaluating all potential resource allocations online would be prohibitively expensive, especially for complex microservice topologies. Instead, the scheduler evaluates a subset of allocations following the set of heuristics shown in Table 2.2. For scaling down operations, the scheduler evaluates reducing CPU allocations of single tiers, and batches of tiers, e.g., scaling down the k tiers with lowest cpu utilization, $1 < k \leq N$, N being the number of tiers in the microservice graph. When scaling up is needed, the scheduler examines the impact of scaling up single tiers, all tiers, or the set of tiers that were scaled down in the past t decision intervals, $1 < t < T$ with T chosen empirically. Finally, the scheduler also evaluates the impact of maintaining the current resource assignment.

The scheduler first excludes operations whose predicted tail latency is higher than

Table 2.2: Resource allocation actions in Sinan.

Category	Actions
Scale Down	Reduce CPU limit of 1 tier
Scale Down Batch	Reduce CPU limit of k least utilized tiers, $(1 < k \leq N)$
Hold	Keep current resource allocation
Scale Up	Increase CPU limit of 1 tier
Scale Up All	Increase CPU limit of all tiers
Scale Up Victim	Increase CPU limit of recent victim tiers, that are scaled down in previous t cycles

$QoS - RMSE_{valid}$. Then it uses the predicted violation probability to filter out risky operations, with two user-defined thresholds, p_d and p_u ($p_d < p_u$). These thresholds are similar to those used in autoscaling, where the lower threshold triggers scaling down and the higher threshold scaling up; the region between the two thresholds denotes stable operation, where the current resource assignment is kept. Specifically, when the violation probability of holding the current assignment is smaller than p_u, the operation is considered acceptable. Similarly, if there exists a scale down operation with violation probability lower than p_d, the scale down operation is also considered acceptable. When the violation probability of the hold operation is larger than p_u, only scaling up operations with violation probabilities smaller than p_u are acceptable; if no such actions exist, all tiers are scaled up to their max amount. We set p_u such that the validation study's false negatives are no greater than 1% to eliminate QoS violations, and p_d to a value smaller than p_u that favors stable resource allocations, so that resources do not fluctuate too frequently unless there are significant fluctuations in utilization and/or user demand. Among all acceptable operations, the scheduler selects the one requiring the least resources.

The scheduler also has a safety mechanism for cases where the ML model's predicted latency or QoS violation probability deviate significantly from the ground truth. If a mispredicted QoS violation occurs, Sinan immediately upscales the resources of all tiers. Additionally, given a trust threshold for the model, whenever the number of latency prediction errors or missed QoS violations exceeds the thresholds, the scheduler reduces its trust in the model, and becomes more conservative when reclaiming resources. In practice, Sinan never had to lower its trust to the ML model.

2.5 Evaluation

We first evaluate Sinan's accuracy, and training and inference time, and compare it to other ML approaches. Second, we deploy Sinan on our local cluster, and compare it against autoscaling [19], a widely-deployed empirical technique to manage resources in production clouds, and PowerChief [141], a resource manager for multi-stage applications that uses queueing analysis. Third, we show the incremental retraining overheads of Sinan. Fourth, we evaluate Sinan's scalability on a large-scale Google Compute Engine (GCE) cluster. Finally, we discuss how interpretable ML can improve the management of cloud systems.

2.5.1 Methodology

Benchmarks: We use the Hotel Reservation and Social Network benchmarks described in Section 2.2.2. QoS targets are set with respect to 99% end-to-end latency, 200ms for `Hotel Reservation`, and 500ms for `Social Network`.

Deployment: Services are deployed with Docker Swarm, with one microservices per

Table 2.3: RMSE, model size, and performance for three NNs — Batch size is 2048. Initial learning rates for MLP, LSTM, and CNN are 0.0001, 0.0005, and 0.001, respectively. All models are trained with a single NVidia Titan Xp.

Apps	Models	Train &Val. RMSE (ms)		Model size (KB)	Train & Inference speed (ms/batch)	
Hotel Reservation	MLP	**17.8**	**18.9**	**1433**	1.9	**3.7**
	LSTM	17.7	18.1	384	**1.3**	**3.2**
	CNN	**14.2**	**14.7**	**68**	**4.5**	3.5
Social Network	MLP	**32.3**	**34.4**	**4300**	6.4	**5.9**
	LSTM	29.3	30.7	404	**4.5**	**5.6**
	CNN	**25.9**	**26.4**	**144**	**16.0**	5.7

container for deployment ease. *Locust* [14] is used as the workload generator for all experiments.

Local cluster: The cluster has four 80-core servers, with 256GB of RAM each. We collected 31302 and 58499 samples for Hotel Reservation and Social Network respectively, using our data collection process, and split them into training and validation sets with a 9:1 ratio, after random shuffling. The data collection agent runs for 16 hours and 8.7 hours for Social Network and Hotel Reservation respectively, and collecting more training samples do not further improve accuracy.

GCE cluster: We use 93 container instances on Google Compute Engine (GCE) to run Social Network, with several replicas per microservice tier. 5900 extra training samples are collected on GCE for the transfer learning.

2.5.2 Sinan's Accuracy and Speed

Table 2.3 compares the short-term ML model in Sinan (CNN) against a multilayer perceptron (MLP), and a long short-term memory (LSTM) network, which is traditionally

Table 2.4: The accuracy, number of trees, and total training time of Boosted Trees using a single NVidia Titan Xp.

Apps	Train & Val. accuracy (%)		Val. false pos. & neg.		# of trees	Total train time (s)
Hotel Reservation	94.4	94.1	3.2	3.1	229	2.3
Social Network	95.5	94.6	3.4	2.0	239	6.5

geared towards timeseries predictions. We rearrange the system history X_{RH} to be a 2D tensor with shape $T \times (F * N)$, and a 1D vector with shape $T * F * N$ for the LSTM and MLP models, respectively. To configure each network's parameters, we increase the number of fully-connected, LSTM, and convolutional layers, as well as the number of channels in each layer for the MLP, LSTM, and Sinan (CNN), until accuracy levels off. Sinan's CNN achieves the lowest RMSE, with the smallest model size. Although the CNN is slightly slower than the LSTM, its inference latency is within 1% of the decision interval (1s), which does not delay online decisions.

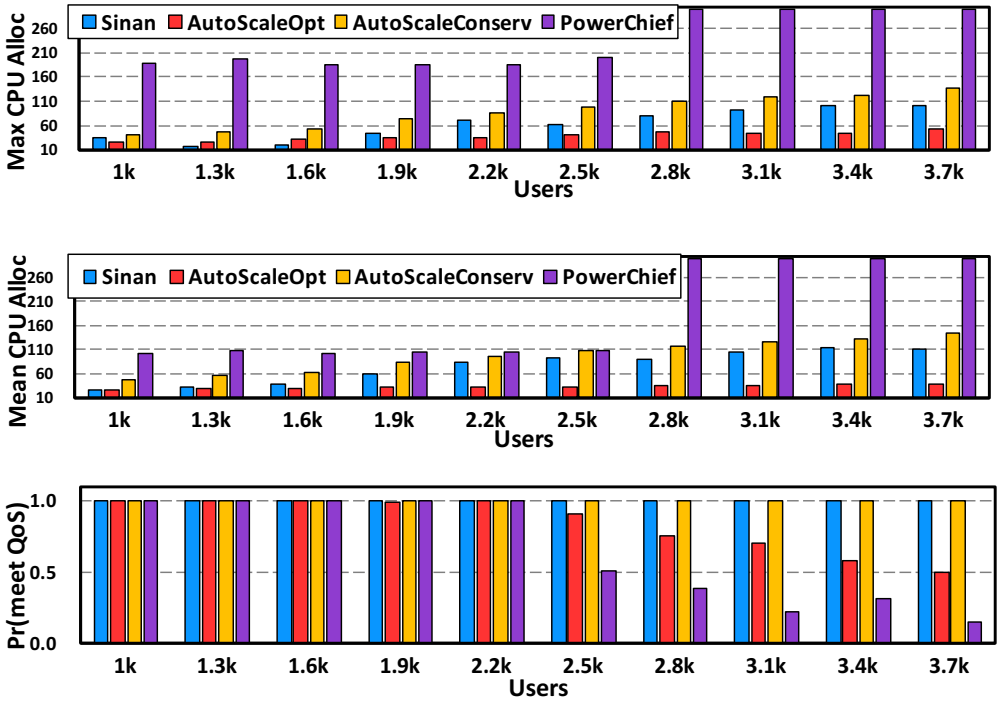
Table 2.4 shows a similar validation study for the Boosted Trees model. Specifically, we quantify the accuracy of anticipating a QoS violation over the next 5 intervals (5s), and the number of trees needed for each application. For both applications, the validation accuracy is higher than 94%, demonstrating BT's effectiveness in predicting the performance evolution in the near future. Sinan always runs on a single NVidia Titan XP GPU with average utilization below 2%.

2.5.3 Performance and Resource Efficiency

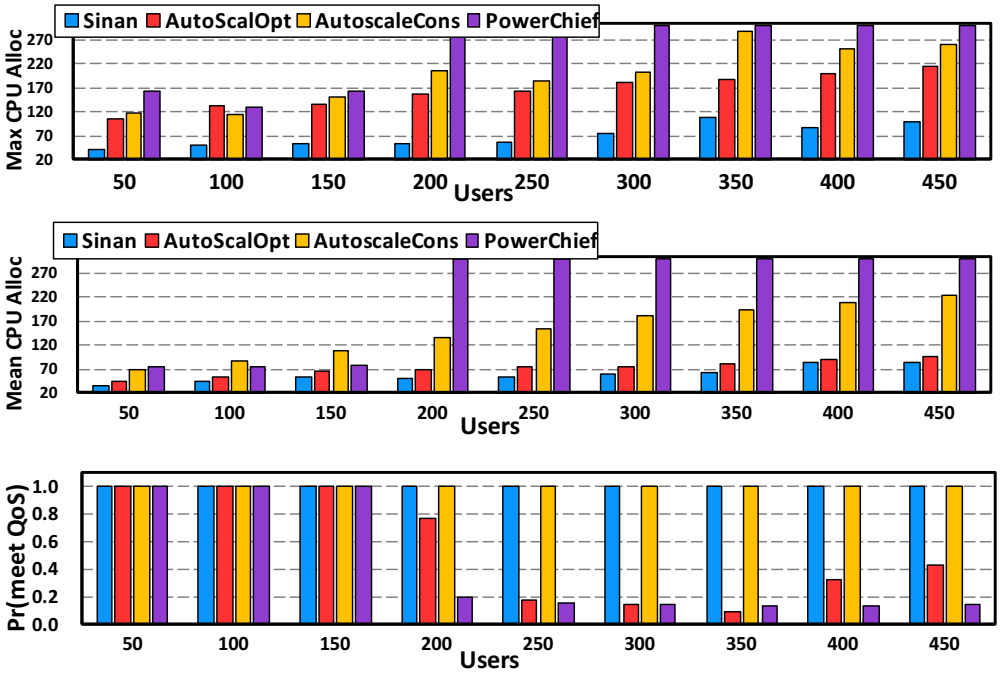
We now evaluate Sinan's ability to reduce resource consumption while meeting QoS on the local cluster. We compare Sinan against autoscaling and PowerChief [141]. We experimented with two autoscaling policies: AutoScaleOpt is configured according to [19], which increases resources by 10% and 30% when utilization is within $[60\%, 70\%)$ and $[70\%, 100\%]$ respectively, and reduces resources by 10% and 30% when utilization is within $[30\%, 40\%)$ and $[0\%, 30\%)$. AutoScaleCons is more conservative and optimizes for QoS, using thresholds tuned for the examined applications. It increases resources by 10% and 30% when utilization is within $[30\%, 50\%)$ and $[50\%, 100\%]$, and reduces resources by 10% when utilization is within $[0\%, 10\%)$. PowerChief is implemented as in [141], and estimates the queue length and queueing time ahead of each tier using network traces obtained through Docker.

For each service, we run 9 experiments with an increasing number of emulated users sending requests under a Poisson distribution with 1 RPS mean arrival rate. Figure 2.10 shows the mean and max CPU allocation, and the probability of meeting QoS across all studied mechanisms, where CPU allocation is the aggregate number of CPUs assigned to all tiers averaged over time, the max CPU allocation is the max of the aggregate CPU allocation over time, and the probability of meeting QoS is the fraction of execution time when end-to-end QoS is met.

For Hotel Reservation, only Sinan and AutoScaleCons meet QoS at all times, with Sinan additionally reducing CPU usage by 25.9% on average, and up to 46.0%. AutoScaleOpt only meets QoS at low loads, when the number of users is no greater than 1900. At 2200 users, AutoScaleOpt starts to violate QoS by 0.7%, and the probability of meeting QoS drops to 90.3% at 2800 users, and less than 80% beyond 3000 users. Similarly, PowerChief meets QoS for fewer than 2500 users, however the probability of



(a) Hotel reservation.



(b) Social network.

Figure 2.10: The mean and max CPU allocation, and the probability of meeting QoS for Sinan, Autoscaling, and PowerChief.

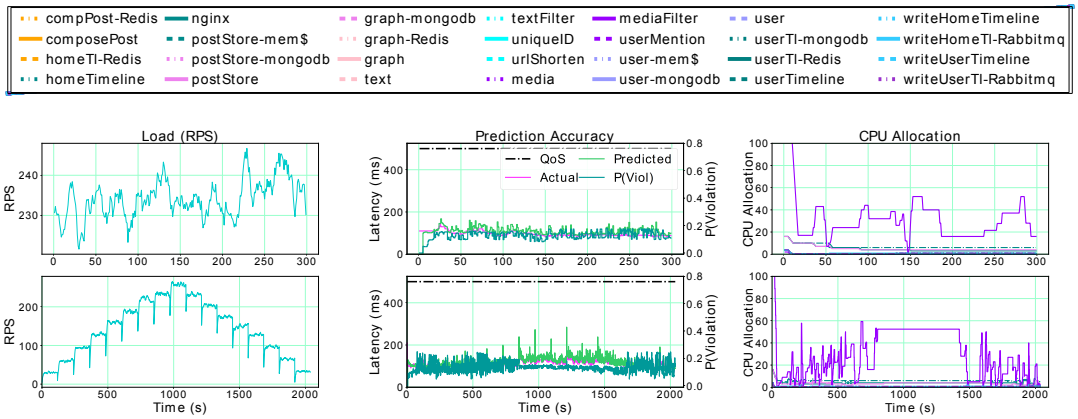


Figure 2.11: (Top) RPS, latency, and allocated resources per tier with Sinan for Social Network with 250 users. (Bottom) RPC, latency, and allocated resources per tier with diurnal load. For both scenarios, Sinan's predicted latency closely follows the end-to-end measured latency, avoiding QoS violations and excessive overprovisioning, while allocated resources per tier take into account the impact of microservice dependencies on end-to-end performance.

meeting QoS drops to 50.8% at 2800 users, and never exceeds 40% beyond 3000 users. AutoScaleOpt uses 53% the amount of resources Sinan requires on average, at the price of performance unpredictability, and PowerChief uses 2.57× more resources than Sinan despite violating QoS.

For the more complicated Social Network, Sinan's performance benefits are more pronounced. Once again, only Sinan and AutoScaleCons meet QoS across loads, while Sinan also reduces CPU usage on average by 59.0% and up to 68.1%. Both AutoScaleOpt and PowerChief only meet QoS for fewer than 150 users, despite using on average 1.26× and up to 3.75× the resources Sinan needs. For higher loads, PowerChief's QoS meeting probability is at most 20% above 150 users, and AutoscaleOpt's QoS meeting probability starts at 76.3% for 200 users, and decreases to 8.7% for 350 users.

By reducing both the average and max CPU allocation, Sinan can yield more resources to colocated tasks, improving the machine's effective utilization [90, 58, 91, 48]. There are three reasons why PowerChief cannot reduce resources similarly and leads to

QoS violations. First, as discussed in Sec. 2.2.3, the complex topology of microservices means that the tier with the longest igress queue, which PowerChief signals as the source of performance issues, is not necessarily the culprit but a symptom. Second, in interactive applications, queueing takes place across the system stack, including the NIC, OS kernel, network processing, and application, making precise queueing time estimations challenging, especially when tracing uses sampling. Finally, the stricter latency targets of microservices, compared to traditional cloud services, indicate that small fluctuations in queueing time can result in major QoS violations due to imperfect pipelining across tiers causing backpressure to amplify across the system.

Figure 2.11 shows the detailed results for Social Network, for 300 concurrent users under a diurnal load. The three columns each show requests per second (RPS), predicted latency vs. real latency and predicted QoS violation probability, and the realtime CPU allocation. As shown, Sinan's tail latency prediction closely follows the ground truth, and is able to react rapidly to fluctuations in the input load.

2.5.4 Incremental Retraining

We show the incremental retraining overheads of Sinan's ML models in three different deployment scenarios with the Social Network applications: switching to new server platforms (from the local cluster to a GCE cluster), changing the number of replicas (scale out factor) for all microservices except the backend databases (to avoid data migration overheads), and modifying the application design by introducing encryption in post messages uploaded by users (posts are encrypted with AES [54] before being stored in the databases). Instead of retraining the ML models from scratch, we use the previously-trained models on the local cluster, and fine-tune them using a small amount

of newly-collected data, with the initial learning rate λ being 1×10^{-5}, $\frac{1}{100}$ of the original λ value, in order to preserve the learnt weights in the original model and constrain the new solution derived by the SGD algorithm to be in a nearby region of the original one. The results are shown in Figure 2.12, in which the y-axis is the RMSE and the x-axis is the number of newly-collected training samples (unit being 1000). The RMSE values with zero new training samples correspond to the original model's accuracy on the newly collected training and validation set. In all three scenarios the training and validation RMSE converge, showing that incremental retraining in Sinan achieves high accuracy, without the overhead of retraining the entire model from scratch.

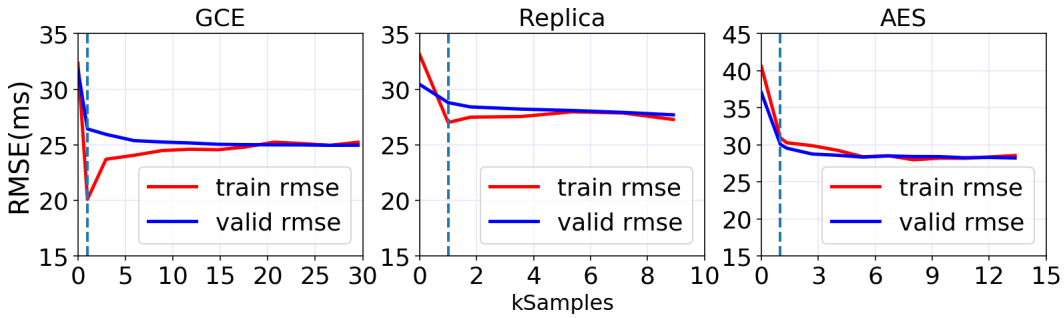


Figure 2.12: Training & validation RMSE of Fine-tunned CNNs with different amounts of samples.

In terms of new server platforms and different replica numbers, the original model already achieve a RMSE of 33.23ms and 33.1ms correspondingly, showing the generalizability of selected input features. The RMSE of original model, when directly applied to the modified application, is higher compared to the two other cases, reaching 40.56ms. In all of the three cases, the validation RMSE is siginificantly reduced with 1000 newly collected training samples (shown by the dotted lines in each figure), which translates to 16.7 minutes of profiling time. The case of GCE, different replica number and modified application stabilize with 5900 samples (1.6 hours of profiling), 1800 samples (0.5 hour of profiling) and 5300 samples (1.5 hours of profiling), and achieve training vs. validation RMSE of 24.8ms vs. 25.2ms, 27.5ms vs. 28.2ms, and 28.4ms vs.

28.3ms correspondingly.

2.5.5 Sinan's Scalability

We now show Sinan's scalability on GCE running Social Network. We use the fine-tuned model described in Section 2.5.4. Apart from the CNN, XGBoost achieves training and validation accuracy of 96.1% and 95.0%. The model's size and speed remain unchanged, since they share the same architecture with the local cluster models.

To further test Sinan's robustness to workload changes, we experimented with four workloads for Social Network, by varying request types. Some requests, like ComposePost involve the majority of microservices, and hence are more resource intensive, while others, like ReadUserTimeline involve a much smaller number of tiers, and are easier to allocate resources for. We vary the ratio of Compose-Post:ReadHomeTimeline:ReadUserTimeline requests; the ratios of the $W0$, $W1$, $W2$ and $W3$ workloads are 5:80:15, 10:80:10, 1:90:9, and 5:70:25, where $W0$ has the same ratio as the training set. The ratios are representative of different social media engagement scenarios [118]. The average CPU allocation and tail latency distribution are shown in Figure 2.13 and Figure 2.14. Sinan always meets QoS, adjusting resources accordingly. $W1$ requires the max compute resources (170 vCPUs for 450 users), because of the highest number of ComposePost requests, which trigger compute-intensive ML microservices.

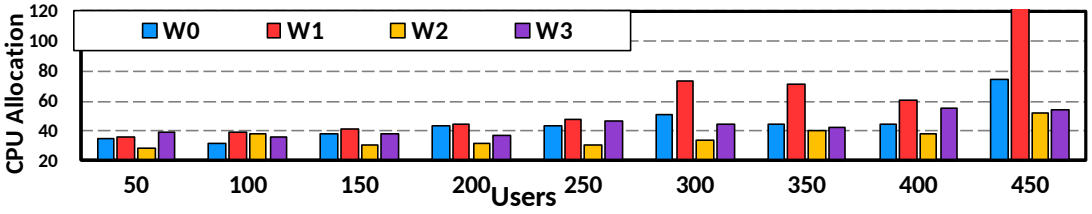


Figure 2.13: Comparison of the average CPU allocation of four request mixes for Social Network on GCE.

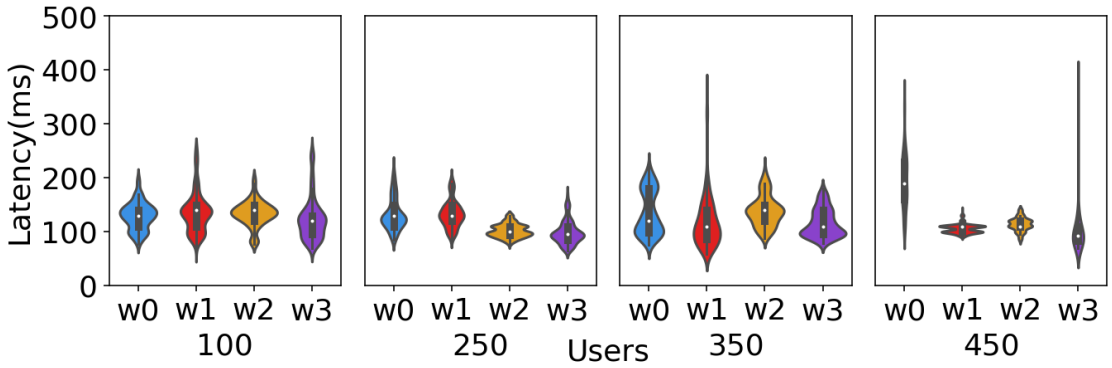


Figure 2.14: 99^{th} percentile latency distribution for four workload types of Social Network on GCE, managed by Sinan.

2.5.6 Explainable ML

For users to trust ML, it is important to interpret its output with respect to the system it manages, instead of treating ML as a black box. We are specifically interested in understanding what makes some features in the model more important than others. The benefits are threefold: 1) debugging the models; 2) identifying and fixing performance issues; 3) filtering out spurious features to reduce model size and speed up inference.

Interpretability Methods

For the CNN model, we adopt the widely-used ML interpretability approach LIME [116]. LIME interprets NNs by identifying their key input features which contribute most to predictions. Given an input X, LIME perturbs X to obtain a set of artificial

samples which are close to X in the feature space. Then, LIME classifies the perturbed samples with the NN, and uses the labeled data to fit a linear regression model. Given that linear regression is easy to interpret, LIME uses it to identify important features based on the regression parameters. Since we are mainly interested in understanding the culprit of the QoS violations, we choose samples X from the timesteps where QoS violations occur. We perturb the features of a given tier or resource by multiplying that feature with different constants. For example, to study the importance of MongoDB, we multiply its utilization history with two constants 0.5 and 0.7, and generate multiple perturbed samples. Then, we construct a dataset with all perturbed and original data to train the linear regression model. Last, we rank the importance of each feature by summing the value of their associated weights.

Interpreting the CNN

We used LIME to correct performance issues in Social Network [66], where tail latency experienced periods of spikes and instability despite the low load, as shown by the red line in Figure 2.15. Manual debugging is cumbersome, as it requires delving into each tier, and potentially combinations of tiers to identify the root cause. Instead, we leverage explainable ML to filter the search space. First, we identify the top-5 most important tiers; the results are shown in the w/ Sync part of Table 2.5. We find that the most important tier for the model's prediction is social-graph Redis, instead of tiers with heavy CPU utilization, like `nginx`.

We then examine the importance of each resource metric for Redis, and find that the most meaningful resources are cache and resident working set size, which correspond to data from disk cached in memory, and non-cached memory, including stacks and heaps. Using these hints, we check the memory configuration and statistics of Redis,

Table 2.5: Top-5 most critical tiers and resources for QoS with/without log synchronization in Social Network — SGrf and WUsr are social graph and write user, respectively.

w/ Sync	**Tiers**	**SGrf Redis**	**post storage**	**WUsr timeline**	**SGrf MongoDB**	**SGrf**
	Weights	5109.9	1609.8	1503.1	849.7	482.7
	Resource utilization	**cache memory**	**RSS**	**# of cores**	**CPU utilization**	**received packets**
	Weights	15181.9	1576.1	658.5	322.7	20.0
w/o Sync	**Tiers**	**WUsr timeline**	**WUsr rabbitmq**	**SGrf MongoDB**	**SGrf**	**SGrf Redis**
	Weights	3948.6	3601.6	1794.0	600.9	451.7

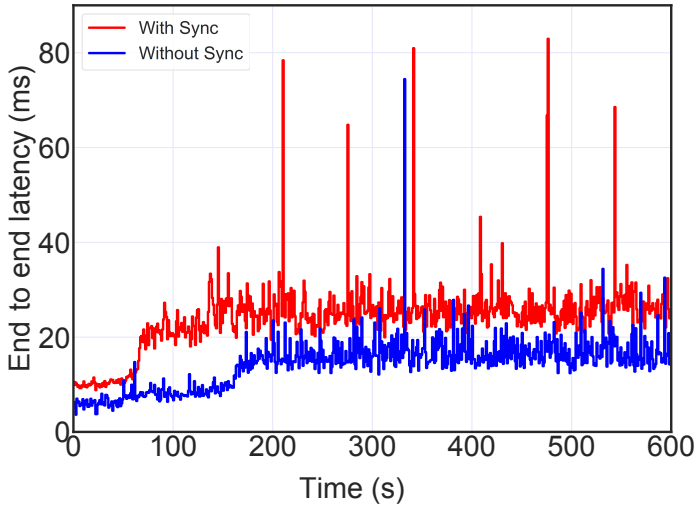


Figure 2.15: Tail latency for the Social Network application when Redis's logging is enabled (red) and disabled (blue). Sinan identified Redis as the source of unpredictable performance, and additionally determined the resources that were being saturated, pointing to the issue being in Redis's logging functionality. Disabling logging significanly improved performance, which is also reflected in that tier's importance, as far as meeting QoS is concerned, being reduced.

and identify that it is set to record logs in persistent storage every minute. For each operation, Redis forks a new process and copies all written memory to disk; during this it stops serving requests.

Disabling the log persistence eliminated most of the latency spikes, as shown by the

blue line in Figure 2.15. We further analyze feature importance in the model trained with data from the modified Social Network, and find that the importance of social-graph Redis is significantly reduced, as shown in the w/o Sync part of Table 2.5, in agreement with our observation that the service's tail latency is no longer sensitive to that tier.

2.6 Conclusion

We have presented Sinan, a scalable and QoS-aware resource manager for interactive microservices. Sinan highlights the challenges of managing complex microservices, and leverages a set of validated ML models to infer the impact allocations have on end-to-end tail latency. Sinan operates online and adjusts its decisions to account for application changes. We have evaluated Sinan both on local clusters and public clouds GCE) across different microservices, and showed that it meets QoS without sacrificing resource efficiency. Sinan highlights the importance of automated, data-driven approaches that manage the cloud's complexity in a practical way.

CHAPTER 3

URSA: LIGHTWEIGHT RESOURCE MANAGEMENT FOR CLOUD-NATIVE MICROSERVICES

3.1 Introduction

Cloud applications, such as Twitter and Netflix, are increasingly built as graphs of microservices [131, 6, 66], and deployed with cloud-native frameworks like Kubernetes [13, 4, 9, 1]. Despite the benefits of modularity and elasticity, resource management for microservices that must meet SLA constraints, e.g., end-to-end latency, is challenging, due to the diverse resource requirement of individual microservices and their inter-service dependencies [65, 66]. Resource management for microservices has been studied recently, and machine learning (ML) models, especially deep neural networks (DNN), have become a popular choice to address the complexity of microservice topologies. Previous studies have either used ML to predict important performance metrics, such as latency and load [142, 65, 51, 138, 93], or to directly adjust resource allocation [113], and demonstrate that ML-driven approaches outperform traditional techniques, such as autoscaling [19], in performance and resource efficiency.

However, ML-driven approaches still face key challenges limiting their adoption. First, ML-driven approaches typically require a lengthy exploration process to collect tens of thousands of data points to train the models. Worse, a large number of SLA violations need to be triggered during the exploration process [113, 142] to produce a balanced dataset, making it impractical to perform the exploration process online with real user requests, and thus hard to track changes in user behavior or to cope with frequent updates to the microservice logic. Second, these ML models are on the critical path for every resource management decision, limiting the speed and scalability of resource

management. Third, previous studies are evaluated using conventional benchmarks that use remote procedure call (RPC) as the only method of inter-service communication and include only lightweight text processing in the business logic [66, 126, 144], whereas a modern cloud-native application use both RPC and message queues (MQ) [92], such as Kafka [3] and Redis streams [18], and handle different user request classes performing different tasks, such as image processing and ML workloads [122, 94, 117], and support different request priorities. Different request classes or priorities exhibit different latencies and therefore have different SLAs, making resource management more challenging.

To address the challenges above, we introduce *Ursa*, a lightweight resource management framework for cloud-native microservices. As a first step, we conduct a case study to understand how latency anomalies propagate through different communication methods, due to improper resource allocation. The results show that backpressure is only significant in RPCs and is most pronounced in the parent service of the culprit (bottleneck microservice). We propose a method to determine the resource utilization threshold for microservices that prevents backpressure in a microservice system. By eliminating backpressure, the number of dependencies required to model microservice latency with N microservices is reduced from the worst case $O(N^2)$ where a service's latency may depend on resources of of its downstream in addition to its own, to $O(N)$ where each service's latency becomes only a function of its own resources. Then, in a backpressure-free system, we develop a performance model based on mixed integer programming (MIP). The model decomposes end-to-end latency SLA constraints into per-service latency constraints, and maps them to resource allocation thresholds for individual services. In addition, the model supports specifying different SLAs for different request classes and priorities. To speed up exploration while reducing SLA violations, Ursa explores as many independent services across different request paths,

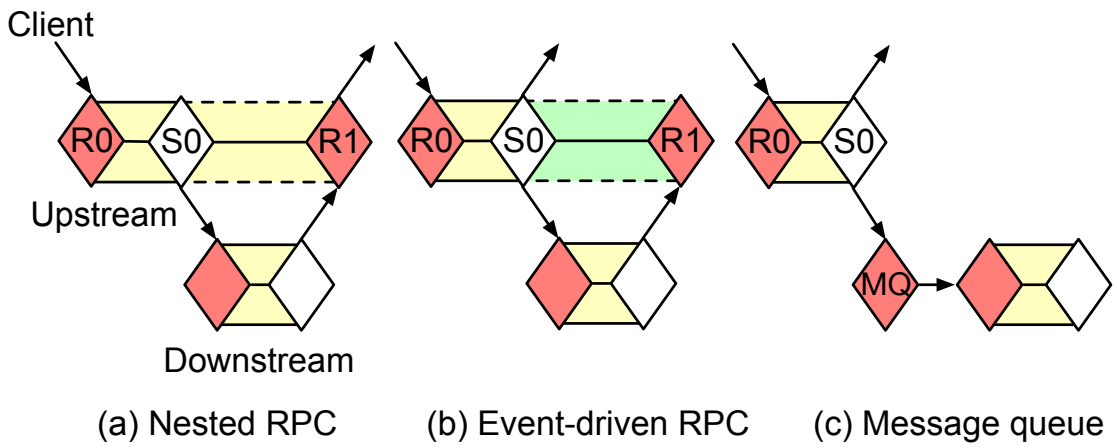


Figure 3.1: Inter-service communication methods.

and swiftly stops exploration when violations occur or the resource utilization reaches the backpressure-free thresholds.

To better reflect modern microservices, we re-implement the DeathstarBench [66] using Dapr [5], a popular microservice framework developed and used by major cloud providers. The re-implemented benchmarks use both RPCs and MQs, and implement different request classes and priorities, executing more diverse business logic than before. We compare Ursa to two representative ML-driven systems, Sinan [142] and Firm [113] as well as traditional autoscaling. Ursa reduces the required sample size by more than 16×, and the SLA violations during exploration by more than 96×, making online exploration using real user data possible. During online deployment, Ursa's control plane is 43× faster than prior work, and Ursa reduces the SLA violation rate by 9.0% to 49.9%, and the CPU allocation by up to 86.2% compared to ML-driven approaches.

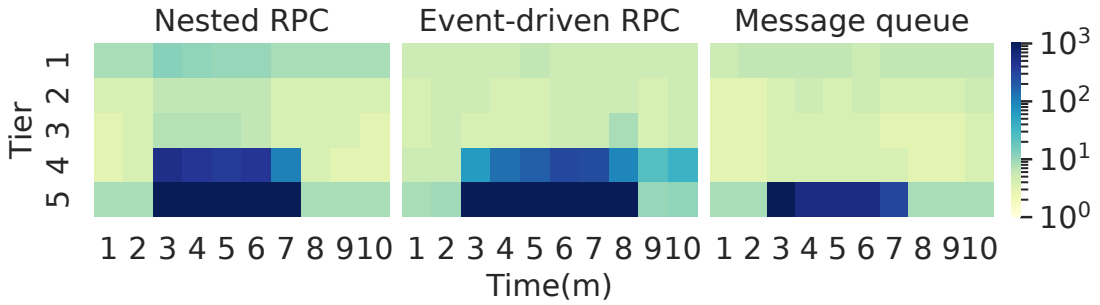


Figure 3.2: Backpressure effect in a service chain.

3.2 Backpressure Effect

Backpressure is one of the major challenges for microservice resource management, and it refers to the phenomenon that the resource allocation of one service can affect the latency of upstream services, in addition to its own. In the presence of backpressure, even modeling the latency distribution of a single service is complicated, as it can be affected by the resources of downstream services. In the presence of backpressure, modeling microservice latency requires modeling $O(N^2)$ dependencies with N services in the worst case, as each service's latency can be affected by the resources of its downstream services. Many microservice resource management frameworks use a centralized performance model that requires global information about all microservices to account for the inter-service dependencies, at the cost of scalability [142, 65, 51]. To achieve scalable resource management for microservices, we first conduct a case study to understand how backpressure propagates through different communication methods, including RPC and MQ.

We study three types of chains connected by nested RPC, event-driven RPC, and message queues, respectively. Nested RPC, as shown in Figure 3.1(a), is a synchronous system where, upon receiving the client request ($R0$), the upstream service forwards the request to the downstream service ($S0$) via RPC, blocks until the response is received

Figure 3.3: Backpressure profiling engine architecture.

($R1$), and then returns the result to the client. Event-driven RPC [140], as shown in Fig. 3.1(b), is more asynchronous in that upon receipt of a client request, the upstream service dispatches the request to another thread and returns immediately ($S0$), while the dispatched thread contacts the downstream service via RPC and waits for the response ($R1$). Message queue, on the other hand, is completely asynchronous. Unlike RPC, MQ mostly uses a publish-subscribe paradigm, where publishers publish messages to topics hosted by the MQ and subscribers consume messages by subscribing to the topics. As shown in Figure 3.1(c), the upstream service sends the client request to the MQ, and the downstream service gets new requests by polling the MQ.

Characterizing backpressure. We implement the RPC service chains with gRPC [10] and the MQ with Redis streams [18]. Each chain is configured to include 5 tiers, with each tier implementing a CPU-intensive loop as the request handler. We record the per-tier *response time* ($S0 - R0$), which is closely related to the resource allocation of the tier itself. We stress test each service chain for 10 minutes, injecting performance anomalies into the leaf tier (tier 5) by throttling its CPU limit between minutes 3 and 6. The resulting backpressure behaviors are shown in Figure 3.2, where each column on the x-axis represents a one minute interval, each row on the y-axis corresponds to a tier (tier 1 is client-facing), and the color of each cell highlights the per-tier 99^{th} response time during that minute. For both nested and event-driven RPC, significant backpressure is observed, especially for tier 4, the parent of the throttled leaf tier, and the backpressure rapidly diminishes up the call chain and becomes negligible above tier 3. In contrast, MQ shows no backpressure behavior, even on tier 4.

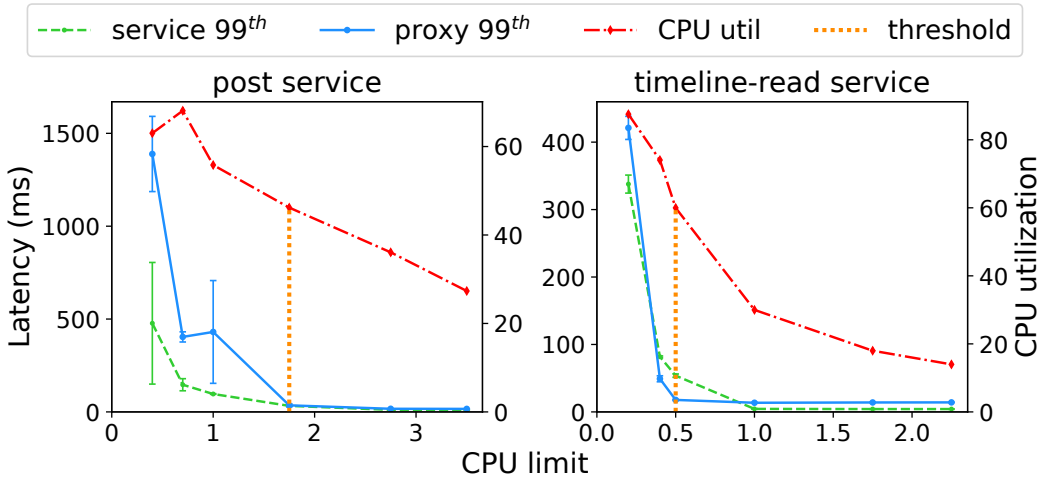


Figure 3.4: Profiling CPU threshold for no backpressure.

Determining conditions for negligible backpressure. Backpressure complicates resource management because the latency of a service also depends on resources of downstream services, in addition to its own. To simplify resource management, a natural approach is to determine safe CPU utilization thresholds to avoid backpressure in the system, as the performance of microservices is most sensitive to CPU utilization [92, 66]. To this end, we use a profiling engine with the 3-tier architecture shown in Figure 3.3, where the proxy acts as the parent service and simply forwards the request to the tested service via RPC. The engine gradually increases the CPU limit of the tested service, and monitors the latency of the proxy and the CPU utilization of the tested service, until the latency of the proxy converges. The convergence of proxy latency is determined by comparing the latency recorded under the last two CPU limits with Welch's t-test [139], a classical hypothesis testing method for identifying whether the means of the two sets of samples are equal. The CPU utilization just before the convergence of the proxy latency is then recorded as the threshold for not triggering back-pressure.

Figure 3.4 shows the profiling process for two microservices in a social network application similar to [66]: the post service and the timeline-read service. The x-axis

corresponds to the CPU limit of the tested service. The left and right y-axis corresponds to latency and CPU utilization, respectively. The blue and green lines show the average 99^{th} percentile latency of the proxy and tested service under different CPU limits, and the error bars represent the standard deviation. The red line indicates the CPU utilization of the tested service. The orange line highlights the point that proxy latency converges, and the corresponding CPU utilization is recorded as the threshold, which are 46.2% for post service and 60.0% for timeline-read service. When significant backpressure is observed, the 99^{th} percentile latencies of the proxy and the tested service have already increased by more than 5× and 10×.

Main insights. The study brings out the following insights.

1. Backpressure complicates resource management because the latency of a service depends on resources of downstream services, in addition to its own. Backpressure is common in RPC but negligible in MQ.
2. Backpressure diminishes along the invocation chain. Of all the upstream services of the culprit, the parent service shows the most significant increase in latency.
3. The backpressure-free resource utilization threshold of a service can be profiled by monitoring the latency of an upstream proxy. By operating within the thresholds, backpressure can be avoided in the microservice system.
4. By eliminating backpressure, the number of dependencies required to model service latency becomes $O(N)$ with N services, because a service's latency is a function of its own resources. Otherwise the number is $O(N^2)$ in the worst case, as a service's latency may depend on resources of all its downstream services.

3.3 Performance model

The latency distribution of a microservice becomes mainly a function of its own resources when the system has negligible backpressure. We then build a performance model for mapping SLAs to resources using two main steps: First, decomposing each end-to-end latency constraint to a set of per-service latency constraints. Second, mapping each per-service constraint to resources for that individual service. In this section we develop the performance model based on mixed-integer programming.

Decomposing end-to-end latency. Without loss of generality, we consider the end-to-end latency of a chain that handles a single type of request, which is the basic structure of microservice DAGs and to which other topologies can be transformed. For examples, a tree consists of multiple chains from the root service to leaf services, and similarly, fan-in and fan-out can be considered as multiple chains from the source service to the sink service.

Theorem 1: Consider a chain of services S_1 to S_n, and their response time distributions t_1 to t_n, where $t_i(x_i)$ is the x_i^{th} percentile latency of service S_i, $x_i \in [0, 100]$. Similarly, we define t_e to be the end-to-end latency distribution, and $t_e(x_e)$ to be the x_e^{th} percentile end-to-end latency, $x_e \in [0, 100]$. Then,

$$t_e(x_e) \leq \sum_{i=1}^{n} t_i(x_i), \text{ if } 100 - x_e \geq \sum_{i=1}^{n} 100 - x_i \tag{3.1}$$

The theorem holds true regardless of the joint distribution of service latencies (i.e., if services are independent or correlated), and it suggests that *the sum of per-service latencies provides an upper bound for the end-to-end latency at an arbitrary percentile, as long the sum of residuals of per-service percentile is no greater than the residual of the end-to-end percentile.* The proof can be found in the supplementary material.

Theorem 1 proposes a method to *verify the end-to-end latency SLA by examining latency of individual services*. For example, in a chain consisting of two services S_1 and S_2, with the SLA defined at the 99^{th} percentile latency, Theorem 1 suggests that the actual 99^{th} percentile is less than the sum of x_1^{th} percentile latency of S_1 and x_2^{th} percentile latency of S_2, as long as $100 - x_1 + 100 - x_2 \leq 1$, and in other words, (x_1, x_2) can be $(99.1, 99.9)$, $(99.5, 99.5)$, $(99.7, 99.3)$, etc. Since all such combinations of x_i are upper bounds of the actual end-to-end latency, the end-to-end SLA must be satisfied as long as the corresponding sum of the per service latency for one combination is less than the SLA target. More generally, given the end-to-end latency SLA of x_e^{th} percentile latency less than T in a chain of length n, the end-to-end SLA is satisfied if

$$\exists[x_1...x_n] \; s.t. \; \sum_{i=1}^{n} t_i(x_i) \leq T \; \& \; 100 - x_e \geq \sum_{i=1}^{n} 100 - x_i \tag{3.2}$$

Mapping per-service latency to resource. For the end of optimizing resource allocation, the per-service latency distributions t_i need to be associated with resource allocations, so that a model that maps SLA to resources can be derived. In addition, the model should be able to handle multiple classes or priorities of requests instead of one single class as in Theorem 1.

In cloud-native frameworks such as Kubernetes [13, 4, 9], dynamic resource tuning is typically achieved by the changing the number of replicas, each with a predefined resource configuration (CPU and memory). Therefore, we use *load per replica (LPR)* as the metric to relate resources to latency, where load is measured in requests per second (RPS). Considering a service S_i that handles c classes or priorities of requests (v_1 to v_c), the load per replica y_i can be represented as a vector $[a_i^1...a_i^c]$ in which a_i^c is the load for request class v_c. If the load per replica vector y_i is used as the resource allocation threshold and the total load to S_i is $[A_i^1...A_i^c]$, the resource consumed by S_i can be conceptually calculated with Equation 3.3, in which u_i is the resource consumption per replica.

$$r_i(y_i) = \max_{1 \le j \le c} \lceil \frac{A_i^j}{a_i^j} \rceil \cdot u_i \tag{3.3}$$

On the other hand, since the latency distribution of request v_j in S_i is a function of LPR y_i, the x_i^{th} percentile latency of v_j can be denoted as $t_i^j(y_i, x_i)$. Then the $t_i(x_i)$ items in Equation 3.2 can be replaced by $t_i^j(y_i, x_i)$, transforming the latency constraint to a resource allocation constraint. Albeit $t_i^j(y_i, x_i)$ can be fitted with profiling data, the resource-latency function can be an arbitrary non-increasing function that is not necessarily convex, which makes it hard to be used in convex optimization models. Instead, we can discretize the variables and use the function in MIP, which can be efficiently solved by modern optimization solvers using heuristics such as branch-and-bound algorithm [87]. Specifically, we discretize the percentile variable x_i and LPR y_i, and represent the latency distributions under different LPRs as a matrix D_i^j, where each element of D_i^j is the latency that corresponds to a certain pair of LPR and percentile. For example, assuming that S_i is profiled under m different LPRs $Y_i = [y_i^1 ... y_i^m]$ and the latency distribution is discretized into h different predefined percentiles $P = [p_1 ... p_h]$, D_i^j will be a $m \times h$ matrix where $D_i^j[\alpha, \beta]$ is the latency at percentile p_β under LPR y_i^α. As a result of the discretization, the LPR variable y_i can be represented by a one-hot vector δ_i of length m, indicating which LPR is chosen as the resource allocation threshold. Similarly, the percentile variable can be presented by one-hot vector γ_i^j of length h, indicating which percentile contributes to sum of per-service latency for request class or priority v_j. With the two one-hot decision variables, the latency of request class or priority v_j in service S_i can be expressed as $\delta_i^T D_i^j \gamma_i^j$, and the resource consumption can be expressed as $\delta_i^T R_i$, in which R_i is a 1-D vector corresponding to resource consumption under the profiled LPRs, computed with Equation 3.3.

Resource optimization model. Given that we can provide an upper bound on the

end-to-end latency using the sum of per-service latency and map per-service latency to resource allocation thresholds, we can eventually design an optimization model that relates end-to-end SLAs to resources, and calculates the most efficient resource allocation threshold. Specifically, the inputs to the model include the load of the application, SLAs for different request classes and priorities, and per-service latency distributions under different LPR thresholds. The output of the model is the most efficient per-service LPR threshold that satisfies SLAs, of all given LPR thresholds. With the described notations summarized in Table 3.1, we derive the following solvable mixed-integer program model that yields optimal resource configuration given a set of end-to-end constraints: for each request class or priority v_j, the x_j^{th} percentile latency should be less than T_j.

	Description
δ_i	Resource (LPR) one-hot vector
γ_i^j	Latency percentile one-hot vector
R_i	Resource consumption under different LPRs
P	Discretized percentile values
D_i^j	Latency distribution matrix
T_j	End-to-end SLA target value
x_j	End-to-end SLA target percentile
$\mathbb{1}$	1-D vector whose elements are all 1

Table 3.1: Notations in the MIP.

$$
\begin{aligned}
\text{minimize} \quad & \textstyle\sum_{i=1}^{n} \delta_i^T R_i, \\
\text{subject to} \quad & \textstyle\sum_i \delta_i^T D_i^j \gamma_i^j \le T_j, \forall j && (1) \\
& \textstyle\sum_i 100 - P^T \gamma_i^j \le 100 - x_j, \forall j && (2) \\
& \mathbb{1}^T \delta_i = 1, \forall i && (3) \quad (MIP\ 1) \\
& \mathbb{1}^T \gamma_i^j = 1, \forall i, j && (4) \\
\text{variables} \quad & \delta_i \ (0 \le \delta_i \le 1 \ \& \ \delta_i \in Z) \\
& \gamma_i^j \ (0 \le \gamma_i^j \le 1 \ \& \ \gamma_i^j \in Z)
\end{aligned}
$$

The objective of MIP 1 is to minimize the total resource consumption. Constraint 1 specifies that for each request class or priority, the sum of per-service latency must

be smaller than the SLA target, and constraint 2 specifies that the sum of per-service latency in the constraint 1 is an upper bound of the actual end-to-end latency. The rest of the constraints enforce the decision variables to be one-hot vectors. For each service, the LPR one-hot vector δ_i produced by MIP 1 corresponds to the most efficient resource allocation threshold among all profiled LPRs, which allows resource allocation of each service to be decided independently, by simply checking the load of the service.

Mitigating latency overestimation. The quality of the solution of MIP 1 is related to the tightness of the upper bound given by Theorem 1, as a loose upper bound well above the actual latency can lead to overprovisioning of resources. An intuitive way to tighten the upper bound is to record the ratio of the upper bound to the actual value and use that ratio to refine the SLA constraint in MIP 1. For example, if the overestimation ratio of request class or priority v_j is α_j and its expectation is $E(\alpha_j)$, constraint 1 in MIP 1 can be refined to $\sum_i \delta_i^T D_i^j \gamma_i^j \leq E(\alpha_j) T_j$. With a fixed resource allocations denoted by δ_i^*, for service $S_i, \forall i$, the upper bound on the latency of request class or priority v_j can be solved using MIP 2. The objective value is the tightest upper bound, because any percentile combination satisfying constraint 1 in MIP 2 establishes a upper bound on latency, and the objective is the smallest among all these upper bounds. Thus, the overestimation ratio α_j is the ratio of the objective and the actual latency, and $E(\alpha_j)$ is the average of α_j with different resource allocations.

$$
\begin{array}{lll}
\text{minimize} & \sum_i \delta_i^{*T} D_i^j \gamma_i^j, & \\
\text{subject to} & \sum_i 100 - P^T \gamma_i^j \leq 100 - x_j, \forall j & (1) \\
 & \mathbb{1}^T \gamma_i^j = 1, \forall i, j & (2) \\
\text{variables} & \gamma_i^j \ (0 \leq \gamma_i^j \leq 1 \ \& \ \gamma_i^j \in Z) &
\end{array}
\qquad (MIP\ 2)
$$

Discussion. In this work we use the performance model to find the most efficient resource allocation given latency SLA constraints, but the model can be extended to other

cases with minor modifications. For example, the model can handle end-to-end latency minimization under resource constraints, by replacing the objective of MIP 1 with the sum of per-service latencies, and using the total available resource as a constraint. In addition, the model can handle SLAs defined in terms of request failure rates. The failure rate of an end-to-end request is no greater than the sum of request failure rates of the services it goes through, and the service's request failure rate is related to its resources, since insufficient resources will cause requests to time out and fail. The sum of per-service failure rate can be then used as a constraint to strengthen MIP 1. The model can also support dynamic request paths by adding recorded paths to the model during deployment. In the case of a service being accessed multiple times in a dynamic path, the model can be simplified by considering the total time spent in each service. We plan to investigate these potential use cases in future work.

3.4 Proof of Theorem 1

We first introduce a lemma and use it to prove the theorem.

Lemma 1. Given random variables X, Y and Z, where $X = Y + Z$.

$$P\{X \geq y + z\} \leq P\{Y \geq y\} + P\{Z \geq z\}$$

Proof:

$$\{X \geq y + z\} \wedge \{Y < y\} \wedge \{Z < z\} = \emptyset$$

$$\{X \geq y + z\} \subset \neg\{\{Y < y\} \wedge \{Z < z\}\} = \{Y \geq y\} \vee \{Z \geq z\}$$

$$P\{X \geq y + z\} \leq P\{Y \geq y\} + P\{Z \geq z\}$$

In the last step we leverage union bound. With *Lemma 1*, we prove Theorem 1, and for ease of explanation, we rephrase it as follows .

Theorem 1. Consider a chain of services $S_1, ..., S_k$, and their latency $t_1, ..., t_k$, where $t_i(x)$ is the latency of the $(100x)^{th}$ percentile, i.e., $P(t_i \leq t_i(x)) = x$, and t_e is the end-to-end latency, with $t_e = \sum_{i=1}^{k} t_i$ for each request. Assuming that end-to-end latency constraint is defined at x_e ($x_e \in [0, 1]$), and decomposed per-service latency constraints are defined at x_i ($x_i \in [0, 1]$). Then,

$$t_e(x) \leq \sum_{i=1}^{k} t_i(x_i), \text{ if } 1 - x_e \geq \sum_{i=1}^{k} 1 - x_i$$

Proof: By iteratively applying *Lemma 1* to the first service and the rest of the services combined, we obtain

$$P\{t_e \geq \sum_{i=1}^{k} t_i(x_i)\} \leq \sum_{i=1}^{k} P\{t_i \geq t_i(x_i)\} = \sum_{i=1}^{k} 1 - x_i \leq 1 - x_e$$

$$x_e \leq 1 - P\{t_e \geq \sum_{i=1}^{k} t_i(x_i)\}$$

$$t_e(x_e) \leq t_e(1 - P\{t_e \geq \sum_{i=1}^{k} t_i(x_i)\}) = \sum_{i=1}^{k} t_i(x_i)$$

In the last step we leverage the fact that $t_e(x_e)$ is non-decreasing.

3.5 Allocation Space Exploration

The task of allocation space exploration is to collect input data for the MIP model, including the potential resource allocation thresholds and the corresponding latency distribution for each service. Exploration should cover the most efficient resource allocation thresholds while triggering the minimum possible SLA violations. In a service chain,

these requirements can be satisfied by exploring one service at each time. All other services are provisioned with sufficient resources, so that the latency slack between SLA target and normal latency is assigned only to the profiled service to determine its minimum feasible resource allocation. The LPR threshold profiling algorithm is shown in Algorithm 1, in which we gradually reduce the replicas of the profiled service to increase the load on each replica and record the corresponding latency distributions. Profiling is terminated when the frequency of SLA violations exceeds a user-defined threshold. Additionally, profiling is also terminated when the CPU utilization of the service exceeds the service's backpressure-free threshold to preserve the independence assumptions of the performance model. Then we restore sufficient resources for the profiled service, and continue to profile the next service.

Algorithm 1: LPR threshold profiling algorithm.

Input: Initial replica R, SLA violation threshold F_{sla}, backpressure-free threshold CPU_{bp}, profiling time T;
Output: Mapping from *LPR* to latency distributions;
Variable: Replica r, replica tuning step $step$, Load L, SLA violation frequency f_{sla}, CPU utilization cpu, latency distribution d_{lat};
Initialize $r \leftarrow R$, $map \leftarrow \{\}$;
while $r > 0$ **do**
    $wait(T)$;
    if $cpu \geq CPU_{bp}$ —— $f_{sla} \geq F_{sla}$ **then**
        $r = R$ and **terminate**;
    else
        $map[\frac{L}{r}] = d_{lat}$, $r = r - step$;
    end
end
return map

In a DAG topology with multiple end-to-end request paths, the most straightforward way for exploration is to profile one service at a time, but such exploration can take a long time. Instead, we can speed up the exploration by profiling services in different paths in parallel. Given a DAG topology and specifications of end-to-end request paths, we use the graph coloring algorithm in Algorithm 2 to identify groups of services that

can be profiled in parallel. The algorithm produces a mapping from the round ID to the group of services that can be profiled simultaneously in that round. The algorithm applies depth-first search (DFS) to the DAG to ensure that an upstream service is assigned an ID greater than that of its downstream services to be profiled later. In addition, the algorithm ensures that only one service is profiled at a time in each request path, by keeping track of the maximum ID in each path and assigning each service an ID greater than any path through the service. The exploration is then performed in rounds, where in each round all services of that round are profiled in parallel. Only after all services in the previous round have been profiled does the exploration move to the next round.

Algorithm 2: Graph coloring algorithm.

```
Input: entry service;
Output: Map from ID to the group of services;
Variable: Service s, path p, map from path to ID C, all paths through the
 service s P_s;
Function DFS(s):
    if s.id == 0 then
        for c ∈ s.childs do
            s.id = max(s.id, DFS(c));
        end
        s.id = max(C[p] ∀p ∈ P_s, s.id) + 1;
        map[s.id].insert(s);
        for p ∈ P_s do
            C[p] = max(C[p], s.id)
        end
    return s.id;
Initialize s.id ← 0, ∀s, C[p] ← 0, ∀p, map ← {};
DFS(entry);
return map
```

There are also situations where only a fraction of the services need to be explored, for example when only a few services have undergone business logic updates or load changes. In this case, we use the same order generated by Algorithm 2 and skip the unaffected services during exploration.

3.6 Design and Implementation

We now present the design and implementation of ***Ursa***, a resource management framework based on the proposed performance model and allocation space exploration mechanism. Ursa is built on top of Kubernetes [13], a popular container orchestration framework adopted by major cloud providers [2, 4, 9, 1], and leverages Kubernetes's APIs to dynamically allocate resources by tuning the number of replicas. Ursa requires the user to provide the topology and the end-to-end SLAs of the microservice application, including request paths, percentiles, and target latencies.

Ursa aims to make resource management decisions fast and scalable. Instead of using ML models to make every resource management decision, Ursa simplifies resource management decisions to threshold-based scaling by implementing the performance model from Section 3.3. In addition, Ursa reduces exploration overhead through the exploration process in Section 3.5. Ursa is implemented in Python with around 10K lines of code. The components of Ursa are shown in Figure 3.5, and the functionality of each component is described below.

1. The **tracing framework** is implemented with Prometheus [16], a time-series database for metrics monitoring. It collects the CPU and memory usage data, as well as the request counts and latency distributions of each service.

2. The **exploration controller** implements the allocation space exploration mechanism. It first determines the backpressure free CPU utilization thresholds for RPC-connected services (Section 3.2). Then, it explores feasible resource allocation thresholds (Section 3.5), by generating the order of service profiling using Algorithm 2, and profiling each service using Algorithm 1.

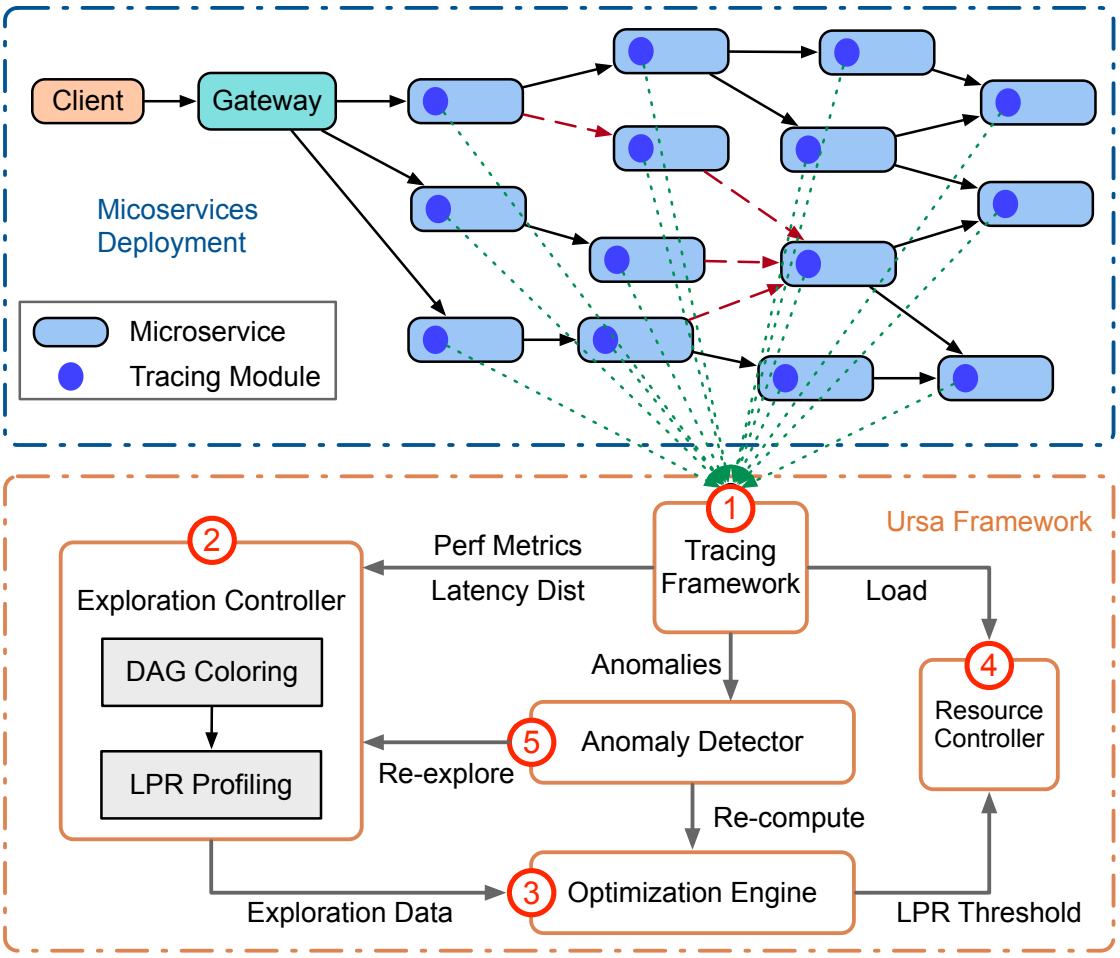


Figure 3.5: System Architecture of Ursa.

3. The **optimization engine** then determines the resource allocation threshold for each service using the performance model in Section 3.3, using exploration data and user load information collected by the tracing framework. The optimization engine is implemented with Gurobi [11]. During deployment, the optimization model is only occasionally computed when the business logic of services is updated or the mix of requests changes significantly, otherwise the results can be reused.

4. Using the load per replica threshold calculated by the optimization engine, the **resource controller** dynamically adjusts the number of replicas as the load changes, ensuring that for any class or priority of request, the average load on each replica does

not exceed the threshold. Specifically, the resource controller determines whether the average load in one replica exceeds the threshold using Welch's t-test [139] to accommodate the noise of load fluctuation. Since the calculation only requires the total load per service as input, resource control can easily scale to complex topologies consisting of many services, by having each service managed by a separate resource controller.

5. During deployment, the **anomaly detector** periodically checks for anomalies in load and latency, and triggers recalculation of resource allocation thresholds or re-exploration, if necessary. Load anomalies refer to drastic changes in the ratio of different classes or priorities of requests that may lead to resource over-provisioning, in which case resource allocation thresholds are recalculated to improve resource efficiency. The anomaly detector identifies changes in request ratios by monitoring the *request ratio deviation* of each service, which measures the difference between load of the service and the load per replica threshold for scheduling. The metric is denoted by $\max_i \frac{l_i}{t_i} \frac{\sum_i t_i}{\sum_i l_i}$, where l_i and t_i are the total load and per-replica load threshold for the i^{th} request class or priority. When the request ratio deviation exceeds a user-defined threshold, the anomaly detector asks the optimization engine to recalculate the thresholds and update the resource controllers. If the re-calculated thresholds still fail to mitigate the request ratio deviation, indicating that the load pattern is not covered by previous exploration, the anomaly detector asks the exploration controller to re-explore the affected service.

On the other hand, latency anomalies refer to SLA violations, which indicate that the latency distribution recorded during exploration needs to be updated. Similar to load anomalies, users can specify an end-to-end SLA violation threshold that triggers the re-exploration process if the SLA violation exceeds the threshold during deployment.

3.7 Benchmarks

Conventional microservice benchmarks [66, 126, 144] have several limitations. First, conventional benchmarks use RPCs as the only method for inter-service communication, whereas MQs are increasingly becoming more common in practice [92]. Second, the business logic of conventional benchmarks involves only lightweight text processing, whereas a modern microservice handles different user request classes performing tasks, such as image processing and ML workloads [122, 94, 117], and even different request priorities, making resource management more challenging. To address these limitations, we implement three benchmark applications using Dapr [5], a popular microservice framework developed and used by major cloud providers, as described below. For all the applications, we implement the business logic in Golang and Python, and use gRPC [10] for RPCs, Redis streams [18] for message queues and Redis [17] for data stores.

Social network. The social network application is a re-implementation of the DeathStarBench [66] application. In addition to original features including uploading text posts and reading timelines, the re-implemented version includes several new features, including uploading images, sentiment analysis of texts, and object detection of images. Sentiment analysis and object detection are implemented with machine learning models from Hugging Face [12], and are connected to other services via MQs.

Media service. The media service is also a re-implementation of the DeathStarBench application. In addition to the original features including reviewing and rating videos, the re-implemented version additionally allows users to upload and download actual videos, and includes video-processing tasks, such as transcoding to different resolutions and generating thumbnails via FFmpeg [8]. The video transcoding and thumbnail services are connected to other services via MQs.

Video processing pipeline. Video processing pipeline consists of three stages: The first stage extracts video metadata, the second stage takes snapshots from the video at fixed intervals, and the third stage performs face recognition on the video snapshots. The first two stages use FFmpeg, the third stage uses OpenCV [15], and stages are connected with MQs. The application handles two request priorities. High-priority requests are always processed immediately when worker threads are available, while low-priority requests are processed only when there is no high-priority request waiting in the queue.

Previous work typically handles a single SLA and only manages synchronous requests. For example, Sinan [142] handles a single SLA of 500ms for the 99^{th} percentile latency of upload-post, read-timeline, and update-timeline in social network. However, different request classes have diverse latencies. For example, in social network, it takes tens of milliseconds to upload a post, hundreds of milliseconds to update timelines, and a few seconds to perform object detection. To reflect these latency ranges, we assign an SLA per each request class and priority. We stress test the applications with high user loads and use the latency before saturation as the SLA. The SLAs of social network, media service, and video processing pipeline are listed in Table 3.2, 3.3, 3.4, respectively. The SLAs are mostly defined as the 99^{th} percentile, except for the low-priority requests in the video processing pipeline, which is defined as the 50^{th} percentile latency.

Request type	99^{th} latency (ms)
upload-post/comment	75
read-timeline	250
update-timeline	500
upload-image	200
download-image	75
sentiment-analysis	500
object-detect	10000

Table 3.2: SLAs of the social network.

Request type	99^{th} latency (ms)
upload-video	2000
download-video	1500
get-info	250
rate-video	400
transcode-video	40000
generate-thumbnail	2000

Table 3.3: SLAs of the media service.

Request type	Percentile	Latency (ms)
high-priority	99^{th}	20000
low-priority	50^{th}	4000

Table 3.4: SLAs of the video processing pipeline.

3.8 Evaluation

We aim to answer the following questions:

1. What is the overhead of Ursa's online exploration process? (Section 3.8.3)
2. How accurate is Ursa's performance model in capturing end-to-end latency (Section 3.8.4)?
3. How effective is Ursa in reducing resource usage and maintaining SLAs? (Section 3.8.5)
4. What is the latency required for Ursa to make resource allocation decisions? (Section 3.8.6)
5. Is Ursa able to adapt to business logic changes of microservices? (Section 3.8.7)

3.8.1 Experimental Setup

We use the benchmarks in Section 3.7 and use Locust [14] to generate input load following a Poisson arrival process. The applications are deployed on a local Kubernetes cluster consisting of 8 machines with 40-88 CPUs and 126-188 GB of memory each, with a NIC bandwidth of 10 Gbps. To reduce interference between containers colocated on the same server, we set the CPU management policy of Kubernetes to the static policy [85], which allows each container to access exclusive CPUs, as long as it is configured with an integer number of CPUs. The CPU configuration of each microservice's container is determined by monitoring the CPU usage of the container at low RPS and rounding it to the nearest integer, and similarly, the memory configuration is set to the maximum profiled memory usage to avoid OOM errors. During online deployment, we adjust the resource allocation for each microservice by changing the number of replicas.

3.8.2 Competing Approaches

We compare Ursa to the following systems.

Sinan. Sinan [142] is a model-based ML-driven resource management framework for microservices. It uses a CNN and boosted trees model, to predict the end-to-end latency of a microservice topology given a certain resource allocation for all services, and we modify the models to adapt to the topologies of our applications. Sinan is implemented as a centralized scheduler that periodically queries the model with different resource allocations, and chooses the one using the least amount of resources, while meeting the SLA. The training data of the models are collected with a process designed to explore unseen resource allocations and keep the ratio of violating to meeting SLAs at $1:1$, so that the trained models are not biased towards either predicting SLA violation or SLA

satisfaction.

Firm. Firm [113] is a model-free, ML-driven framework for microservice resource management. Unlike Sinan that trains models to predict latency, Firm assigns a reinforcement learning agent to each service that directly adjusts the resource allocation for the service, given its resource usage and end-to-end SLA status. The reward for each agent is designed to be the weighted sum of the reduced resource usage and the SLA violation status after applying the resource allocation decision. The agents are trained by injecting performance anomalies during online deployment.

Autoscaling. Autoscaling [19] is a widely adopted resource management method. The autoscaling controller relies on manually configured resource utilization thresholds based on expert knowledge to dynamically adjust resource allocation. In our experiments, we configure the autoscaling controller according to [19], which increases resources when CPU utilization exceeds 60%, and reduces resources when the CPU utilization is below 30%.

3.8.3 Online Exploration Overhead

We now compare the online exploration overheads of Ursa, Sinan and Firm. During exploration, the user load for each application is the same across the three approaches. Specifically, for the social network application, the RPS averages 1000 and the ratios of post, comment, download-image and read-timeline are approximately 1:75:15:25, adopted from [142, 86, 69]. For the media service application, the RPS averages 300 and the ratios of upload-video, get-info, download-video, and rate-video are approximately 1:100:25:25. For the video processing pipeline, we experiment with four different ratios of high and low priority requests, including 5:95, 25:75, 50:50, and 75:25, with an

average RPS of 10. Across all approaches, the sampling frequency is set to once per minute and the number of SLA violations is the number of samples with at least one request class or request priority violating the SLA.

We run Ursa's online exploration process (as described in Section 3.5). The average number of services that can be profiled in parallel (essentially the speedup compared to profiling one service at a time), as determined by the graph coloring algorithm (Algorithm 2), is 2.3, 2.3, and 1.0 for social network, media service, and video processing pipeline, respectively. The initial replica of each service to be profiled is determined by an autoscaling controller that keeps CPU utilization below 10% to reduce SLA violations. Each service is profiled using Algorithm 1, and in each iteration we reduce the number of replicas by 1 and collect 15 samples, until the frequency of SLA violations exceeds 10%, or the CPU utilization exceeds the backpressure-free threshold. For Sinan and Firm, we run their data collection algorithm and online training process separately and collect 10k samples for each application, matching the order of magnitude in Sinan for DeathStarBench [142].

Table 3.5 summarizes the number of samples collected and the SLA violations during online exploration. Compared to the ML-driven approaches, Ursa reduces the required sample size and exploration time by a factor of 16.7 up to 35.1, and the number of SLA violations by a factor of 96.8 up to 431.8. Ursa's SLA violation rates during exploration range from 4.2% to 9.0%, while the ML-driven approaches result in SLA violation rates of 38.4% to 61.6%.

The online exploration overheads of ML-driven approaches result from the nature of deep neural networks, which require a large amount of data to generalize, due to their large parameter space. The collected training data also needs to include a large number of SLA violations, which is crucial for ML models to learn which resource allocations

violate SLA. Otherwise, a classifier predicting whether an SLA is violated can naively predict that the SLA is met in a training dataset dominated by SLA satisfaction and achieve high training accuracy, while failing to identify SLA violations during actual deployment. Sinan [142] also demonstrates that an imbalanced dataset, i.e., dominated by meeting SLAs or violating SLAs, will lead to a model that constantly underestimates or overestimates latency, and thus fails to manage resources correctly. In contrast, Ursa's analytical model inherently contains fewer parameters than deep neural networks. The model calculates end-to-end latency as the sum of per-service latencies, and foresees end-to-end SLA violations when the latency of individual services increases rapidly. As a result, Ursa's exploration algorithm just needs to trigger SLA violations occasionally to find the the most efficient resource allocation thresholds. Notably, despite the small sample size and the low SLA violation rate during exploration, Ursa still maintains SLA and achieves high efficiency during deployment, as shown in Section 3.8.5.

App	System	Samples	Time(h)	#Viol	Viol Rate
Social	Ursa	360	6.0	18	5.0%
	Sinan	10000	166.7	4551	45.5%
	Firm	10000	166.7	3843	38.4%
Media	Ursa	285	4.8	12	4.2%
	Sinan	10000	166.7	5181	51.8%
	Firm	10000	166.7	4237	42.4%
Video	Ursa	600	10.0	54	9.0%
	Sinan	10000	166.7	6156	61.6%
	Firm	10000	166.7	4226	42.3%

Table 3.5: Online exploration overheads.

3.8.4 Model Accuracy

As described in Section 3.3, the performance model estimates the end-to-end latency by multiplying the latency upper bound with the expected overestimation rate. To evaluate

the accuracy of the estimated latency, we record the per-service and end-to-end latency distributions every 5 minutes for a total of 150 minutes during online exploration with dynamically changing resource allocations, and calculate the estimated latency for each type of request.

Figure 3.6 shows the measured and estimated latency of four representative request types in the social network application, including post, update-timeline, object-detection, and sentiment-analysis. The blue line indicates the measured 99^{th} percentile latency and the red line indicates the estimated 99^{th} percentile latency. For each class of request, the estimated latency closely follows the measured latency, with the average ratio of estimated to measured latency ranging from 0.97 to 1.05. Additionally, Figure 3.7 shows the measured and estimated latency of the video processing pipeline which includes two request priorities, with SLAs defined at the 50^{th} and 99^{th} percentiles, for low and high priority requests, respectively. For both request priorities, the estimated latency is close to the measured latency, with the average ratio of estimated to measured latency being 0.96 and 1.00 for low and high priority requests, respectively.

3.8.5 Performance Comparison

We now compare resource usage and SLA violations during deployment between Ursa and prior work, with Ursa and ML-driven systems using exploration data from Section 3.8.3. In addition to the three applications described in Section 3.7, we also show the results for the vanilla social network, by disabling the newly added ML services.

For each application, we experiment with three types of user loads; *constant load*, *dynamic load*, and *skewed load*. *Constant load* refers to Poisson arrival processes with constant RPS, with RPS of 250 to 1000, 100 to 350, and 5 to 10 for Social network,

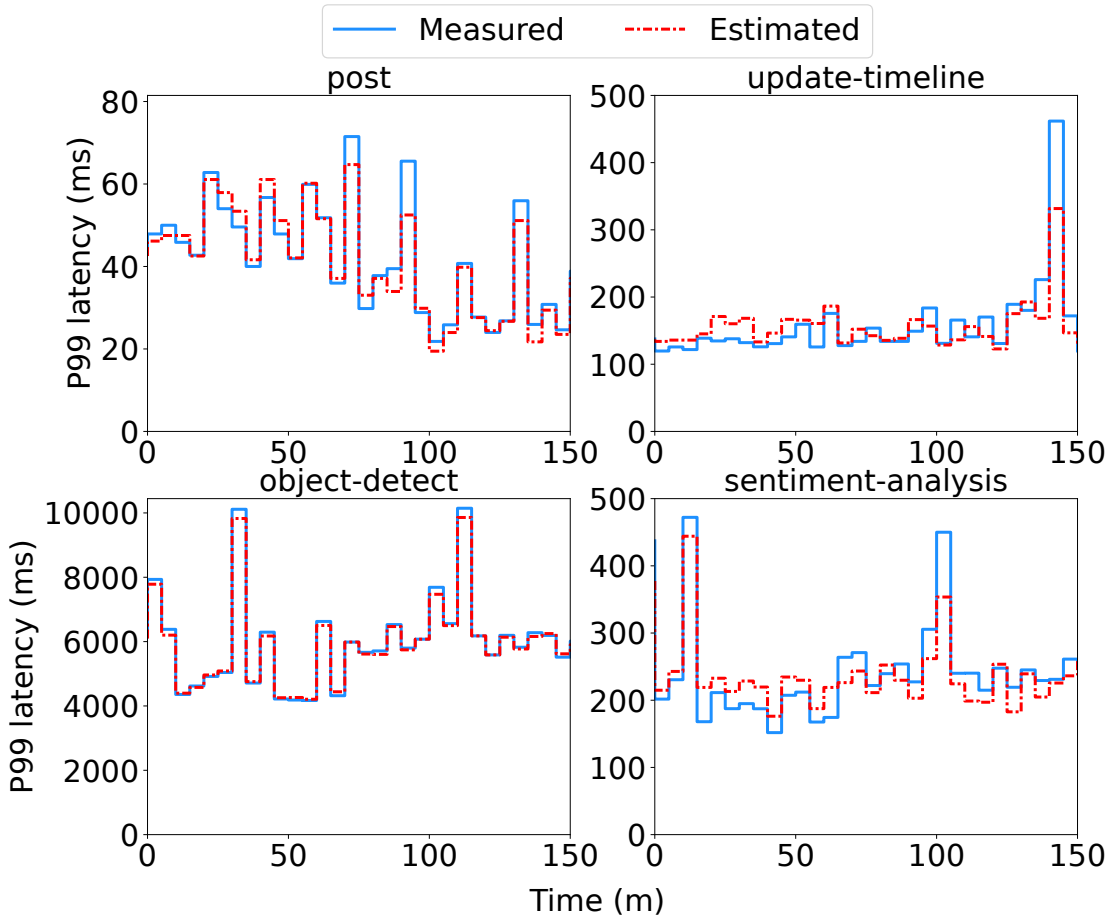


Figure 3.6: Estimated vs. measured latency for social network.

Media service, and Video processing pipeline, respectively. In contrast, *dynamic load* has time-varying RPS, including diurnal patterns where the RPS first gradually increases and then gradually decreases, and burst patterns where the RPS increases sharply by 50% to 125%. The ratio of different types of requests for constant and dynamic loads is the same as in online exploration. On the other hand, in *skewed load*, the ratio of request types differs from that in the online exploration. For social network and media service, we experiment with two other request combinations, the first doubling the frequency of update requests, including post and comment for social network, and upload-video and rate-video for media service, and the second halving the frequency of update requests. The user load patterns are the same for both versions of social network. For the video

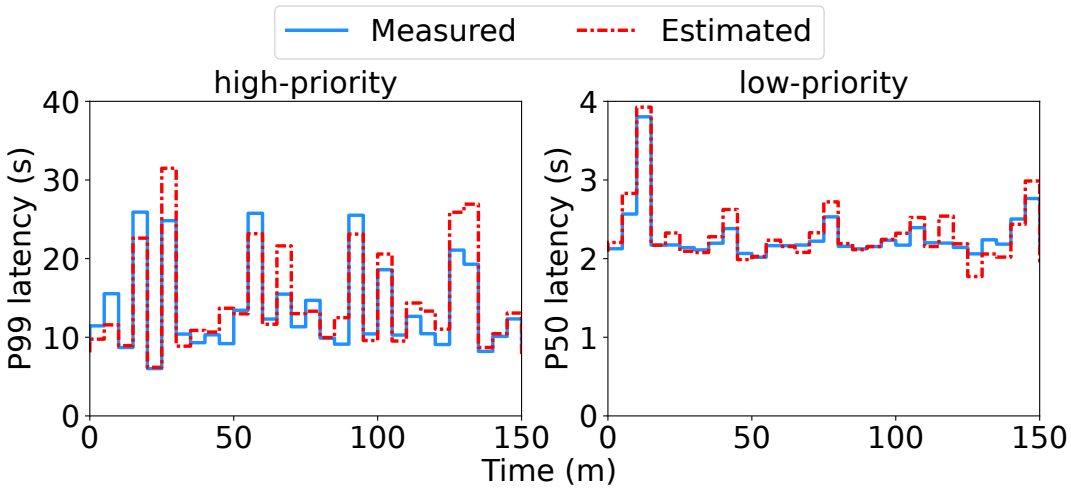


Figure 3.7: Estimated vs. measured latency for the video processing pipeline.

processing pipeline, the ratios of high-priority to low-priority requests include 40:60 and 60:40, which do not exist in online exploration. For Ursa specifically, the skewed load stresses the case where the request mix changes, and the optimization engine needs to calculate load per replica thresholds using available exploration data that do not include the current request mix. For each type of load, Ursa calculates the optimal load-per-replica thresholds once, at the beginning of the experiment.

Figure 3.8 shows the SLA violation rate, and Figure 3.9 shows the average CPU allocation. Compared to ML-driven systems, Ursa significantly reduces SLA violation rates, achieving 0.1% to 8.5% SLA violation rates under constant and dynamic loads, and 0.5% to 2.0% SLA violation rates under skewed load, whereas ML-driven systems incur 9.1% to 29.2% SLA violation rates under constant and dynamic loads, and 14.2% to 51.9% SLA violation rates under skewed load. ML-driven systems cause higher SLA violation rates for the new social network than for vanilla social network, because the latency of ML services is less stable and more challenging for resource management, compared to lightweight text processing. In terms of resources, Ursa reduces CPU allocation by 2.3% to 86.2% for constant and dynamic loads. For skewed loads, Ursa uses

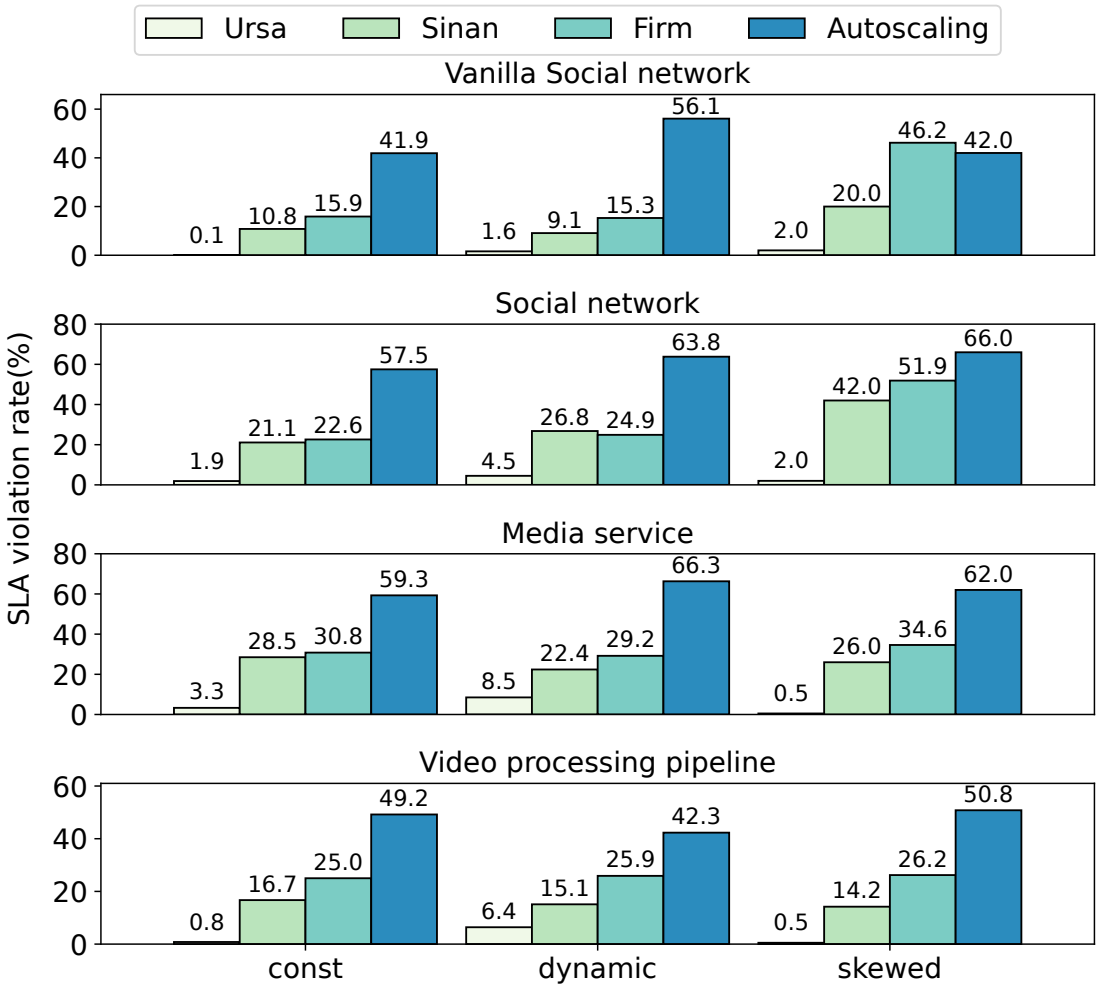


Figure 3.8: SLA violation rate.

an average of 8.2% more CPUs, but the ML-driven systems result in SLA violation rates significantly higher than Ursa. The autoscaling controller uses the least resources but results in SLA violation rates of over 60%. Ursa may use more resources under skewed loads because it prioritizes maintaining SLAs and makes conservative decisions with the available exploration data. As a conceptual example, assume a service handles two classes of requests, and its total load is $(4, 6)$, where each element of the vector is the load of one class of requests. If the service's exploration data only includes one feasible LPR threshold $(3, 2)$, Ursa will provision 3 replicas for the service to ensure that the load of any request class is below the threshold, while in reality the actual per-replica

load will be $(1.3, 2)$, which is below the $(3, 2)$ threshold. Figure 3.10 shows the load and CPU allocation for four representative services in the social network under a diurnal load when managed with Ursa, where the left Y-axis represents the RPS of load and the right Y-axis represents the CPU allocation. For each service, Ursa is able to scale out and scale in promptly as the load increases and decreases.

Ursa outperforms ML-driven systems with much lower exploration overheads, because Ursa's analytical model accurately decomposes the end-to-end latency to per-service latencies, which can be mapped directly to per-service resource allocation. However, the ML-driven techniques need to learn the relation between resource allocation and SLA from scratch in a much larger parameter space, requiring more data and leading to lower accuracy. Specifically, Sinan's SLA violation predictor can only achieve 80% to 85% accuracy due to the presence of multiple request classes with different SLAs in an application, resulting in more SLA violations and higher resource usage. On the other hand, in addition to the issue of large parameter space, Firm does not always prioritize preserving SLAs because its agent's reward function is a weighted sum of the SLA violation rate and the resource utilization, which makes Firm prioritize resource savings over SLA if the savings are significant, and results in more violations.

3.8.6 Control Plane Latency

The latency of resource allocation decisions determines how quickly the system adapts to load fluctuations, and also determines the scheduling throughput, i.e., the number of services that can be managed, according to Little's Law. Resource allocation decisions are fast in Ursa because the critical path only includes the resource controller, which calculates the number of replicas, based on the load-per-replica thresholds. In contrast,

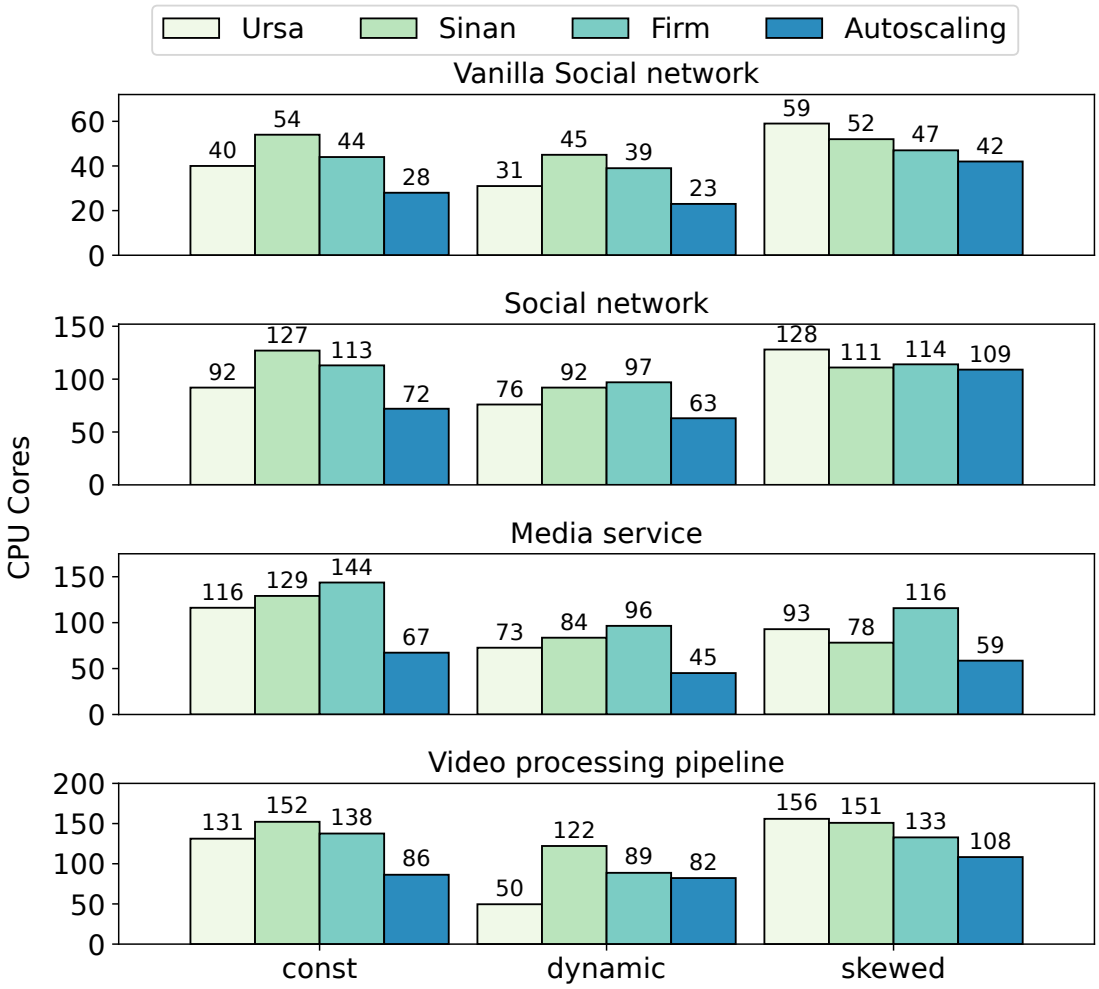


Figure 3.9: Average CPU allocation.

ML-driven systems typically need to wait for the ML models to predict performance metrics or make decisions. There are also situations where the mix of user requests changes significantly, or the business logic of a service is updated, requiring an updated performance model. In this case, Ursa needs to recompute the optimization models, and ML-driven approaches also need to update the model parameters. Table 3.6 shows the average control plane latency (in milliseconds) across the different approaches, in the case of deployment and model update. In the comparison, the control planes are always allocated 4 CPUs. Autoscaling is undoubtedly the fastest, as it involves only a single threshold check. In terms of deployment, Ursa is on average 691.6× faster than Sinan

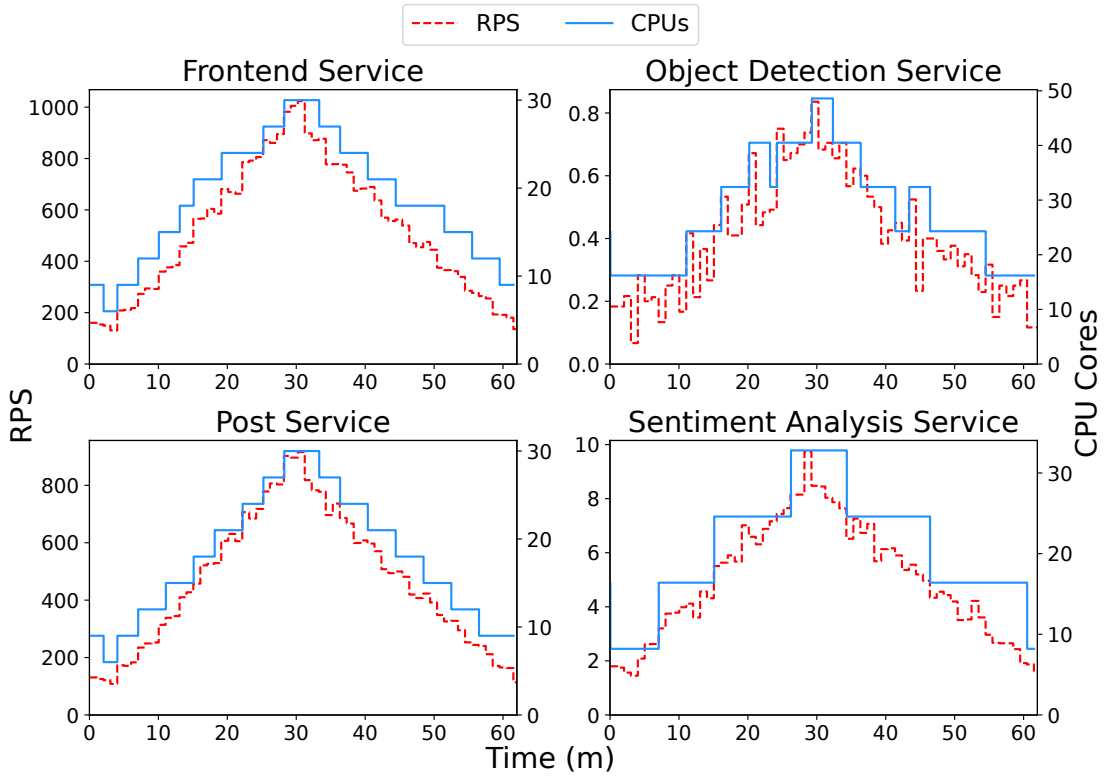


Figure 3.10: Ursa's CPU allocation under diurnal load.

using a centralized ML model, and 43.4× faster than Firm, which uses per-service RL agents. In terms of model updates, Sinan's retraining time is linear to the size of the dataset, and takes minutes even on a dedicated GPU. Firm can adapt to load changes gradually by updating the RL agent online, but is still slower than Ursa by 4.4× even for a single iteration. The RL agent may require thousands of iterations to update its weights and fully learn new resource usage patterns, whereas Ursa only needs to solve the optimization problem once to fully adapt to the changes.

	Ursa	Sinan	Firm	Autoscaling
Deploy	0.5	345.8	21.7	0.1
Update	271.7	N/A	1.2×10^3	0.1

Table 3.6: Control plane latency (ms).

3.8.7 Adapting to Service Changes

The logic of microservices can be updated frequently. We now conduct a case study to demonstrate Ursa's ability to adapt to such business logic updates. Specifically, we modify the object-detection service in the social network application, and change the model to more lightweight Mobilenet [73]. The exploration controller performs a partial online exploration to profile only the modified object-detect service. It collects a total of 75 samples in 1.25 hours, during which 4 SLA violation are triggered, resulting in an SLA violation rate of 5.3%. Then the optimization engine recalculates the LPR threshold of each service. We deploy the modified social network application under various RPS, and Figure 3.11 shows the distribution of the 99^{th} percentile latency of the end-to-end object-detect requests for the original and the updated object-detect service. The red line represents the SLA and the blue line represents the cumulative distribution function, with SLA violation rates of 0.62% and 0.50% for the original and updated microservice, respectively.

3.8.8 Summary

Compared to ML-driven approaches, Ursa reduces the required sample size by 16× to 35× during exploration, and achieves SLA violation rates of no more than 9.0%, making online exploration with real user data possible. Ursa also achieves good performance, maintaining low SLA violation rates of 0.1% to 8.5% during deployment, 9.0% to 49.9% lower than ML-driven approaches, and reducing resource allocation by up to 86.2%. In addition, Ursa's control plane is 43× faster than ML-driven approaches during deployment, enabling faster and more scalable management decisions. Finally, we demonstrate that Ursa is able to adapt to service changes and maintain SLAs while

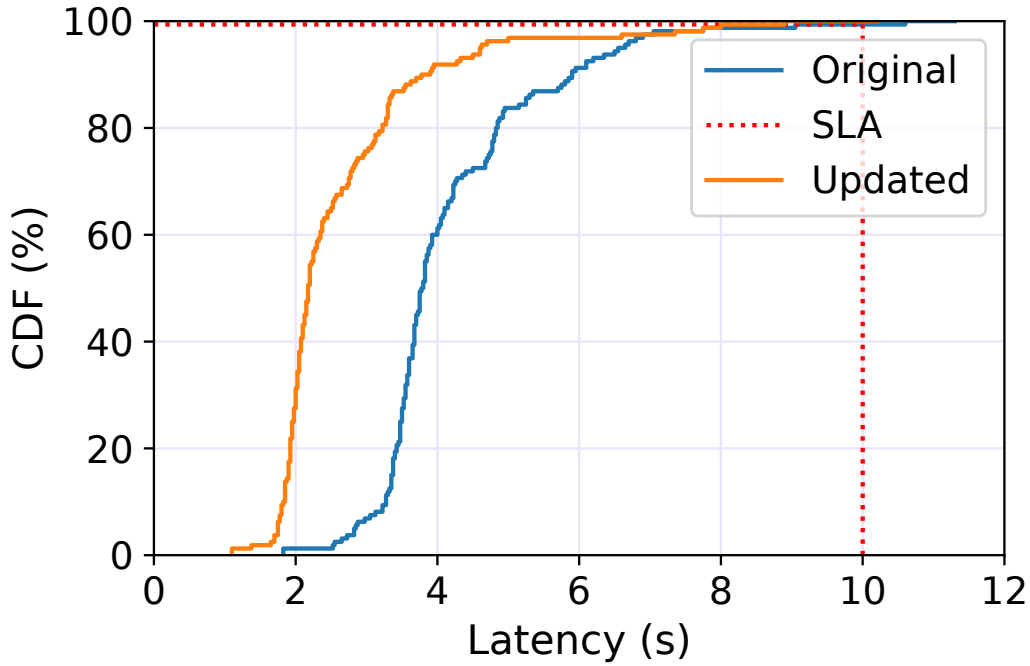


Figure 3.11: 99^{th} latency distribution of object-detect.

incurring low exploration overheads.

3.9 Conclusion

We present Ursa, a lightweight resource management framework for microservices. Ursa uses an analytical model to decompose the end-to-end SLA into per-service SLAs, and maps them to resource allocations. During exploration, Ursa explores as many independent services as possible across different request paths, and swiftly stops exploration in the case of SLA violations to keep SLA violation rate low. Using benchmarks implemented with popular microservice frameworks, we demonstrate that Ursa outperforms ML-driven approaches in both SLA maintenance and resource efficiency, with significantly lower exploration overheads.

In brief summary of Chapter 2 and 3, we aim to tackle the problem of allocating the minimal resources while preserving the SLAs, and we investigate the use of ML and analytical models to achieve the goal. While our proposed approaches are primarily geared towards microservice workloads, they can also be extended to serverless workflows, which typically have simpler DAG topology and resource usage patterns. Our study leads to two key findings.

First, although ML, particularly DNNs, is effective in solving blackbox problems, its use for resource management and control can come at a significant cost. This is particularly true if exploration is time-consuming or performance disruptive, resulting in substantial exploration overhead. In such cases, ML necessitates the use of cheap sampling techniques, such as accurate simulation, which is commonly utilized in robotics or autonomous driving, to rapidly obtain large amounts of training data at little cost. Alternatively, ML can be a good fit when datasets are already available, such as in predicting user loads.

Second, analytical models can be effective in resource management. This is primarily because computer systems are inherently less complex than natural problems, such as computer vision or natural language processing, where DNNs are currently the only viable solution. Analytical models usually have a much smaller parameter space, enabling them to achieve high accuracy with a small dataset, and they are also more robust to overfitting or distribution shift. Furthermore, designers have more control over explainable analytical models than blackbox ML models. For instance, Ursa is designed to prioritize SLA maintenance under skewed loads, which is challenging to encode in DNNs.

CHAPTER 4

FASTER AND CHEAPER SERVERLESS COMPUTING ON HARVESTED RESOURCES

4.1 Introduction

Orthogonal to the preceding two chapters which seek to reduce operational expenses by providing the minimum amount of resources necessary without impacting performance, this chapter takes a different approach to cost reduction by utilizing less expensive but less dependable resources. Our focus in this chapter is on serverless workloads, which are the most suitable for leveraging harvested resources to their full potential.

Serverless computing is becoming an increasingly popular cloud programming paradigm, especially in the form of Functions as a Service (FaaS), with offerings from several commercial providers [99, 35, 71]. These FaaS platforms offer intuitive event-based interfaces for application development. The interface obviates the need for users to explicitly configure resources, such as the number and size of virtual machines (VMs) or containers to run the functions. FaaS is also cheaper for users, as they only pay for the exact amount of resources they use during function execution. This is in contrast to Infrastructure as a Service (IaaS), where users pay for long-term reserved resources in the form of VMs. FaaS is an ideal candidate for applications with high data-level parallelism and/or intermittent activity (*e.g.*, online sites that are driven by fluctuating user load). However, the *serverless provider* still needs to provision, manage, and pay the *IaaS provider* for the VMs hosting its platform. This ties the cost of serverless to the cost of the underlying VMs. Worse, the serverless provider must pre-provision a large amount of VM capacity to provide fast elasticity and the illusion of infinite resources, while the FaaS users pay only for the resources their functions actually use.

Harvested resources. Fortunately, IaaS providers offer their surplus resources as VMs at a much lower price (and relaxed guarantees), such as Spot [42, 38] and Burstable VMs [40, 39]. Along similar lines, Harvest VMs [36] are an even cheaper and more efficient alternative. Each Harvest VM is evictable and has a minimum size, but it grows by harvesting unallocated CPU cores in its host server beyond this minimum. When a new "regular" (non-evictable) VM is placed on the server, the Harvest VM shrinks. The IaaS provider only evicts Harvest VMs when their minimum size is needed for a regular VM.

Serverless functions, which are mostly single-threaded and short-running [122], are a natural fit for running on harvested resources. Despite their low cost, Harvest VMs introduce two challenges: workloads can be evicted, and VMs have dynamic variations in terms of compute and/or memory resources. Not only do Harvest VMs have the potential to reduce the cost of hosting FaaS platforms, but they can also provide better performance at the same cost.

Our work. This chapter tackles the challenges of running serverless platforms on Harvest VMs. To understand the impact of evictions and of the variability in harvested resources on a FaaS platform, we first characterize both a FaaS offering (Azure Functions) and the resources available to Harvest VMs using production traces from Azure. We contrast the duration of function executions with the lifetime of Harvest VMs and the durations over which resources are available for harvesting. Our characterization suggests a good match between FaaS platforms and Harvest VMs. Thus, we next study how to adapt a FaaS platform to run on harvested resources.

To address Harvest VM evictions, we explore the space of regular and Harvest VMs mixes, for short- and long-running functions, and quantify the trade-off between cost and reliability. Using detailed simulations combining FaaS and Harvest VM traces, we

find that when running FaaS solely on Harvest VMs, evictions cause at most 0.0015% of invocations to fail.

To make this practical, we must address resource variations inherent to Harvest VMs. To this end, we design and implement a load balancer for FaaS platforms that places functions in VMs according to the availability of harvested resources. Our load balancer reduces resource contention while keeping the function cold start rate low.

Our implementation modifies OpenWhisk [107], a widely-used open-source FaaS platform, to monitor the availability of harvested resources and balance the load accordingly. Our experimental results demonstrate the performance improvement over the existing OpenWhisk load balancer and other widely used policies, achieving 22.6× throughput than vanilla OpenWhisk. We finally demonstrate the performance improvement and cost savings of serverless computing on Harvest VMs, compared to regular and Spot VMs. Under the same cost budget, serverless platforms hosted on Harvest VMs are able to achieve 2.2× to 9.0× throughput than regular VMs. When provisioned with the same amount of resources, serverless platforms hosted on Harvest VMs are 45% to 89% cheaper than regular VMs and 0% to 44% cheaper than Spot VMs.

4.2 Background and Related Work

Serverless and FaaS. Serverless computing, especially Functions as a Service (FaaS), is gaining popularity as the way to deploy applications on the cloud [120]. The FaaS programming model offers simplicity of just uploading application code without having to manage resources or configurations. In the FaaS platform we study, functions are logically grouped to form applications and the application is the unit of scheduling and resource allocation. The platform provides elasticity by automatically scaling up re-

sources with increasing load and scaling down to zero during idle periods. The user only gets billed for the resources consumed during function executions. All these properties make FaaS a compelling option for programming the cloud from the user's perspective.

The serverless provider faces the challenge of ensuring high performance while minimizing cost. To provide the illusion of always-on and infinitely scalable resources to the user, the provider needs to have the resources ready whenever a function is invoked. Shahrad *et al.* [122] show that 50% of functions execute for less than 1s and about 90% execute for less than 10s on average. A function can start quickly when the code is already in memory (*warm start*) and does not have to be brought in from persistent storage (*cold start*). Since these function executions are generally short lived, cold starts can dominate the overall execution time if the resources are not available at invocation time. To mitigate this, providers typically set a keep-alive threshold for which the function container is kept available after the invocation completes in anticipation of an upcoming invocation to the same function.

There has been a wealth of research on serverless computing, both to expand the set of applications that can use the model, and to improve the serverless infrastructure. Broadly, it spans: (a) scheduling policies for making serverless platforms cost-effective and performant [122, 78]; (b) performance-aware and cost-effective storage [83, 84, 103, 117]; (c) secure and light-weight container infrastructure [34, 106, 102, 137, 133, 125, 33]; (d) characterization of existing serverless workloads [122]; and (e) enabling applications to run in a serverless-native manner, including data processing and analytics [75, 112], video processing [63], ML training [46], DNA sequence visualization [88] and compilation [62]. We show that mindfully using cheaper resources without performance/reliability degradation is the right way to minimize the hosting cost of FaaS.

Harvest VMs. Harvest VMs were proposed in [36]. Users select and deploy them as they do any other VM. Each Harvest VM is defined by its minimum size (in terms of physical CPU cores, memory, disk space, and network bandwidth) and how many harvested physical cores they are capable of using. While the number of physical cores assigned to a Harvest VM may change dynamically, the other resources do not. The workload running on the Harvest VM can query the number of physical cores assigned to it in */proc* in Linux and the registry in Windows. The Harvest VM receives a 30-second notice before an eviction happens. These mechanisms allow the workload to take appropriate actions.

Users pay for the minimum size at a heavy discount, like those for Spot VMs, compared to regular VMs. For example, Spot VMs are 48% to 88% cheaper than the same size regular VMs in Azure [41]. The additional harvested cores are even cheaper because they vary over time. The total cost for the users is the sum of the minimum cost and the harvested one.

Despite offering a large amount of resources at low price, evictions and resource variation can impact the system reliability and performance [36]. This chapter addresses those issues.

Cluster scheduling and load balancing. A large body of work [44, 61, 67, 70, 74, 77, 111, 132, 55, 80, 110] has focused on cluster scheduling frameworks, such as Kubernetes [13] and Apache YARN [134]. However, these works assumed that the underlying resources (VMs or bare-metal servers) are constant over time.

In contrast, Harvest VMs may experience significant variation in their number of cores over their lifetime. Ambati *et al.* did adapt YARN to run on Harvest VMs [36]. However, the batch and Big Data analytics workloads common of YARN deployments

are quite different than those of FaaS platforms [115, 122]. For example, function executions are typically substantially shorter than data analytics tasks, so FaaS workloads can more easily adjust to the frequent changes in the numbers of cores. On the other hand, each function typically consumes fewer resources (e.g., memory) than an analytics task, meaning that many of them can be packed on the same VM so an eviction may affect more computations.

4.3 Characterization

While previous work has studied some production characteristics of Harvest VMs [36] and FaaS workloads [122], in this section we take a closer look with the goal of understanding how they might interact. We are interested in the impact of Harvest VM evictions and core variations on function executions. In particular, we look at the distribution of Harvest VM lifetimes and the distribution of intervals between Harvest VM core changes. Before each eviction, the Harvest VM receives a 30-second grace period, which can be used to stop sending new invocations to the VM, and to finish ongoing function executions. Invocations that last longer than 30 seconds are at risk of being killed, and below we pay particular attention to these long invocations. Compared to the previous characterization of FaaS workloads [122], we feature a new analysis addressing the issues involved in hosting FaaS on Harvest VMs: the impact of VM evictions and the capacity needed to host the FaaS workloads.

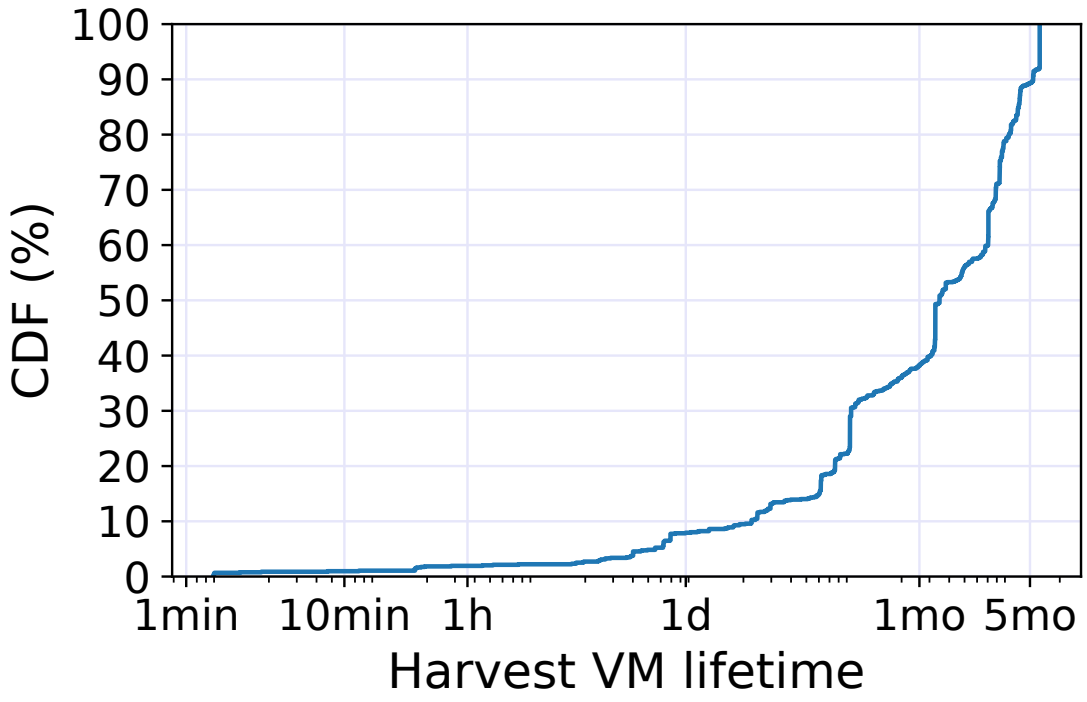


Figure 4.1: Distribution of the Harvest VM lifetime [36].

4.3.1 Harvest VMs

Evictions. To study Harvest VM evictions, we use a trace of the private cluster described in [36]. The trace includes 1075 Harvest VM instances deployed between October 8^{th} 2019 and March 28^{th} 2020. We include both evicted and not evicted Harvest VMs, and remove from the VM lifetime the 10 minutes required to install the FaaS platform and dependencies. Despite this overhead, 96.7% of all Harvest VMs are suitable for hosting FaaS. Figure 4.1 shows the lifetime distribution of these Harvest VMs. The average lifetime is 61.5 days, with more than 90% of Harvest VMs living longer than 1 day. More than 60% survive longer than 1 month.

Resource variability. To study the resource variation patterns of Harvest VM, we look at a smaller and more detailed trace of 37 Harvest VMs running in Azure production clusters between January 1^{st} and February 24^{th} 2021. To match the memory size of the

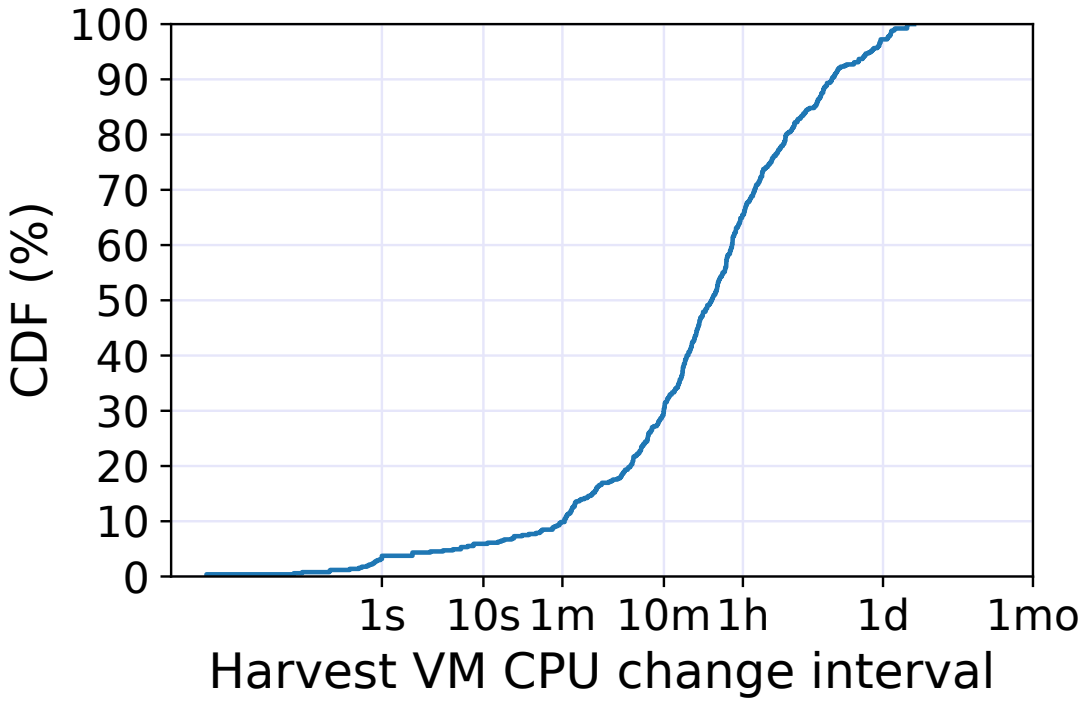


Figure 4.2: Intervals between Harvest VM CPU changes.

smallest Harvest VM (*i.e.*, 16 GB), the maximum CPUs of each Harvest VM is limited to 32. Figure 4.2 shows the distribution of intervals between changes in Harvest VM CPUs. The expected interval is 17.8 hours, with around 70% of them being longer than 10 minutes, and around 35% longer than 1 hour. 62.2% of the studied Harvest VMs experienced at least one CPU shrinkage and 54.1% experienced at least one CPU expansion. 35.1% VMs never experienced any CPU changes.

Figure 4.3 shows a histogram of individual CPU changes for the studied Harvest VMs. Positive numbers represent expansions and negative numbers represent shrinkage. The points at 0 represent the VMs that did not change during the period covered by the traces. The distribution tends to be symmetric with most of CPU changes falling within 20 CPUs. The average and maximum CPU change size are 12 and 30 for both shrinkage and expansion. Considering the maximum CPUs of the profiled Harvest VMs is 32, the size of the changes has a significant impact on instantaneous capacity of Harvest VMs.

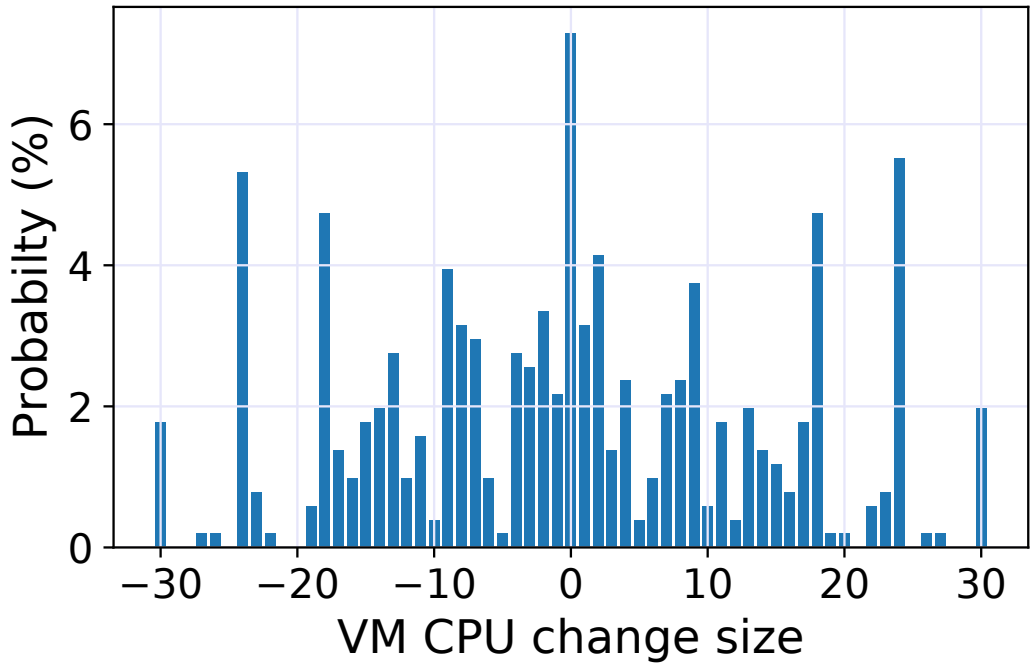


Figure 4.3: Distribution of Harvest VM CPU change sizes and correlation of change sizes and change interval.

Trace	F_{Large}	F_{Small}
Duration Data	Percentiles	Start/End Times
Granularity	Per App	Per Invocation
Dates	2021-01-31	2021-01-31 to 2021-02-13
#Apps	20,809	119
Invocations	910M	2.2M

Table 4.1: Details on the two FaaS traces used in the chapter.

We did not find a significant correlation between the size of the change and the change interval.

4.3.2 Serverless Functions

We now study the duration of function invocations. We obtained two traces (Table 4.1) of invocations from Azure Functions: F_{Large} is a coarse 1-day trace with invocation

duration percentiles for a subset of a cloud region, and F_{Small} is a detailed trace of a small cluster with precise invocation timings. We look at the overall trends with F_{Large}, and use F_{Small} for deeper analysis, including trace-driven simulations.

Duration per application. Figure 4.4 shows the distribution of maximum invocation durations per application from the F_{Large} trace, as well as those of the mean and other duration percentiles. The invocations are generally short. The graph shows the 30-second grace period of Harvest VM eviction. Invocations shorter than this are safe from evictions, while longer invocations could be terminated. 20.6% of the applications have at least one invocation (maximum) longer than 30 seconds. We refer to these applications as "long" applications. 16.7% and 12.3% of applications have 99.9^{th} and 99^{th} percentile durations longer than 30 seconds, respectively.

Figure 4.5 compares the same distributions between the two traces. The traces are similar with respect to the tails of the per-application invocation durations, with the applications in the F_{Small} trace having higher fractions of longer invocations. This is acceptable for our purposes, as it makes our analysis more pessimistic. We base our analysis in the remainder of the chapter on the F_{Small} trace.

Durations per invocation. The F_{Small} trace allows us to look at the duration of every invocation. Figure 4.6 shows the latency distribution of all considered invocations. The vast majority are short, with more than 85% of invocations shorter than 1 second, and 96% of the invocations shorter than 30s. The longest recorded invocation is 578.6 seconds.

Long applications. In terms of sensitivity to Harvest VM evictions, only 4.1% of the invocations are 'long', but these long invocations take over 82.0% of the total execution time of all invocations. At the granularity of application, 58 applications (48.7% of

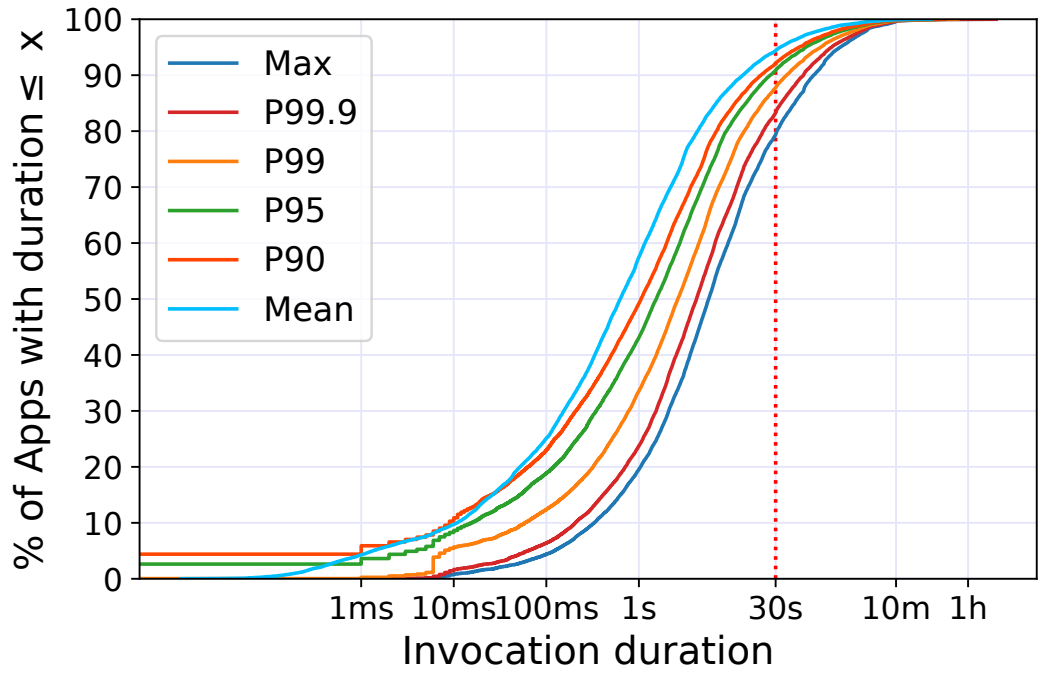


Figure 4.4: CDFs of the average and top percentiles of the invocation durations per application in the F_{Large} trace.

all) are long applications. These long applications take up 67.5% of all invocations and 99.68% of the total invocation time. These long invocations (and applications) are vulnerable for evictions if placed on a Harvest VM. As we see in §4.4, naïvely allocating the long applications to regular VMs, and running the others on Harvest VMs may be too conservative a strategy, with very modest gains.

Looking closer, Figure 4.7 shows the duration distribution of the long applications, where each point on x-axis corresponds to one application, and the error bar shows the standard deviation of the durations. There are big gaps between the max and mean duration of long applications, especially for applications with max duration longer than 100 seconds, indicating that long applications fall under this category mainly due to a small fraction of invocations in the tail of their duration distribution. We use this to our advantage in the next section.

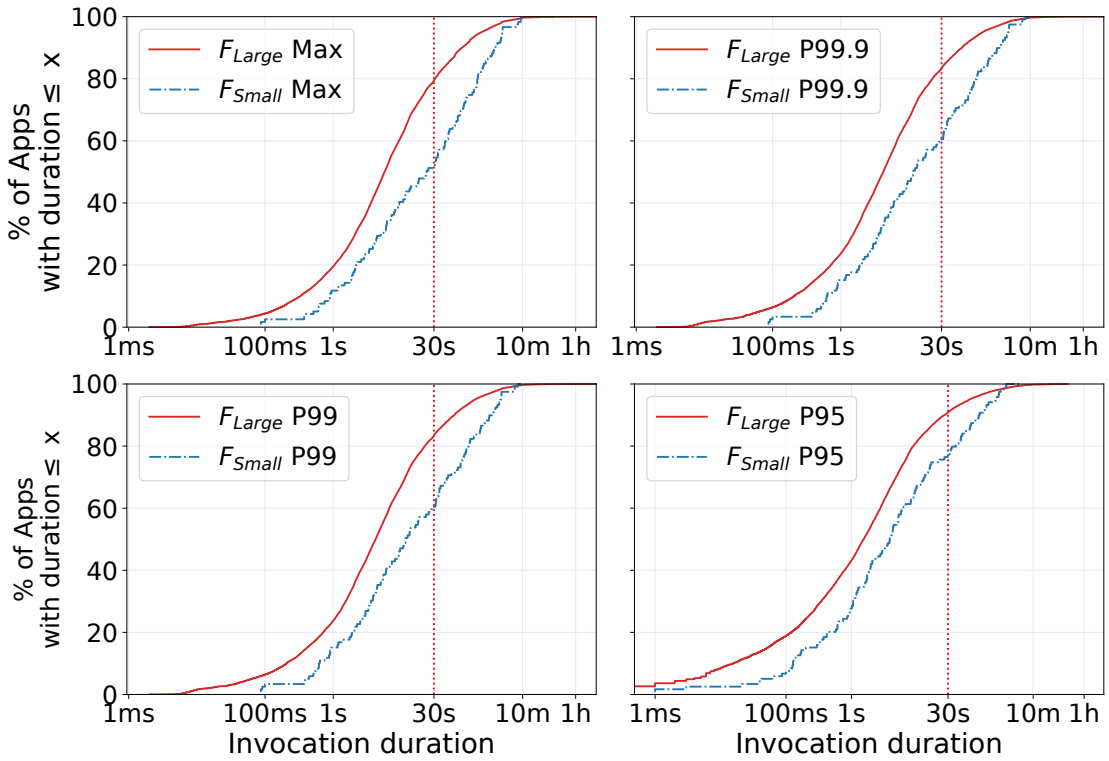


Figure 4.5: Invocation durations per app for F_{Large} and F_{Small}.

4.3.3 Implications

Combining the characteristics of Harvest VMs (Section 4.3.1) and serverless workloads (Section 4.3.2) shows that, intuitively, FaaS workloads are a good fit for Harvest VMs. The short duration of the majority of the invocations (only 4.1% are longer than 30 seconds) and the relatively much longer Harvest VM lifetime (more than 90% of Harvest VMs live longer than 1 day) make serverless workloads unlikely to be affected by Harvest VM evictions. Based on this intuition, in Section 4.4 we use trace-drive simulations to more precisely characterize the reliability of serverless compute on Harvest VMs.

Resource variation on Harvest VMs is much more common than evictions, but compared to the short duration of most invocations, the number of CPUs of Harvest VMs can be considered relatively stable: 70% of CPU change intervals are longer than the

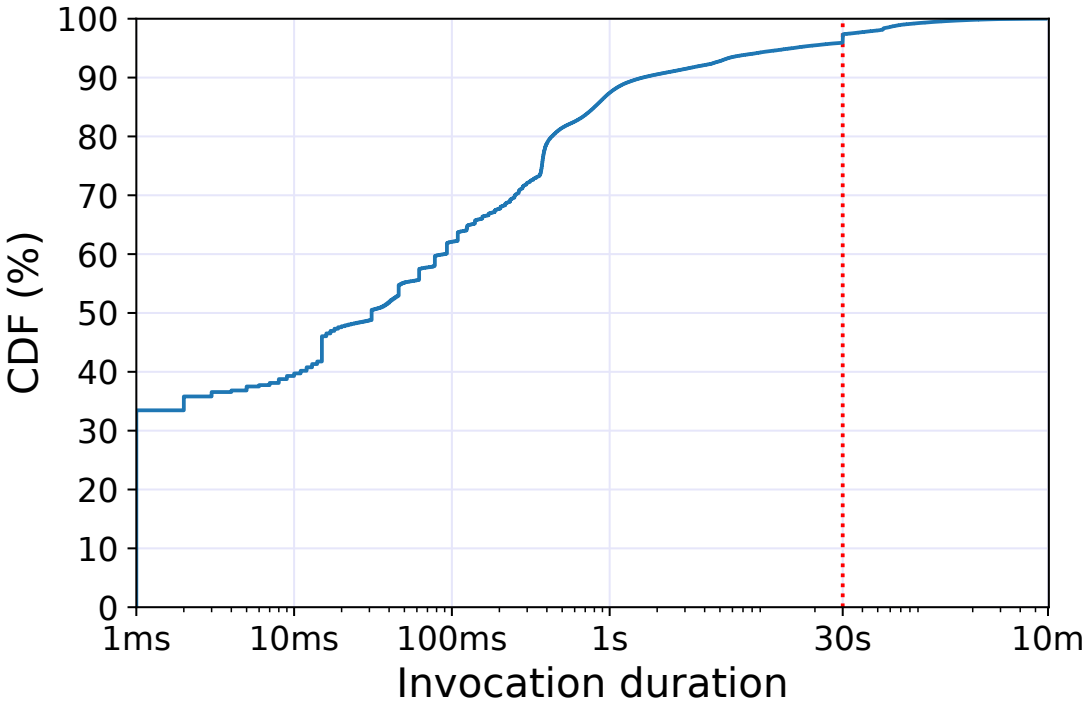


Figure 4.6: Durations of all invocations in the F_{Small} trace.

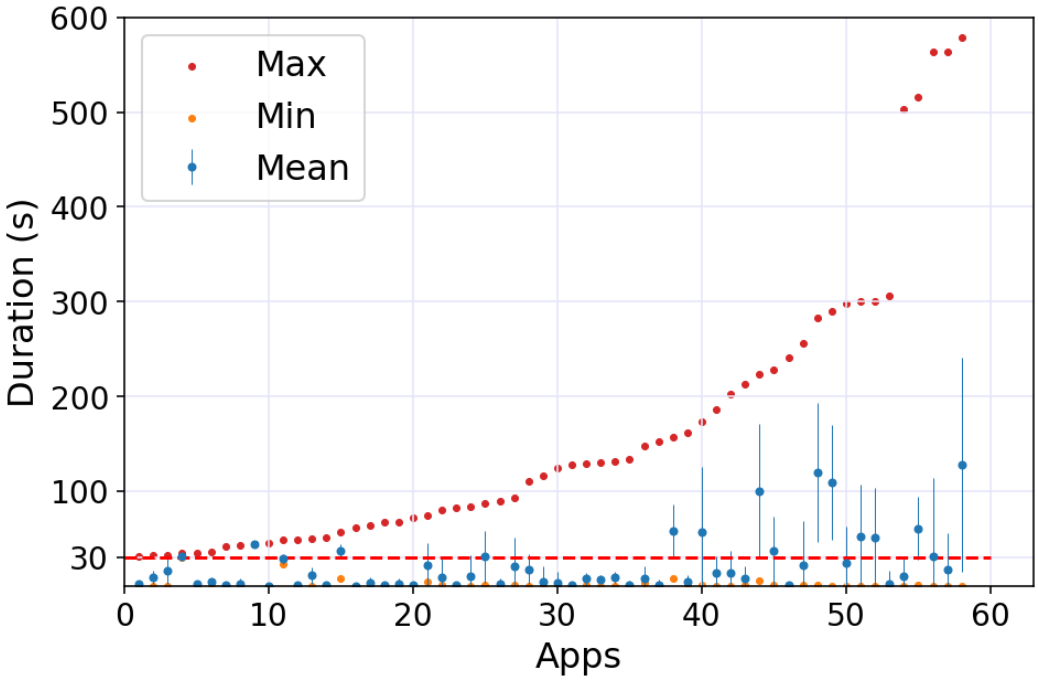


Figure 4.7: Durations of long applications invocations.

longest invocation in the studied serverless workload trace (578.6 seconds). However, because of the frequency and magnitude of resource changes (Figure 4.3), Harvest VM-aware load balancing is essential to guarantee system performance. In addition to this, even if mostly stable, Harvest VMs tend to be more heterogeneous than regular VMs, reinforcing the importance of proper load balancing.

4.4 Handling Evictions

In this section, we study the impact of Harvest VM evictions when running serverless workloads. When an eviction occurs, any function running at the time fails. *What is the best strategy to eliminate or minimize these failures?*

4.4.1 Methodology

While the comparison of the distributions in the previous section provides bounds on the failure rates, the interaction of evictions and long executions is not trivial, and we resort to trace-driven simulations to answer this question.

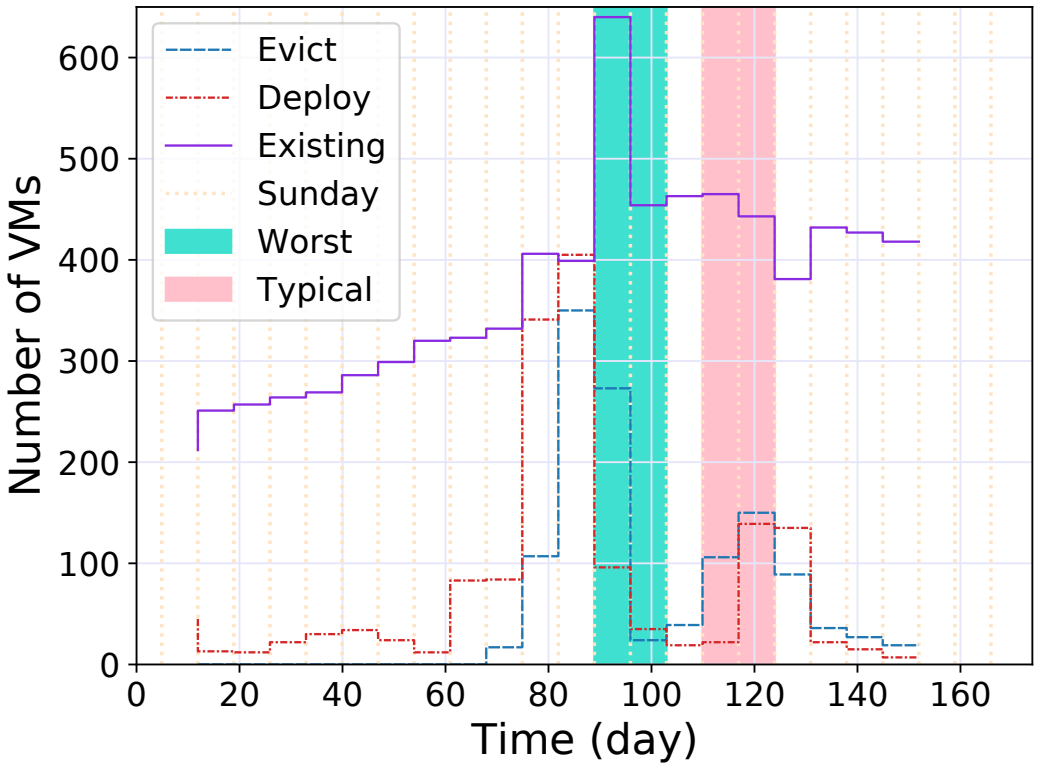
We used the Harvest VM trace from Figure 4.1 and the F_{Small} functions trace (§4.3). Since the Harvest VM trace (173 days) is longer than the serverless workload trace (14 days), we select a 14-day period from the Harvest VM trace which aligns with the serverless workload trace. Figure 4.8a shows, for the 14-day period starting at each Sunday (dotted vertical lines), the total number of VMs, and the number of VM creations and evictions. We use the Harvest VM eviction rate defined as number of VM evictions over the number of existing VMs, as the metric to categorize the Harvest VM trace periods. The average eviction rate of all 14-day periods is 13.1%. We select two periods:

(1) one with the max VM eviction (86.4%), as worst case, and (2) one with an eviction rate close to average (8.4%), as the typical case. Starting days of the worst and typical cases are marked as *Worst* and *Typical* in Figure 4.8a.

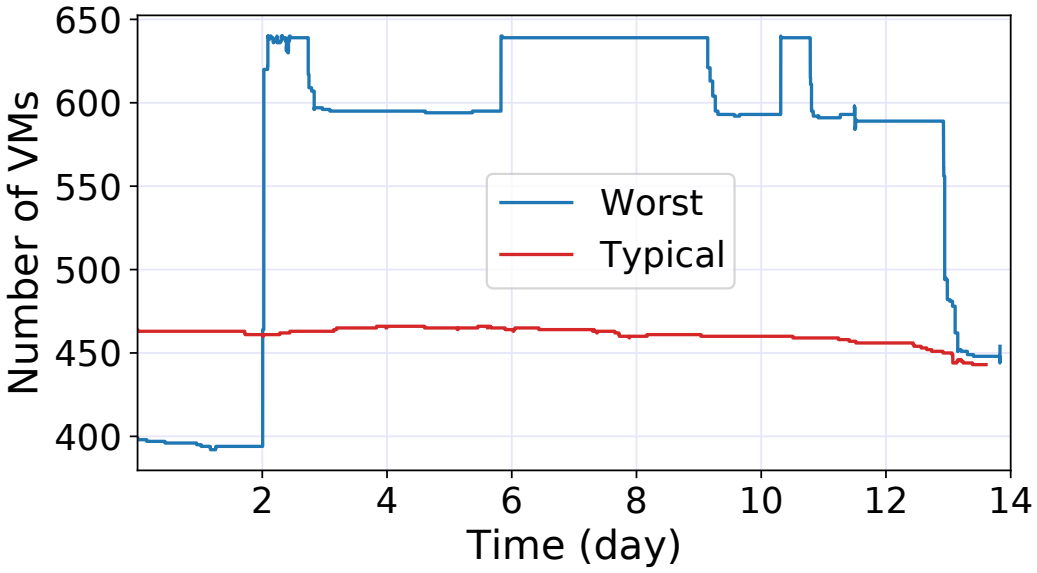
We simulate the serverless framework as a global pool of containers; an invocation is able to use any existing container of the same application and randomly chooses one if there are multiple candidates. Invocations have a keep-alive time set to 10 minutes (the default in OpenWhisk [107]). A container is removed if it does not execute any invocations for the entire keep-alive period. Each container is randomly allocated to VM that has not been warned of eviction. The number of concurrent invocations that each container can host is set to 1. Since our traces do not record CPU usage, we assume that the CPU usage of all applications is identical.

For Harvest VMs, when we receive the 30-second eviction warning for a VM, the load balancer stops sending new invocations to it. Pending invocations continue to execute on the VM and fail if they do not complete before the VM eviction. In the event that resources start to decline below a pre-configured threshold, it spins up additional VMs.

For each 14-day Harvest VM trace snippet, we run the simulation 1000 times and show the aggregated results. Our simulation models the key components of serverless frameworks, including container pool and keep-alive. It can model potential future workload changes by simply acquiring new traces, assuming no changes to the serverless framework.



(a) 14-day period in the entire trace.



(b) Selected 14-day periods for simulation.

Figure 4.8: Harvest VM creations and eviction patterns.

4.4.2 Combining Regular and Harvest VMs

Strategy 1: No failures. We start with the most conservative provisioning where all long applications (*i.e.*, those with at least one invocation longer than 30 seconds) are allocated in regular VMs and the rest in Harvest VMs. This guarantees that no invocation longer than 30s will run on Harvest VMs, but is the least efficient provisioning strategy. Section 4.3.2 showed that long applications take up to 67.5% of all invocations but 99.7% of the invocation time. However, we also need to account for the keep-alive period to prevent cold starts. We ran a simpler version of our simulation here to estimate the computation capacity taken by the two application types, while accounting for their arrival times and keep-alive behavior.

For 10-minute keep-alive, the simulation shows that only 12.0% of computation capacity can be hosted by low-cost Harvest VMs. While this is much higher than the fraction of execution time for short applications (0.32%), it is significantly lower than the fraction of invocations that corresponds to short applications (32.5%). This is due to their shorter invocation times on average, and to their inter-arrival times. Figure 4.9 shows that a larger fraction of the inter-arrival times for short applications is below 10s, and multiple close invocations reduce the wasted idle time due to keep-alive. We verified that these results do not change significantly for different keep-alive periods ranging from 1 minute to 24 hours. Ultimately, this strategy is too conservative, and 94% of the invocations that run on the regular VMs are still short.

Strategy 2: Bounded failures. Given the high operational cost of Strategy 1, we study a relaxation of the bound on failures caused by eviction. If we are willing to tolerate a small fraction of eviction failures, we can allocate more applications to Harvest VMs, and trade reliability for efficiency.

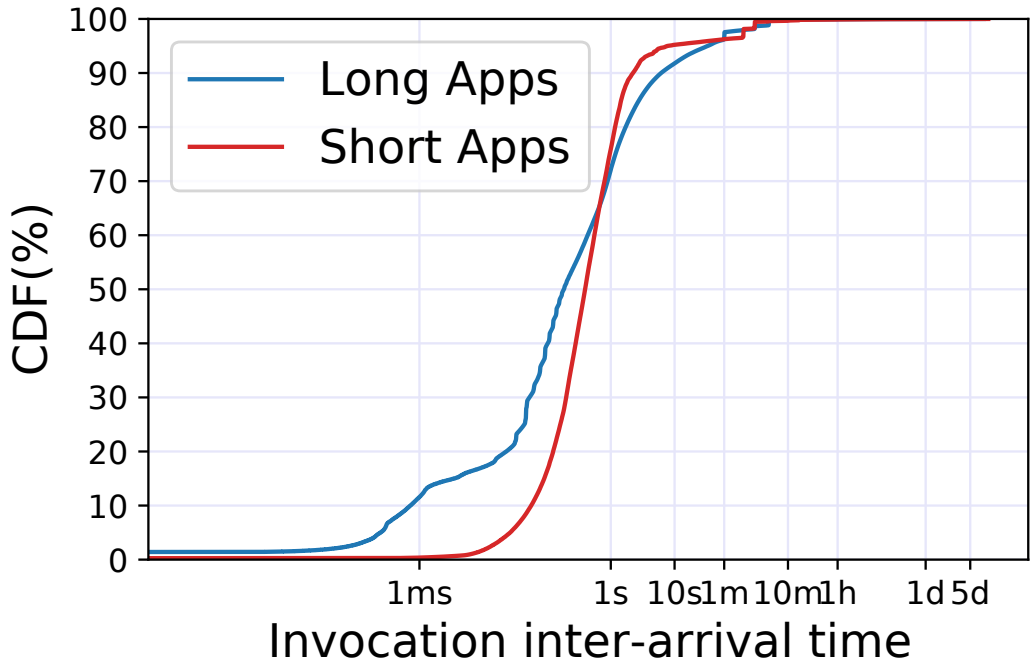


Figure 4.9: Inter-arrival times for short vs. long apps.

We can provide an upper bound $(100-x)\%$ (say, 1%) on the per-application eviction failure rate by allocating to regular VMs applications with the x^{th} (say, 99^{th}) percentile duration longer than 30s, instead of the maximum. In effect, some long applications from Strategy 1 are allocated to Harvest VMs in this strategy, but only those where $(100-x)\%$ of the invocations are longer than 30s.

To characterize the trade-off between reliability and efficiency of the policy, we perform the same trace-driven simulation as in §4.4.2, and sweep the percentile x from 95 to 99.9, with increments of 0.1. Figure 4.10 shows the results, with the decision percentile in the x-axis, and the resulting fraction of computing capacity used by Harvest VMs.

In summary, bounding the failure rate to less than 0.1% allows 28% of computation to be hosted by Harvest VMs. A rate lower than 1% allows 45.7% of computation to be hosted by regular VMs. Although efficiency improves compared to Strategy 1, it is

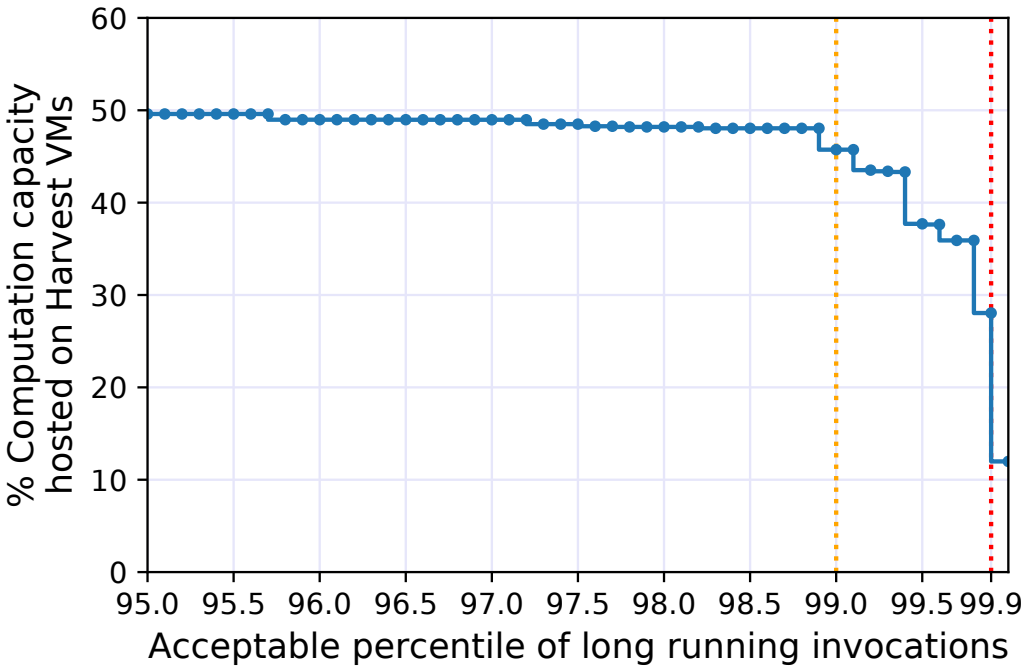


Figure 4.10: Fraction of Harvest VM capacity versus acceptable percentile of per-app long invocations.

still pessimistic, as most invocations that run in regular VMs are still short, and even the long invocations would only fail if they run in a Harvest VM and start less than 30s before an eviction.

4.4.3 Running on Harvest VMs

Strategy 3: Live and Let Die. We next examine running a full serverless workload solely on Harvest VMs. We ran the full simulation described in §4.4.1. For the *Worst* period in the Harvest VM trace (*i.e.*, max VM eviction rate), the average invocation failure rate is 0.0015% (99.9985% success rate). The *Typical* period has a failure rate of 3.68×10^{-8} (*i.e.*, "7 nines" of reliability).

Intuitively, failures caused by VM evictions are rare because they require two low-

probability events to happen simultaneously: a Harvest VM gets evicted *while* it is running a long invocation. VM evictions are also correlated and frequently happen in bursts, with a large number of VMs evicted within a few seconds, as shown in Figure 4.8b.

Cold starts are also minimal when the workload runs on Harvest VMs. The average simulated cold rate is 1.1967% in the Typical period, and 1.1981% in the Worst period, increasing by 0.0084% and 0.1254% compared to regular VMs.

4.4.4 VM Migration/Snapshotting

An alternative, or even complementary approach, to increase the reliability of the serverless framework hosted on Harvest VMs is to use VM live migration [52] or snapshot/restore [60, 133]. The idea is to run serverless applications in nested VMs hosted by Harvest VMs, and migrate the nested VMs that correspond to long invocations when the Harvest VM is warned of eviction. The main metric, however, is not the downtime of the application, but the total time for which the source VM must be available. Because of the low invocation failure rate from Strategy 3, we leave using VM migration to improve system reliability as future work.

4.4.5 Conclusion

When running solely on Harvest VMs, the failures caused by VM evictions are rare while fully utilizing the cheap harvested resources. This is caused by the low joint probability of a rare long-running execution during a Harvest VM eviction. As a result, in the rest of the chapter, we assume all applications are hosted on Harvest VMs.

4.5 Handling Resource Variability

In this section, we develop a resource variation-aware load balancing policy for serverless frameworks on harvested resources. We start with the well-known algorithm *join-the-shortest-queue (JSQ)* [72], which aims to minimize the resource contention caused by CPU variation. Based on that, we then present our *min-worker-set (MWS)* algorithm, which aims to reduce the container cold start rate for serverless workloads while reducing resource contention.

4.5.1 Join-the-Shortest-Queue (JSQ)

JSQ is a CPU-aware load balancing algorithm. The load balancer monitors the compute load of each backend VM and allocates an invocation to the VM that has the least amount of pending work. This effectively reduces queueing time and resource contention, leading to shorter end-to-end latencies.

Since the ground truth of pending compute work is unknown in advance, we approximate it with a weighted sum of CPU and memory utilization $w_c \frac{cpu_{used}}{cpu_{avail}} + w_m \frac{mem_{used}}{mem_{avail}}$, with $w_c > w_m$ to reflect the scarcity of allocated CPUs. We show that the weighted utilization of CPU and memory is a better usage metric than the number of pending invocations (queue length) at an invoker, or the sum of expected resource usage of pending invocations (weighted queue length). This is because queue length does not account for varying function resource needs, and weighted queue length can deviate from the ground truth, due to insufficient samples and different function inputs. The utilization metric also captures the variation of allocated CPUs of Harvest VMs, and avoids starvation by stopping assigning invocations to VMs that suffer from excessive resource

shrinkage. In terms of overhead, the complexity of each scheduling operation is $O(N)$, where N is the number of backend VMs in the system. The scheduling overhead can be reduced by randomly sampling a subset of d backend VMs and choosing the least loaded one [45, 136, 110], although at the expense of scheduling quality.

4.5.2 Min-Worker-Set (MWS)

In serverless computing, the end-to-end latency of an invocation includes cold start time, queueing time and execution time. Although JSQ can reduce queueing time by preventing long queues and execution time by alleviating resource contention, it can potentially increase the cold start rate and harm the end-to-end latency. Assuming that a function has a Poisson arrival process with arrival rate λ, and the serverless platform has N backend VMs, JSQ will distribute the invocations across all N backend VMs. The resulting invocation arrival rate on each backend VM will be $\frac{\lambda}{N}$. In a large system (*i.e.*, large N), the expected inter-arrival time $\frac{N}{\lambda}$ is more likely to be larger than the container keep-alive time of serverless platform, increasing the chance of cold starts.

We design the MWS algorithm to jointly reduce queueing time, execution time, and cold starts. This is inspired by the intuition that, in the common case, where the compute resources of the system are not overloaded, slight imbalance of invocation assignment among invokers is unlikely to cause resource contention and queueing leading to increased latency. MWS consolidates each function to a minimal set of k backend VMs that have adequate resources to accommodate all invocations of the function. The invocation arrival rate on individual backend VMs becomes $\frac{\lambda}{k}$. With $k \ll N$, the invocation inter-arrival time in MWS is much shorter than JSQ. Thus, it is very likely to be shorter than the container keep-alive time, enabling warm starts. The sketch of MWS is shown

in Algorithm 3. For each function f, the load balancer assigns it a home VM as the beginning of the search process, and estimates its resource usage u_f as the product of requests per second (RPS), expected resource usage, and expected duration. The load balancer keeps adding new VMs to the worker set s until the total usable resources r of all VMs in the set s exceeds the estimated usage of the function u_f. Finally, the load balancer picks the least loaded VM in the worker set s to execute the invocation, where the load is defined as the weighted sum of CPU and memory utilization as in JSQ.

Algorithm 3: Min-worker-set (MWS) algorithm

Input: Function f;
Function: Expectation E; Consistent hashing CH;
Variable: Requests per second RPS_f;
Variable: CPU usage CPU_f;
Variable: Invocation latency lat_f;
$u_f = RPS_f \cdot E(CPU_f) \cdot E(lat_f)$;
$r = 0$, $s = \emptyset$;
$VM = CH(f)$;
while $r < u_f$ **do**
    $r = r +$ usable_resources(VM);
    $s = s \cup VM$;
    $VM =$ next(VM);
end
return $argmin_{VM}\{\text{load}(VM) \mid VM \in s\}$

In the common case that the system is not overloaded, MWS is more scalable than JSQ, with minimum scheduling overhead. For each invocation, the controller only needs to search for the least loaded invoker in the worker set of the function (*i.e.*, usually a small number) rather than searching among all invokers. In the worst case that the system is running at full utilization, the scheduling overhead increases with the load of the system and converges to JSQ as MWS spans all backend VMs. Compared to JSQ, MWS can also reduce the number of functions allocated to each VM, thus reducing the storage space occupied by function images.

Dealing with VM evictions. Harvest VM evictions can be detrimental to the perfor-

mance of the MWS algorithm, because VM failure and redeployment lead to variation in the number of VMs in the system, and thus reshuffling of home VMs for all functions, making cold starts dominant. To minimize the number of functions that need to be reshuffled and thus minimize cold starts, we use consistent hashing. Thus, whenever the number of VMs changes, home VMs are only reshuffled for a minimal number of functions.

In consistent hashing [81], all VMs in the system are assigned a hash ID within $[0, I]$ where I is much larger than the number of VMs in the system, so that VMs are uniformly distributed in the ID space. Conceptually, all VMs in the system are organized into a ring with VM IDs increasing clockwise, except VM I, whose next VM in the ring is VM 0. Functions are mapped to and uniformly distributed in the same ID space $[0, I]$ and are assigned next VM in clockwise direction (to the ID of the function) as home VM. As the VM IDs are uniformly distributed, the expected number of functions assigned to each VM are identical. When an existing VM crashes or a new VM joins the system, only functions originally assigned to the crashed VM or the new VM are reshuffled.

4.6 Implementation

We implement our proposed resource-variation-aware load balancing scheme on OpenWhisk [107], a popular open source serverless platform developed by IBM. In this section we first describe the architecture of OpenWhisk and then the changes we made for Harvest VM-aware load balancing.

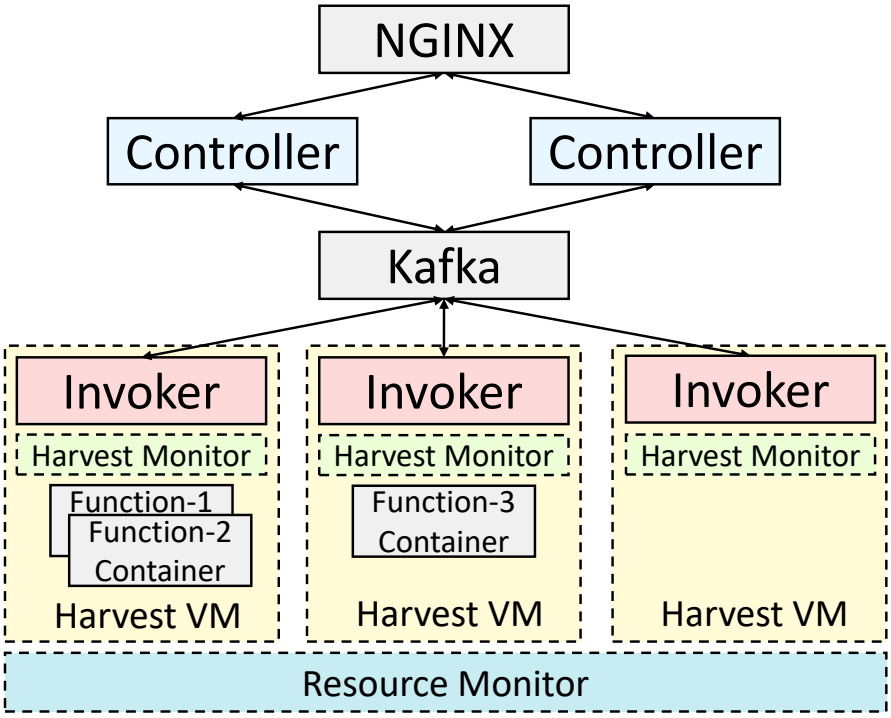


Figure 4.11: Architecture of our resource-variation-aware load balancing solution on OpenWhisk. The dotted lines show our modifications and components not present in vanilla OpenWhisk.

4.6.1 OpenWhisk Architecture

Figure 4.11 shows the architecture of OpenWhisk including the modifications we have made represented by in dotted lines. NGINX acts as a reverse proxy of the system and exposes a public HTTP endpoint to clients and forwards user requests to Controllers. The Controller performs load balancing and selects an Invoker instance to execute the function invocation. OpenWhisk by default implements memory bin packing: the Controller keeps track of memory usage of all pending invocations that are issued and iteratively directs all incoming invocations to one Invoker until the memory quota of that Invoker is exhausted. Controllers do not communicate with each other, and each Controller has access to all Invokers. The message delivery system between Controllers and Invokers is implemented using Kafka [3]. Invokers are usually deployed per VM and

each manages a pool of containers, which are Docker containers by default. Depending on whether a suitable container exists, a function invocation is assigned to an existing container (warm start), or a newly created one (cold start). Existing containers are removed after a fixed keep-alive period (10 minutes by default) and when usable memory is inadequate to allocate a new container. Invocation results are stored in CouchDB for later retrieval.

4.6.2 Harvest VM-Aware Load Balancing

We modify both the Invoker and the Controller to implement the resource variation-aware MWS load balancing algorithm.

Invoker. We modify the Invoker so it can efficiently use the dynamically changing number of available CPUs. We introduce a module called *Harvest Monitor* in each Invoker that is responsible for periodically gathering: (a) the latest number of CPUs allocated to the Harvest VM using Hyper-V Data Exchange Service [43]; (b) the cumulative CPU time using *cpuacct.usage* interface from *cgroups* [89]; and (c) any scheduled deallocation event for the VM using Azure Metadata Service [100]. This information is embedded into the health pings that the Invoker sends to the Controller every second.

All function containers for the same user run in the same cgroup so that we can gather CPU utilization statistics. For each function invocation, the Invoker collects its (a) execution duration and (b) CPU usage by querying the cgroup for its container. The Invoker embeds the information in the invocation response message back to the Controller. In addition, the Invoker performs admission control by computing the current utilization as $(\frac{cpu_{usage}}{cpu_{avail}})$; if this is higher than a predefined threshold, new function invocations are delayed.

Controller. We modify the Controller to receive the additional information collected by the Harvest Monitors through the Invoker health pings. The Controller updates its local data structures with this information (off the critical path, using Scala Actors). It maintains (a) CPU usage, (b) available CPUs, and (c) eviction notifications events for each Invoker. If an Invoker has an eviction notification, the Controller stops sending new invocations to it. The Controller maintains local per-function histograms of the observed execution times and CPU usage. Each Controller independently constructs these histograms which eventually converge to similar values as more samples are collected. The Controller also maintains a per-function invocation arrival rate which is periodically updated. We multiply the arrival rate observed locally with the number of Controllers in the system (available at startup) to get an estimated total invocation arrival rate.

We use the expected values computed from the execution time and CPU usage histograms along with the estimated total invocation arrival rate as inputs to execute the MWS algorithm for the function. To mitigate the potential user load oscillation and smooth the worker set size changes, we set a minimal interval of 30 seconds between worker reductions.

Finally, the Controller also maintains the mapping of functions to their hash ID (used in assigning home VM based on consistent hashing) and hash IDs to list of functions (used for function to home VM assignment update in the face of Invoker arrival/departure) as described in Section 4.5.2.

Resource Monitor. We introduce a separate module per deployment, called *Resource Monitor*, to track the resource variation in our system. It periodically queries for the total available resources (*e.g.* CPUs) and spins up new VMs to maintain a minimum pool of available resources, if they fall below a pre-configured threshold. As mentioned in Section 4.4.1, this is important because reduction in the CPUs or eviction of Harvest

VMs reduces the available resources and can adversely impact service quality.

4.7 Evaluation

We first demonstrate the benefits of the MWS algorithm compared to JSQ and the default load balancing algorithm of OpenWhisk. Then we demonstrate the benefits and cost savings of Harvest VMs for hosting serverless workloads.

4.7.1 Experiment Setup

For this evaluation, we deploy OpenWhisk (PR#4611) [108] on Azure with Ansible [37]. We use one controller VM and a variable number of invokers with their own VMs. The controller VM contains core OpenWhisk components, including two controllers, NGINX and CouchDB. In this section, we use cluster size to refer to the number of invoker VMs.

We port multiple Python serverless functions from FunctionBench [82] to OpenWhisk as the benchmark (Table 4.2). We create a Docker image for each function for a total of 401 functions. We use Locust [14] to generate the workload with a Poisson arrival process, and MinIO [101] as the object store to serve the input data. We use the 99^{th} percentile latency denoted P99 as the SLO metric and set it to 50 seconds, which clearly indicates saturation for our benchmarks.

The experiments use actual Harvest VMs (where we cannot control how their resources vary) and Harvest VM traces. When using traces, each trace corresponds to a Harvest VM. To emulate the CPU changes from the trace on a regular VM, we use

Functions	Description
Floatop	Sine, cosine & square root
Matmult	Square matrix multiplication
Linpack	Linear equation solver
Chameleon	HTML table rendering
Pyaes	AES encryption & decryption
Image processing	Flip, rotate, resize, filter & grayscale images
Video processing	Grayscale video
Image classification	MobileNet inference
Text classification	Logistic regression

Table 4.2: The examined serverless functions from FunctionBench [82] and their description.

cgroups to set the CPU limit of the parent Docker group of all user invocations. Each experiment runs for 20 minutes unless otherwise stated.

4.7.2 Impact of Load Balancing

First, we compare three load balancing algorithms: min-worker-set (MWS), join-the-shortest-queue (JSQ), and vanilla OpenWhisk (*Vanilla*). We deploy OpenWhisk with 10 invokers, each hosted by a regular VM with 32 CPUs and 128 GB of memory. The CPUs of the invokers are asymmetric, with the maximum of 28 and the minimum of 5, to mimic the resource heterogeneity in Harvest VM clusters. Figure 4.12 depicts the $P99$ latency of the three algorithms.

Throughput. We evaluate the throughput without breaking the SLO of each policy. MWS achieves a throughput 22.6× higher than the vanilla OpenWhisk load balancing. *Vanilla* has the worst throughput because it only considers memory and keeps allocating invocations to an invoker until the memory capacity of the invoker is exhausted. However, because of the scarcity of CPUs on some Harvest VMs, CPU tends to satu-

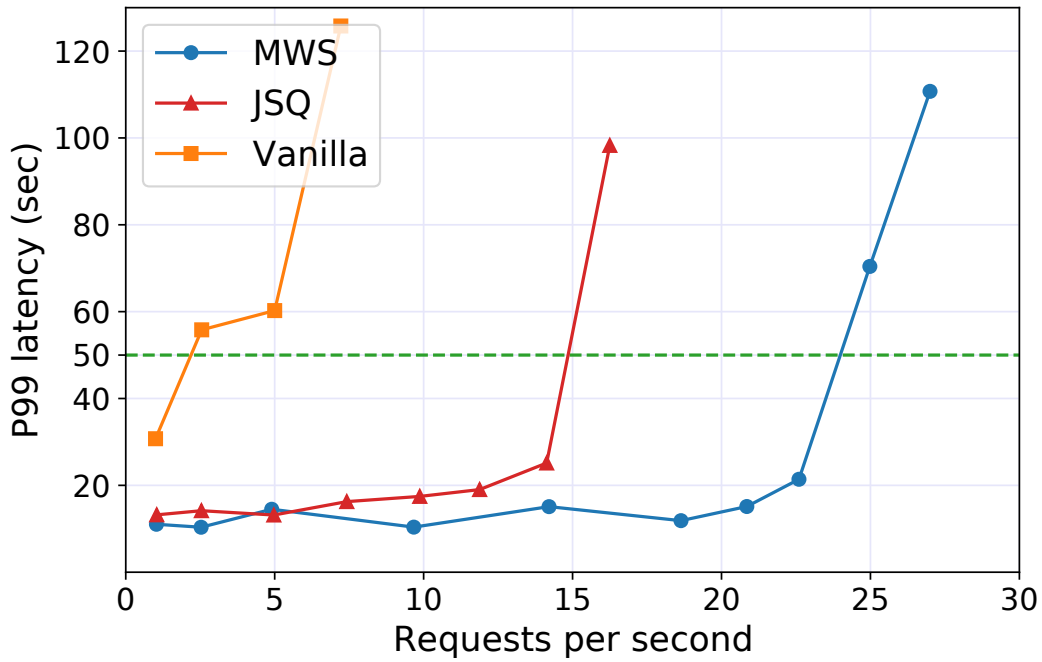


Figure 4.12: P99 latency across load balancing algorithms.

rate at much lower load than memory. The CPU saturation caused by *vanilla* is also exacerbated by the heterogeneity of VM CPUs in the test cluster because even if there are additional CPU resources in the cluster, the invoker with the least CPUs will always saturate at low loads. MWS has a throughput 1.6× higher than JSQ because it improves locality.

Cold starts. Better locality reduces the cold start rate. Figure 4.13 compares the cold start rate of MWS and JSQ at their non-saturating loads. At the same input loads, MWS reduces cold starts between 56.0% and 75.9%. Figure 4.14 compares the latency of both policies. It shows that reducing cold starts, we also reduce the latency. With our strategy, we can provide the same latency with fewer VMs.

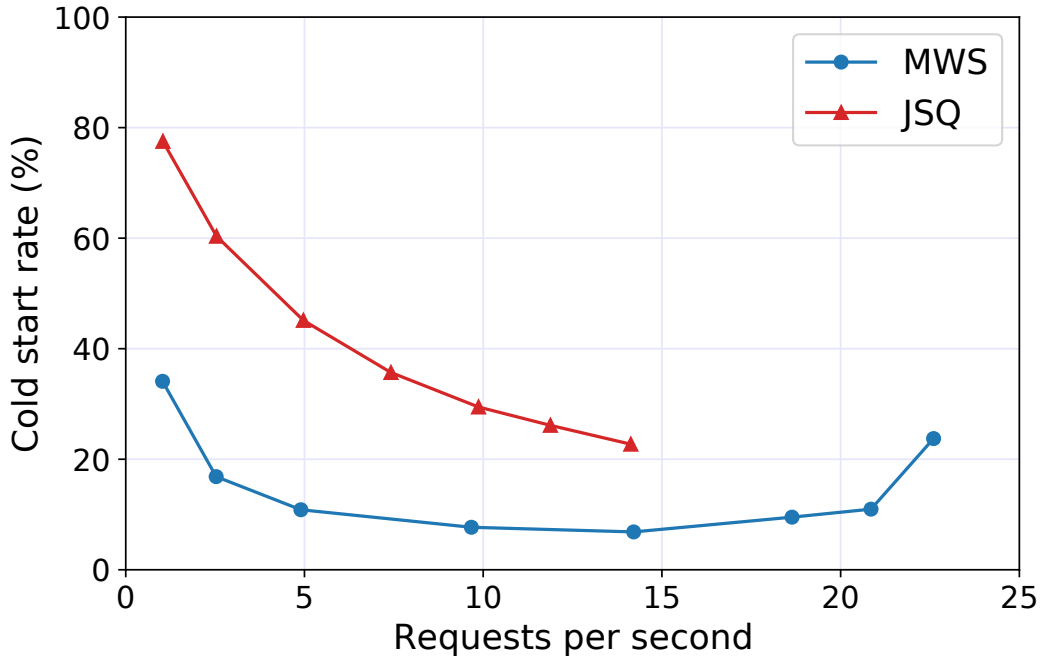


Figure 4.13: Cold start rate of MWS vs. JSQ.

4.7.3 Impact of Resource Variability

We use Harvest VM traces that have frequent CPU changes with large change sizes to show the worst-case performance. Although CPUs of Harvest VMs are relatively stable in the normal case, they can also experience frequent and significant resource changes as depicted in Figure 4.3. Frequent CPU changes are challenging to handle since they require the load balancer to detect the changes and adjust task assignment promptly. Significant CPU shrinkage has a direct impact on system performance since pending activations on the Invoker that experiences significant shrinkage are prone to severe resource contention, especially at high loads.

To study the worst-case performance of Harvest VMs, we select a set of Harvest VM traces that have both extremely frequent and significant CPU changes. Specifically, we choose 8 real Harvest VM traces among the traces with the highest change frequency and large change size, and also synthesized 2 traces to control the total capacity of the

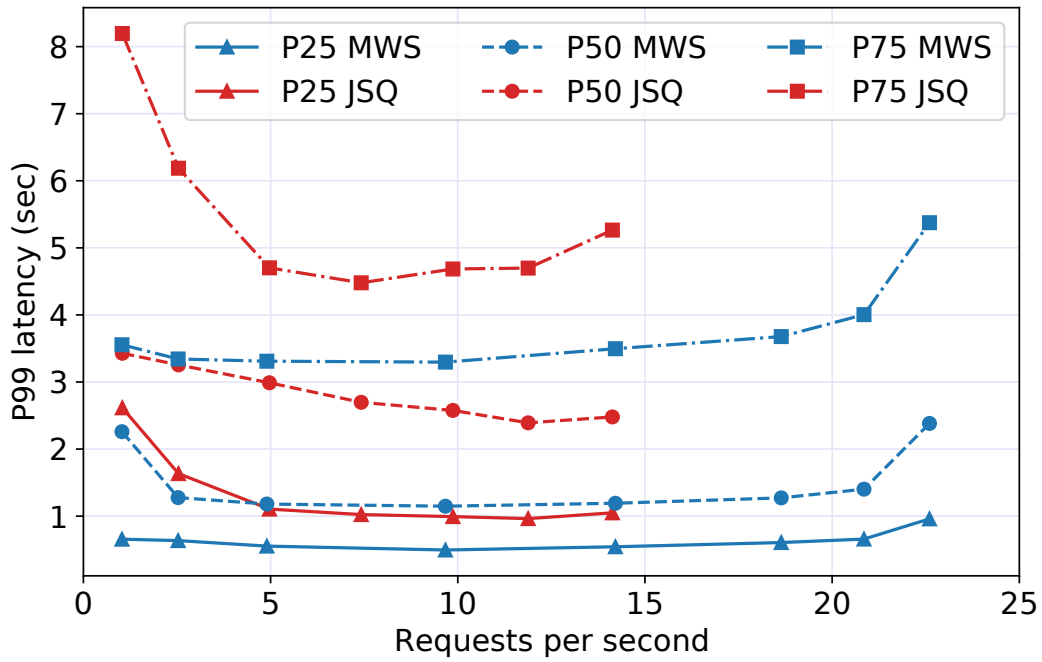


Figure 4.14: Low percentile latency of MWS vs. JSQ.

cluster. The average CPU change interval in the set of 10 traces is 3.6 minutes, orders of magnitude shorter the expected CPU change interval for the common case as discussed in Section 4.3.1. The traces also include significant CPU shrinkage, with the maximum shrinkage size being 26 CPUs, meaning that 81.3% of all CPUs are suddenly taken away from an Invoker.

We compare the performance of this actively changing Harvest VM cluster ("Active") to two clusters: "Normal", a Harvest VM cluster with normal variations, and "Dedicated", a cluster with dedicated resources using regular VMs. All three clusters have 180 CPUs total. The "Normal" harvest cluster has stable per-VM CPUs, but the size of each Harvest VM varies, with the largest VM having 28 CPUs and the smallest VM having 5 CPUs. The "Dedicated" cluster has both stable and homogeneous per-VM CPUs.

We compare the performance of these three clusters in Figure 4.15. "Active"

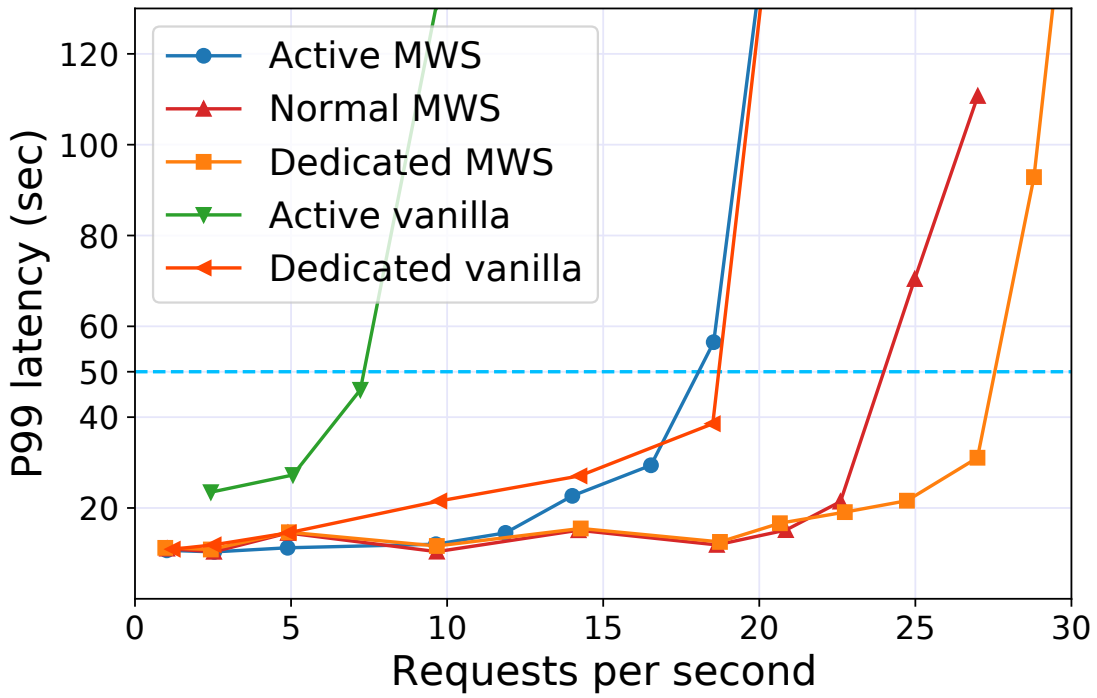


Figure 4.15: Performance of harvest clusters in normal case ("Normal"), under frequent and significant CPU changes ("Active"), and of the "Dedicated" cluster.

achieves 73.1% throughput of the "Normal" harvest cluster and 61.2% of the "Dedicated" cluster. The frequent and significant CPU changes result in a higher cold start rate for the "Active" harvest cluster compared to "Normal" harvest cluster at similar loads as shown on the left side of Figure 4.16. The "Dedicated" cluster achieves 19% higher throughput than the "Normal" cluster because the small VMs in "Normal" are more prone to saturation at high loads. We also experiment deploying vanilla OpenWhisk on the "Active" and "Dedicated" clusters. Vanilla OpenWhisk only achieves 39.0% throughput on "Active" compared to "Dedicated" cluster. This demonstrates that MWS can better handle active resource variations. Even with the 26.9% performance loss for the worst case, we show in the next section that running serverless computing workloads on Harvest VMs significantly outperforms running them on regular VMs under the same cost budget.

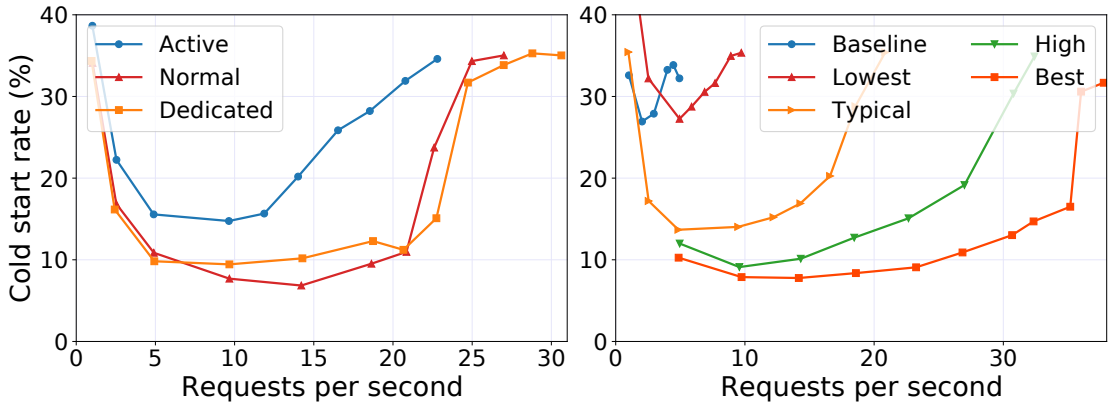


Figure 4.16: Cold start rate against load for fixed budget.

Discount	$d_{evict}(\%)$	$d_{harv}(\%)$	#VMs
Baseline (dedicated)	0	0	2
Lowest	48	48	6
Typical	70	80	12
High	80	90	18
Best	88	90	21

Table 4.3: Number of Harvest VMs with the same budget, based on the discount level.

4.7.4 Cost vs Performance

Cost. To evaluate the benefits of using harvested resources, we set a fixed budget and compare how many Harvest VMs we can provision and the load we can serve. As the budget baseline, we use two regular VMs with 16 CPUs and 64 GB of memory. We use the cost model introduced in Section 4.2 where the minimum resources and the harvested cores have a discount of d_{evict} and d_{harv} respectively. Table 4.3 shows the impact of the discounts. With the most pessimistic discount, we obtain 6 Harvest VMs, and up to 21 with an optimistic discount configuration.

Performance. Figure 4.17 compares the performance of each of these cluster configurations. These harvest clusters have 1.9×, 4.6×, 7.8× and 9.7× more CPUs than the baseline with 2 regular VMs. The throughput is 2.2×, 4.6×, 7.7× and 9.0× better as a

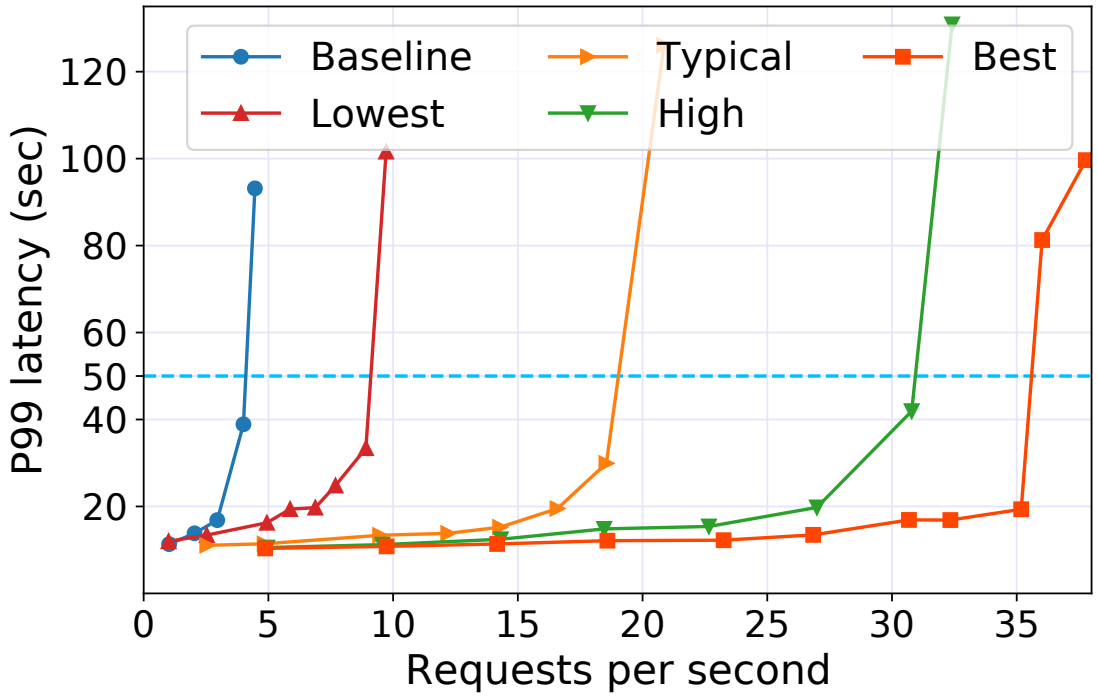


Figure 4.17: Regular vs Harvest VMs with same budget.

result of cheaper harvested CPUs.

Cold start rate. The improvement in throughput is also reflected with lower cold start rates for each load value as depicted in Figure 4.16 (right side). Notice that at very low loads, all clusters have high cold start rates since the function invocations are spread across many VMs but without significant impact on latency. As the load increases, the cold start rate initially decreases in all configurations, and then increases as load approaches system capacity (around 25% at saturation).

4.7.5 Harvest VMs vs Spot VMs

Harvest VMs and Spot VMs are both evictable VMs that leverage surplus resources. In this section, we compare hosting serverless workload on Harvest VMs and Spot VMs

via simulation, and focus on reliability and cost.

Experiment setup. For a fair comparison, we create synthetic Spot VM and Harvest VM traces with the idle resources of the same physical cluster (described in the characterization of resource variability in Section 4.3.1). For Harvest VMs, we place one VM on each node as long as the node can accommodate its base size, and the VM can harvest all idle resources on the node. For Spot VMs, we place as many as VMs as will fit on each node. Both Harvest VM and Spot VM are given a 30-second grace period before eviction. We use the same serverless workload trace as in Section 4.3.2, and pick the 5-day snapshot with aligning weekdays as the VM traces. We also extend the simulation framework in Section 4.4.1 to incorporate CPU usage: each invocation consumes one CPU and an invocation is buffered when the cluster runs out of CPUs. New containers are created on the VM with the least CPU utilization.

Sensitivity analysis. We analyze Harvest VMs with base size of 2, 4 and 8 CPUs (referred to as H2 to H8), and Spot VMs with size of 2, 4, 8, 16, 32 and 48 CPUs (referred to as S2 to S48), and the results are shown in Figure 4.18. $CPUs \times time$ is normalized against the idle $CPUs \times time$ of the physical cluster, and price is normalized against regular CPUs under the *Typical* configuration in Table 4.3.

Reliability. H2 achieves the lowest invocation failure with 4.31×10^{-6} (*i.e.*, "5 nines" of reliability). For Harvest VMs, the invocation failure rate increases with base size, reaching 3.54×10^{-5} at H8. For Spot VMs, invocation failure rate reaches its minimum of 1.00×10^{-4} at S2, but is significantly higher than Harvest VMs, being at least 23.2× higher than H2. The invocation failure rate on Spot VMs reaches the maximum at S16 and decreases with VM size afterwards. This is because the fragmentation caused by large VMs creates a larger buffer of unused resources that prevents VM eviction upon shrinkage of idle resources. Cold start rates show similar trends for the same reason.

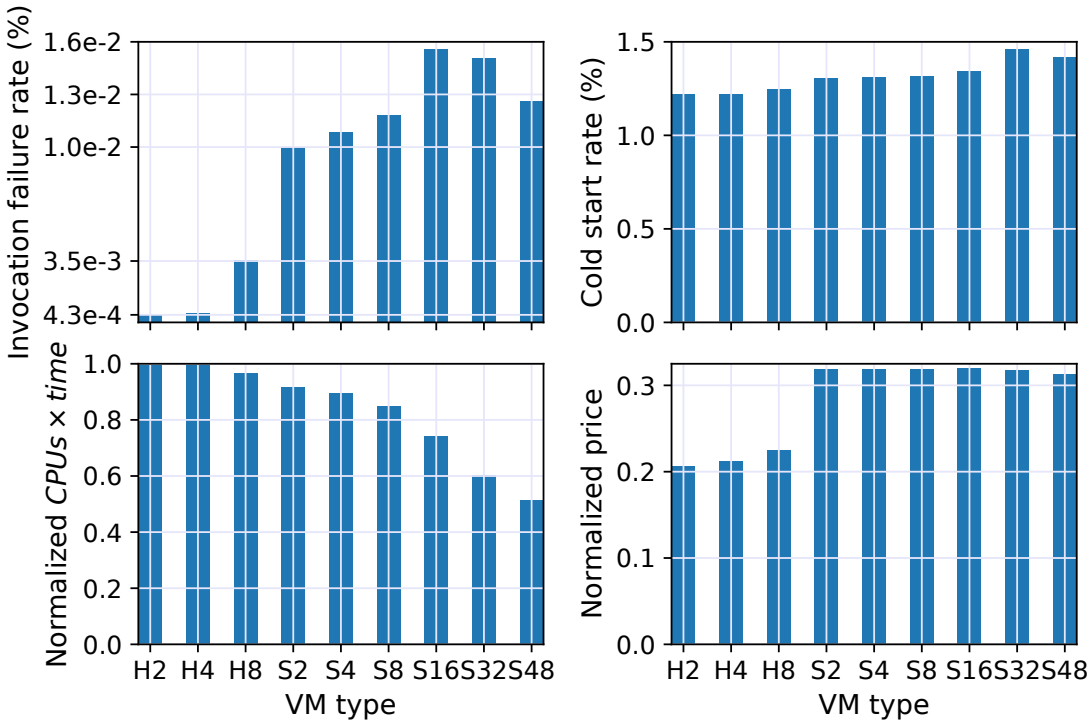


Figure 4.18: Harvest VMs vs Spot VMs. Hx refers to Harvest VMs with base size of x CPUs, and Sx refers to Spot VMs with x CPUs.

Cost. To calculate the price, we incorporate the additional per-VM cost incurred by the framework installation as in [36]. Assuming an installation time of 10 minutes as in Section 4.3.1, we use the following equation:

$$\frac{base\ core\ time \times d_{evict} + harvest\ core\ time \times d_{harv}}{base\ core\ time + harvest\ core\ time - install\ core\ time}$$

With the same idle resources, Harvest VMs also provide more effective compute power ($CPUs \times time$) than Spot VMs at cheaper prices. H2 can utilize 99.62% of the total idle compute power, and S2 can only utilize 91.67%. H2 offers an amortized per-CPU price of 0.211\$/*hour*, while the lowest per-CPU price of Spot VM is 0.313\$/*hour* (offered by S48). Harvest VMs are cheaper for two reasons: d_{harv} being smaller than d_{evict}, and less installation overhead as a result of less VM evictions. For Spot VMs, the effective compute power decreases with VM size as a result of fragmentation.

VM type	Base CPUs	Max CPUs	Memory
Harvest	2	6	16GB
Regular	8	8	32GB
Spot-4	4	4	16GB
Spot-48	48	48	192GB

Table 4.4: Characteristics of the Harvest VMs, regular VMs, and Spot VMs used in the experiment in §4.7.6.

4.7.6 Running on Real Harvest VMs

We now demonstrate executing snapshots of the function traces on real Harvest VMs. For this experiment, we cannot control the number of available CPUs and just report the organic numbers.

Experiment setup. To reproduce the invocations from the function trace, we use CPU-intensive loops with the same duration. Because the maximum number of concurrent running invocations in the function trace is too high to fit in the size of our cluster, we combine multiple 2-hour snapshots with fewer concurrent running invocations, making it feasible to replay the function trace.

Figure 4.19 reports the number of concurrent running invocations (the peak is 120 invocations), and we provision a cluster with 150 CPUs so that its CPU utilization is below 80%.

We test four clusters, consisting of Harvest VMs, baseline regular VMs, Spot-4 VMs and Spot-48 VMs (Table 4.4). We deploy MWS OpenWhisk on the Harvest VM and Spot VM cluster and the vanilla OpenWhisk on the regular VM cluster.

Utilization and performance. Figure 4.20 shows the total number of CPUs and the utilization for the Harvest, regular and Spot clusters. All clusters show similar CPU utilization patterns and the Harvest and Spot-48 clusters run all the functions with no

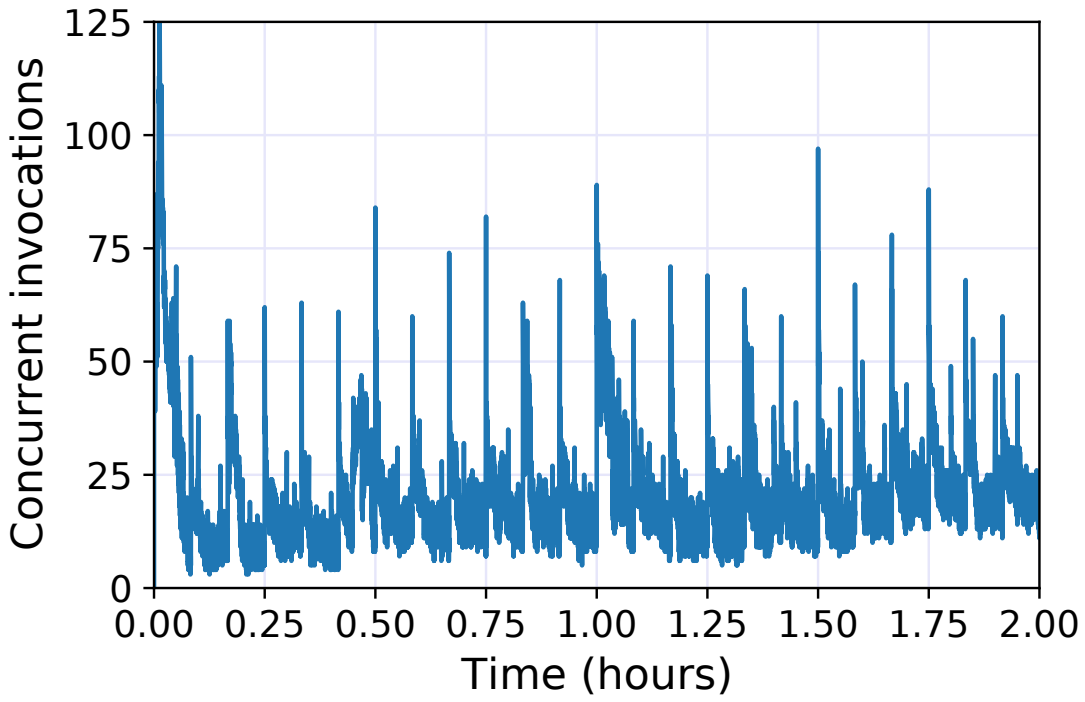


Figure 4.19: Invocations in the combined function trace.

Percentile	Harvest	Spot-4	Spot-48
25^{th}	56%	53%	53%
50^{th}	47%	43%	52%
75^{th}	32%	4%	38%
90^{th}	41%	15%	55%
95^{th}	74%	35%	83%
99^{th}	62%	16%	81%

Table 4.5: Latency reduction at multiple percentiles of Harvest and Spot VM clusters over regular VM clusters.

failure. Figure 4.21 shows the invocation latency distribution of the tested clusters. Table 4.5 lists the latency reduction of Harvest and Spot VM clusters over the regular VM cluster at different percentiles. Harvest VMs outperform other alternatives except Spot-48 VM, because large VMs are less likely to be saturated, both in terms of CPUs and memory. However, using large Spot VMs leads to lower resource utilization and higher failure rate, as discussed in Section 4.7.5.

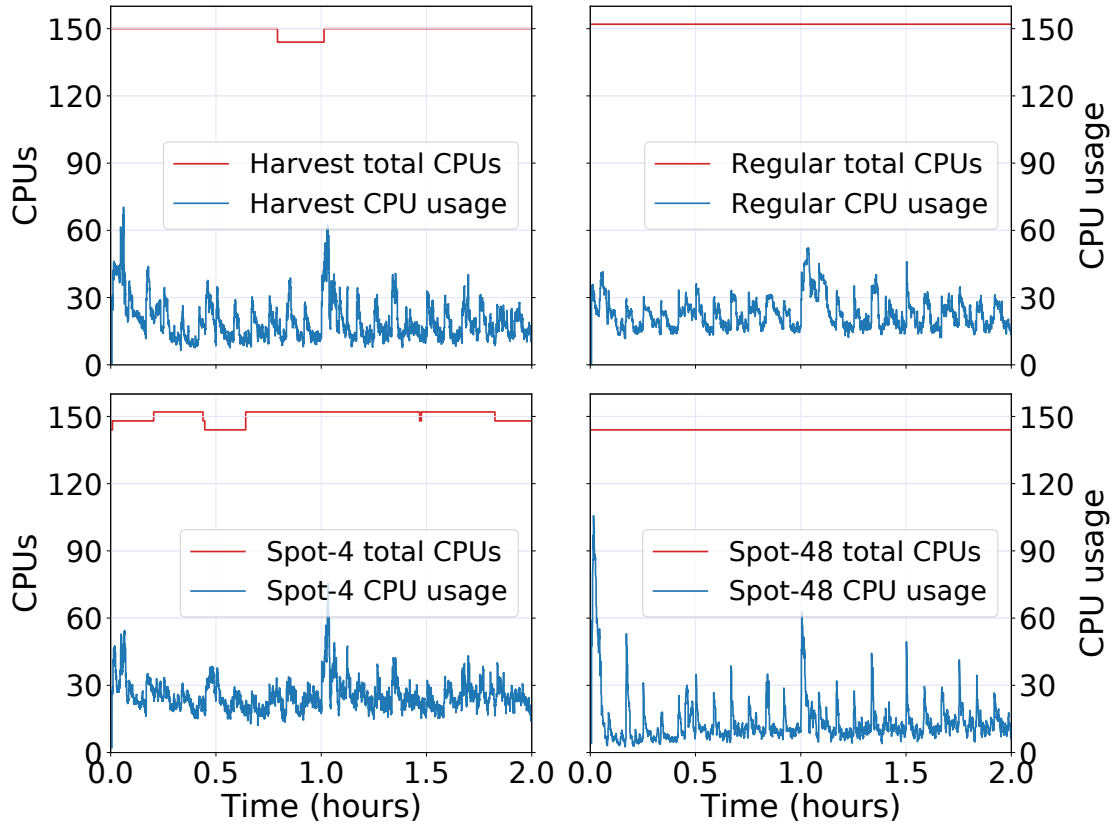


Figure 4.20: CPU number and cluster CPU utilization for Harvest VMs (upper left), regular VMs (upper right), and Spot VMs with 4 CPUs (lower left) and 48 CPUs (lower right).

Cost. We now compare the cost of Harvest VMs against regular VMs and Spot VMs, and we assume that the Spot VM cluster has the same configuration as the regular VM cluster. We analyze the four configurations of d_{evict} and d_{harv} from Table 4.3. Compared to regular VMs, Harvest VMs are 49%, 77%, 83% and 89% cheaper, respectively. Compared to their Spot-4 VMs counterparts, they are 0%, 22%, 45% and 11% cheaper, respectively. The worst case achieves no savings compared to Spot VMs because it pessimistically assumes harvested CPUs have the same price as evictable CPUs. This pricing is unlikely to happen in practice.

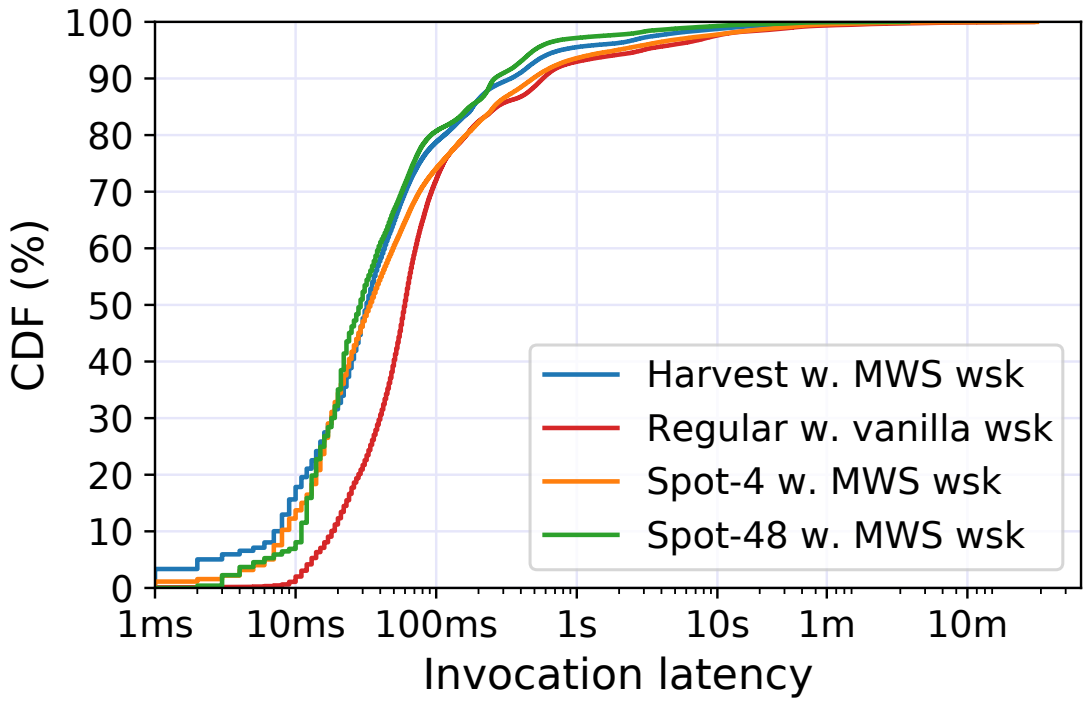


Figure 4.21: Response latency comparing MWS on harvested resources to vanilla OpenWhisk running on dedicated resources.

4.7.7 Summary

We demonstrate the performance benefit of MWS load balancing. It achieves 22.6× higher throughput than vanilla OpenWhisk, as it addresses resource variations. It also improves locality, resulting in lower cold start rates. With MWS, we realize the benefits of running serverless platforms on harvested resources, achieving lower cost and better performance: Under the same cost budget, running serverless platforms on harvested resources achieves 2.2× to 9.0× higher throughput compared to using dedicated resources; and with the same amount of provisioned resources, running serverless platforms on harvested resources achieves 48% to 89% cost savings, with lower latency due to better load balancing.

4.8 Conclusion

In this chapter, we propose to host serverless platforms on harvested resources. We quantify the challenges of using harvested resources for serverless invocations, including Harvest VM evictions and resource variation by characterizing the serverless workloads and Harvest VMs of Microsoft Azure. We demonstrate the reliability of hosting serverless workloads on harvested resources with trace-driven simulation. We also design and implement a harvesting-aware serverless load balancer on OpenWhisk, with which we demonstrate the performance and economic benefits of hosting serverless platforms on harvested resources.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Summary and Contributions

In this dissertation, we have presented three resource managers for cloud-native systems, including both microservices and serverless. In particular, we have made the following contributions:

1. **ML-based resource management of microservices:** In the realm of microservice resource management, the dependencies induced by inter-service communication, or backpressure effect, as well as the diverse resource requirements of individual microservices, pose significant challenges. Sinan demonstrates the effectiveness of Machine learning (ML) models in addressing these challenges. Specifically, Sinan (Chapter 2) shows that ML models can capture both the spatial and temporal correlations between per-service performance metrics, leading to accurate predictions of end-to-end performance and Service Level Agreement (SLA) violation status. The prediction can then be used to guide resource allocation decisions. Moreover, Sinan highlights the importance of a balanced dataset that includes an equal share of SLA violation and satisfaction, for the effectiveness of ML models. To create the balanced dataset, Sinan proposes the use of an exploration algorithm designed to trigger the corner cases of SLA violations.
2. **Lightweight models with better performance and minimal overhead:** Despite outperforming traditional approaches, ML-driven approaches require a lengthy exploration process that triggers to a high number of SLA violations, hindering their practical use. To reduce the complexity of modeling microservice perfor-

mance, we further investigate the backpressure-free conditions which guarantee that each service can considered independent for the purpose of resource management. With the backpressure-free conditions, we proposes Ursa, which uses an analytical model to decompose the end-to-end SLA into per-service SLAs, and maps them to resource allocations. Ursa outperforms ML-driven approaches with a more lightweight model while significantly shortening the exploration process and reducing the exploration-induced SLA violations. Ursa highlights the benefits of domain specific analytical models: compared to deep neural networks that can fit arbitrary function but requires large amounts of training data, analytical models designed with expert knowledge can be more performant, efficient and lightweight.

3. **Efficiently and safely hosting serverless with harvested resources:** In order reduce the infrastructure provisioning cost and deliver a better serverless product, we propose to host serverless platforms on harvested resources. We demonstrate the reliability of hosting serverless workloads against Harvest VM evictions, and design a harvesting-aware load balancer to handle Harvest VM resource variation while minimizing cold start. We also demonstrate the performance and economic benefits of Harvest VMs, and its superiority to other type of resource harvesting VMs.

5.2 Open Problems

Despite the research contributions presented in the thesis, there are still open problems regarding efficient resource management for cloud native systems.

Transparently reducing network stack overhead. Network stack contributes a

substantial portion of latency and computation to microservices and is often a bottleneck for increasing resource utilization. This is due to frequent interrupts and switches between user and kernel space that are required for network processing. While user space networking technologies such as DPDK [26] and RDMA [79] have shown promise in reducing network stack latency and boosting resource utilization for network-intensive applications, they require substantial modifications to the applications and present additional challenges in multi-tenant scenarios. Therefore, developing user space networking technologies that are compatible with the socket interface and can operate effectively in multi-tenant environments would represent a significant advance in the performance and resource utilization of microservices. This area requires further research and engineering efforts.

Resource management for blackbox services. In the thesis we assume a relatively transparent setting where operators have access to information such as microservice topologies, user loads, and service latency. However, in some cases, microservice topologies may be unknown, and real-time latency measurements may be difficult to obtain. Furthermore, in some situations, only resource utilization at the VM level may be available. Managing resources and maintaining SLAs in these blackbox scenarios can be particularly challenging, and further research is needed to develop effective strategies for resource allocation and SLA enforcement in these contexts.

Fast VM checkpointing and migration. Fast VM checkpointing and migration techniques are essential to eliminate invocation failures caused by VM evictions. Although such failures are rare in practice as demonstrated in the thesis, a theoretical guarantee on failure rate is lacking. To completely eliminate invocation failure caused by VM eviction, the culprit VM should be checkpointed and migrated within the eviction grace period. To achieve this, further research and engineering efforts are required

on lightweight VM technologies that can support fast checkpointing and migration. By developing such techniques, we can ensure that serverless workloads are always robust and reliable in the face of Harvest VM evictions.

5.3 Future Work

We believe our contributions open up several directions for future work

1. **Unified framework for microservices and serverless:** Microservices and serverless are similar and share much of the infrastructure. For the stateless part, the difference can be attributed to event-driven or RPC-connected long-running containers versus event-driven 'one-off' containers. There are trade-offs to be made between performance and cost when using both approaches, so a unified framework is needed that can automate these trade-offs and minimize cost while providing good performance. The framework should keep the programming model as simple as serverless and dynamically adapt the underlying implementation to serverless or microservices depending on load patterns and function execution times. For the stateful part, serverless frameworks typically offer fault tolerance by persisting intermediate results, and provide commit guarantees such as causally consistent commit, whereas microservices require user to implement such guarantees themselves. A unified framework should provide an interface for user to specify desired commit guarantees and provide default options for both serverless and microservice implementations.
2. **Hosting microservices with harvested resources:** Hosting microservices with Harvest VMs also has the potential of significant cost savings, but there are several challenges. Microservice containers are long-running, which makes them

more susceptible to failures caused by Harvest VM evictions. The orchestration framework should be able to predict the lifetime of Harvest VMs, and migrate or gracefully terminate microservice containers in advance. Furthermore, long running containers are also more susceptible to performance instability due to Harvest VM resource variation. The orchestration framework also needs to anticipate resource variation and adjusts load balancing decisions in advance to avoid performance degradation.

3. **Overcoming backpressure:** The backpressure effect in RPC-connected microservices increases the complexity of resource management and also limits resource utilization. Backpressure is mainly caused by queuing in the network stack when the system runs out of resources such as connections. Accelerated networks, particularly user-level networking techniques such as DPDK and RDMA, have the potential to help alleviate this phenomenon. The challenge, however, is to apply these techniques transparently to the microservices framework.

4. **Optimal resource allocation for serverless workflow:** We demonstrate the performance benefit of applying ML or analytical models to microservice resource management. The proposed techniques can also be applied to serverless workflows which also have topologies and are subject to SLA constraints.

BIBLIOGRAPHY

[1] Alibaba Cloud Container Service for Kubernetes. https://www.alibabacloud.com/product/kubernetes.

[2] Amazon Elastic Kubernetes Service. https://aws.amazon.com/eks/.

[3] Apache Kafka. https://kafka.apache.org/.

[4] Azure Kubernetes Service. https://azure.microsoft.com/en-us/products/kubernetes-service/#overview.

[5] Dapr: APIs for building portable and reliable microservices. https://dapr.io/.

[6] Decomposing twitter: Adventures in service-oriented architecture. https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture.

[7] Docker Container. https://www.docker.com/.

[8] FFmpeg: A complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org/.

[9] Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine.

[10] gRPC: A high performance, open source universal RPC framework. https://grpc.io/.

[11] Gurobi Optimization. https://www.gurobi.com/.

[12] Hugging Face. https://huggingface.co/.

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

[14] Locust: A modern load testing framework. https://locust.io/.

[15] OpenCV Face Recognition. https://opencv.org/.

[16] Prometheus. https://prometheus.io/.

[17] Redis. https://redis.io/.

[18] Redis streams tutorial. https://redis.io/docs/data-types/streams-tutorial/.

[19] Step and simple scaling policies for amazon ec2 auto scaling. https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-simple-step.html.

[20] The evolution of microservices. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, 2016.

[21] Advantages of Cloud Computing. https://cloud.google.com/learn/advantages-of-cloud-computing, 2023.

[22] Azure blob storage. https://azure.microsoft.com/en-us/products/storage/blobs, 2023.

[23] Azure: what is cloud native? https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/definition, 2023.

[24] Benefits of Cloud Migration. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/benefits-of-cloud-migration/#benefits, 2023.

[25] Containerd: an industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/, 2023.

[26] Data plane development kit (dpdk). https://www.dpdk.org/, 2023.

[27] Istio: simplify observability, traffic management, security, and policy with the leading service mesh. https://istio.io/latest/, 2023.

[28] Koordinator: QoS based scheduling system for hybrid workloads orchestration on Kubernetes. https://koordinator.sh/, 2023.

[29] Open Container Initiative. https://opencontainers.org/, 2023.

[30] Podman (Pod Manager tool). https://podman.io/, 2023.

[31] Zipkin. https://zipkin.io/, 2023.

[32] Microservices workshop: Why, what, and how to get there. http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference.

[33] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*, 2020.

[34] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In *USENIX ATC*, 2018.

[35] Amazon Web Services. AWS Lambda. https://aws.amazon.com/lambda/, 2021.

[36] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, and Ricardo Bianchini. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *OSDI*, 2020.

[37] Ansible. Ansible is Simple IT Automation. https://www.ansible.com/, 2023.

[38] AWS. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot, 2021.

[39] AWS. AWS Burstable performance instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html, 2021.

[40] Azure. Azure Burstable VMs. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable, 2021.

[41] Azure. Pricing - Linux Virtual Machines — Microsoft Azure . https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/, 2021.

[42] Azure. Use Azure Spot Virtual Machines. https://docs.microsoft.com/en-us/azure/virtual-machines/spot-vms, 2021.

[43] Microsoft Azure. Hyper-V Integration Services. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/integration-services.

[44] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, 2014.

[45] Maury Bramson, Yi Lu, and Balaji Prabhakar. Randomized Load Balancing with General Service Time Distributions. 2010.

[46] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, 2018.

[47] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[48] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120. ACM, 2019.

[49] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[50] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[51] Ka-Ho Chow, Umesh Deshpande, Sangeetha Seshadri, and Ling Liu. Deeprest: deep resource estimation for interactive microservices. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 181–198, 2022.

[52] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *NSDI*, 2005.

[53] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167. ACM, 2017.

[54] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[55] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *USENIX ATC*, 2015.

[56] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, USA, 2013.

[57] Christina Delimitrou and Christos Kozyrakis. Quality-of-Service-Aware Scheduling in Heterogeneous Datacenters with Paragon. In *IEEE Micro Special Issue on Top Picks from the Computer Architecture Conferences*. May/June 2014.

[58] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014.

[59] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[60] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS*, 2020.

[61] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys*, 2012.

[62] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *USENIX ATC*, 2019.

[63] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, 2017.

[64] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[65] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2023.

[66] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Yuan He, and Christina Delimitrou. 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*, pages 3–18. ACM, 2019.

[67] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*, 2011.

[68] John Gittins, Kevin Glazebrook, and Richard Weber. *Multi-armed bandit allocation indices*. John Wiley & Sons, 2011.

[69] Kristina Gligorić, Ashton Anderson, and Robert West. How constraints affect content: The case of twitter's switch from 140 to 280 characters. In *Twelfth International AAAI Conference on Web and Social Media*, 2018.

[70] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *OSDI*, 2016.

[71] Google. Google cloud functions. `https://google.com/functions/`, 2023.

[72] Varun Gupta, Mor Harchol Balter, Karl Sigman, and Ward Whitt. Analysis of

Join-the-Shortest-Queue Routing for Web Server Farms. *Performance Evaluation*, 64(9-12):1062–1081, 2007.

[73] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[74] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[75] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In *SoCC*, 2017.

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

[77] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *SOSP*, 2016.

[78] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Centralized Core-Granular Scheduling for Serverless Functions. In *SoCC*, 2019.

[79] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance {RDMA} systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 437–450, 2016.

[80] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *USENIX ATC*, 2015.

[81] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, 1997.

[82] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *CLOUD*, 2019.

[83] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *USENIX ATC*, 2018.

[84] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, 2018.

[85] Kubernetes. Kubernetes cpu management policy. `https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/`, 2023.

[86] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. AcM, 2010.

[87] Ailsa H Land and Alison G Doig. An automatic method for solving discrete programming problems. In *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer, 2010.

[88] Benjamin D Lee, Michael A Timony, and Pablo Ruiz. DNAvisualization.org: A Serverless Web Tool for DNA Sequence Visualization. *Nucleic acids research*, 47(W1):W20–W25, 2019.

[89] Linux. Cgroups. `https://www.kernel.org/doc/Documentation/cgroup-v2.txt`, 2023.

[90] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. Minneapolis, MN, 2014.

[91] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, 2015.

[92] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency

and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2023.

[93] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 355–369, 2022.

[94] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. {ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, 2022.

[95] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017.

[96] Llew Mason, Jonathan Baxter, Peter Bartlett, and Marcus Frean. Boosting algorithms as gradient descent. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 512–518, Cambridge, MA, USA, 1999. MIT Press.

[97] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 319–330, 2011.

[98] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.

[99] Microsoft Azure. Azure functions. `https://microsoft.com/en-us/services/functions/`, 2023.

[100] Microsoft Azure. Azure metadata service: Scheduled events for linux vms. `https://docs.microsoft.com/en-us/azure/virtual-machines/linux/scheduled-events`, 2023.

[101] MinIO. Minio - high performance, kubernetes native object storage. `https://min.io/`, 2023.

[102] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile Cold Starts for Scalable Serverless. In *HotCloud*, 2019.

[103] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: An Opportunistic Caching System for FaaS Platforms. In *EuroSys*.

[104] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2023.

[105] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 385–398, 2013.

[106] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *USENIX ATC*, 2018.

[107] OpenWhisk. Apache OpenWhisk Open Source Serverless Cloud Platform. `https://openwhisk.apache.org/`, 2023.

[108] OpenWhisk. OpenWhisk Pull Request 4611. `https://github.com/apache/openwhisk/pull/4611`, 2023.

[109] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of SOSP*. Farminton, PA, 2013.

[110] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *SOSP*, 2013.

[111] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A Kozuch, and Gregory R Ganger. 3sigma: distribution-based cluster scheduling for runtime uncertainty. In *EuroSys*, 2018.

[112] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI*, 2019.

[113] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Firm: An intelligent fine-grained resource management framework for slo-oriented microservices. *arXiv preprint arXiv:2008.08509*, 2020.

[114] Charles Reiss, Alexey Tumanov, Gregory Ganger, Randy Katz, and Michael Kozych. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of SOCC*. 2012.

[115] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[116] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144, 2016.

[117] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. *arXiv preprint arXiv:2104.13869*, 2023.

[118] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[119] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[120] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Communication of the ACM*, 2021.

[121] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of EuroSys*. Prague, Czech Republic, 2013.

[122] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[123] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of SOCC*. Cascais, Portugal, 2011.

[124] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 121–135, 2019.

[125] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *USENIX ATC*, 2020.

[126] Akshitha Sriraman and Thomas F Wenisch. μ suite: a benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[127] Akshitha Sriraman and Thomas F. Wenisch. µtune: Auto-tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, October 2018. USENIX Association.

[128] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 26–32, 2023.

[129] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. Distributed resource management across process boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 611–623. ACM, 2017.

[130] Apache thrift. `https://thrift.apache.org`.

[131] Tony Mauro. Adopting microservices at Netflix: Lessons for architectural design. `https://www.nginx.com/blog/microservicesat-netflix-architectural-best-practices/`.

[132] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*, 2016.

[133] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. *Benchmarking, Analysis, and Optimization of Serverless Function Snapshots*. 2021.

[134] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, 2013.

[135] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[136] Nikita Dmitrievna Vvedenskaya, Roland L'vovich Dobrushin, and Fridrikh Izrailevich Karpelevich. Queueing system with selection of the shortest of two queues: An asymptotic approach. *Problemy Peredachi Informatsii*, 32(1):20–34, 1996.

[137] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *USENIX ATC*, 2018.

[138] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, KK Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X Liu. Deepscaling: microservices autoscaling for stable cpu utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 16–30, 2022.

[139] Bernard L Welch. The generalization of 'student's'problem when several different population varlances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[140] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS operating systems review*, 35(5):230–243, 2001.

[141] Hailong Yang, Quan Chen, Moeiz Riaz, Zhongzhi Luan, Lingjia Tang, and Jason Mars. Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained cmp. In *Proceedings of the*

44th Annual International Symposium on Computer Architecture, ISCA '17, page 133–146, New York, NY, USA, 2017. Association for Computing Machinery.

[142] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181, 2023.

[143] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161. ACM, 2018.

[144] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.