Machine Learning Algorithms for Provisioning Cloud/Edge Applications

by

Constantine Ayimba

in partial fulfillment of the requirements for the degree of Doctor in

Telematic Engineering

Universidad Carlos III de Madrid

Advisor(s):

Vincenzo Mancuso

Paolo Casari

April 2022

Machine Learning Algorithms for Provisioning Cloud/Edge Applications

Prepared by:
Constantine Ayimba, IMDEA Networks Institute, Universidad Carlos III de Madrid
contact: constantine.ayimba@imdea.org

Under the advice of:
Vincenzo Mancuso, IMDEA Networks Institute
Telematic Engineering Department, Universidad Carlos III de Madrid

This work has been supported by:

Unless otherwise indicated, the content of is thesis is distributed under a
Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA).

To my late brother, Benjamin.

"Tis grace hath brought me safe thus far and grace will lead me home."
–John Newton, 1779

Acknowledgements

Firstly, I thank God for His providence that led me to IMDEA Networks and for the bountiful graces to persevere through seemingly insurmountable odds. I am also truly grateful to Dr. Vincenzo Mancuso and Dr. Paolo Casari, my supervisors. It was a good humoured conversation with them that set me on the path to this defining moment. This good humor persisted throughout and was a great support during the course of the PhD.

I thank Dr. Vincenzo Mancuso for his firm insistence on excellence coupled with good humor that inspired me to think of novel solutions to the research problems we were considering. His ability to elicit good ideas and refine them to their essential rudiments ensured that I avoided many pitfalls in pursuing unworkable approaches. His unyielding yet measured critiques resulted in important improvements of the algorithms presented in this thesis.

I would like to thank Dr. Casari in particular for his meticulous attention to detail in every aspect of the projects we undertook. His insightful questions, in our meetings, gave me a more profound understanding of the problems we were seeking to solve and inspired the creativity of the approaches we proposed and implemented. His watchful eye caught many misstatements in our papers and helped clarify them prior to submission and this went a long way in securing their eventual publication. I also appreciate his painstaking efforts in guiding me through draft versions and improving their look and feel.

I also extend my sincere gratitute to Gaetano Somma who took pains during his internship to implement our algorithms on a new testbed with docker containers. His untiring efforts to work over the Christmas break to meet the submission deadline was the epitome of dedication. My gratitude also goes to Dr. Michele Segata who guided me through my first steps in using state of the art simulators for vehicular networks. His help was indispensable in our foray into platooning and in correcting missteps in our simulations. Special thanks go to my IMDEA colleagues who were always generous with their time for consultations and even more crucially as a reminder of the grandeur of life aside from the PhD.

My heartfelt thanks go to my parish community, especially Jorge and Pilar, as well as my family for their untiring support throughout. Saving the best for last, my warmest gratitute to my fiancée María for bearing with me over these last gruelling months.

Published Content

The ideas and investigations of this thesis resulted in the following refereed publications:

[1] **C. Ayimba**, P. Casari and V. Mancuso. "Adaptive Resource Provisioning based on Application State". Published in *proc. 2019 International Conference on Computing, Networking and Communications (ICNC)*, February 18-21, 2019, Honolulu, Hawaii, U.S. https://doi.org/10.1109/ICCNC.2019.8685605

- This work is fully included and its content is reported in Chapter 3.
- The author participated in writing most of this paper, worked on the Discrete Time Markov Chain modelling of the Application Under Test and the scaling algorithm. Having set up the testbed, the author investigated the performance of the proposed scheme.

[2] **C. Ayimba**, P. Casari and V. Mancuso. "SQLR: Short-Term Memory Q-Learning for Elastic Provisioning". Published in *IEEE Transactions on Network and Service Management, vol. 18, no. 2, pp. 1850-1869*, June 2021. https://doi.org/10.1109/TNSM.2021.3075619

- This work is fully included and its content is reported in Chapter 4.
- The author participated in writing most of this paper, developed the Context Aware Q-Learning scaling algorithm in this work as well as the exploration/exploitation trade-off mechanism. Additionally, the author set up the test-bed and investigated the performance of the proposed scheme.

[3] G. Somma, **C. Ayimba**, P. Casari, S. P. Romano and V. Mancuso, "When Less is More: Core-Restricted Container Provisioning for Serverless Computing". Published in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2020* https://doi.org/10.1109/INFOCOMWKSHPS50562.2020.9162876

- This work is partially included in Chapter 5.
- The author participated in writing several parts of this paper and in designing the Context Aware Q-Learning scaling algorithm and testbed.

[4] **C. Ayimba**, M. Segata, P. Casari, and V. Mancuso, "Closer than close: Mec-Assisted Platooning with Intelligent Controller Migration". published in *proc. 24th Inter-national ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems.* November 22 - 26, 2021, Alicante, Spain. https://doi.org/10.1145/3479239.3485681

- This work is fully included and its content is reported in Chapter 6 and Chapter 7.
- The author participated in writing most of this paper, developed the Context Aware Q-Learning migration algorithm and configured the simulation frameworks to run in tandem to create coordinated simulations of the wireless network and the communicating vehicles.

[5] **C. Ayimba**, M. Segata, P. Casari, and V. Mancuso, "Driving under influence: Mec-Assisted Platooning". Under review in *Elsevier, Computer Communications*

- This work is fully included and its content is reported in Chapter 6 and Chapter 7.
- The author participated in writing most of this paper, developed the Context Aware Q-Learning migration algorithm, ancillary safety mechanism and configured the simulation frameworks to run in tandem to create coordinated simulations of the wireless network and the communicating vehicles.

Abstract

Reinforcement Learning (RL), in which an agent is trained to make the most favourable decisions in the long run, is an established technique in artificial intelligence. Its popularity has increased in the recent past, largely due to the development of deep neural networks spawning deep reinforcement learning algorithms such as Deep Q-Learning. The latter have been used to solve previously insurmountable problems, such as playing the famed game of "Go" that previous algorithms could not. Many such problems suffer the curse of dimensionality, in which the sheer number of possible states is so overwhelming that it is impractical to explore every possible option.

While these recent techniques have been successful, they may not be strictly necessary or practical for some applications such as cloud provisioning. In these situations, the action space is not as vast and workload data required to train such systems is not as widely shared, as it is considered commercialy sensitive by the Application Service Provider (ASP). Given that provisioning decisions evolve over time in sympathy to incident workloads, they fit into the sequential decision process problem that legacy RL was designed to solve. However because of the high correlation of time series data, states are not independent of each other and the legacy Markov Decision Processes (MDPs) have to be cleverly adapted to create robust provisioning algorithms.

As the first contribution of this thesis, we exploit the knowledge of both the application and configuration to create an adaptive provisioning system leveraging stationary Markov distributions. We then develop algorithms that, with neither application nor configuration knowledge, solve the underlying Markov Decision Process (MDP) to create provisioning systems. Our Q-Learning algorithms factor in the correlation between states and the consequent transitions between them to create provisioning systems that do not only adapt to workloads, but can also exploit similarities between them, thereby reducing the retraining overhead. Our algorithms also exhibit convergence in fewer learning steps given that we restructure the state and action spaces to avoid the curse of dimensionality without the need for the function approximation approach taken by deep Q-Learning systems.

A crucial use-case of future networks will be the support of low-latency applications involving highly mobile users. With these in mind, the European Telecommunications

Standards Institute (ETSI) has proposed the Multi-access Edge Computing (MEC) architecture, in which computing capabilities can be located close to the network edge, where the data is generated. Provisioning for such applications therefore entails migrating them to the most suitable location on the network edge as the users move. In this thesis, we also tackle this type of provisioning by considering vehicle platooning or Cooperative Adaptive Cruise Control (CACC) on the edge. We show that our Q-Learning algorithm can be adapted to minimize the number of migrations required to effectively run such an application on MEC hosts, which may also be subject to traffic from other competing applications.

Table of Contents

Acknowledgements vii

Published Content ix

Abstract xi

Table of Contents xiii

List of Tables xvii

List of Figures xix

List of Acronyms xxv

1. Introduction **1**
- 1.1. Enabling technologies in cloud provisioning 1
- 1.2. Provisioning as a sequential decision process 3
- 1.3. Design of Reinforcement Learning Agents 4
- 1.4. Contributions 5
- 1.5. Outline of the thesis 6

I Cloud Provisioning **7**

2. Background and Related Work **9**
- 2.1. Threshold setting 9
- 2.2. Reactive methods 10
- 2.3. Proactive methods 10

3. Provisioning for a delay tolerant application **13**
- 3.1. System response modelling 13
  - 3.1.1. System functions 14
  - 3.1.2. Large scale deployments 17

3.2. Experimental results . . . 17
3.2.1. Testbed setup . . . 18
3.2.2. System calibration . . . 19
3.2.3. Evaluation . . . 20
3.3. Summary and discussion . . . 22

4. Provisioning without application knowledge **23**
4.1. RL modeling and modified Q-Learning approximation . . . 24
4.1.1. Key idea . . . 24
4.1.2. System model for decision making . . . 25
4.1.3. RL short-memory decision agents . . . 26
4.1.4. Decoupled learning of agents . . . 27
4.1.5. Modified Q-learning approximation . . . 27
4.1.6. Exploration/exploitation tradeoff mechanism . . . 28
4.2. SQLR design . . . 30
4.2.1. Key idea . . . 30
4.2.2. Problem formalization . . . 30
4.2.3. Load Balancer . . . 33
4.2.4. Admission Control . . . 33
4.2.5. Scaling agent . . . 37
4.3. Experiments . . . 43
4.3.1. Testbed . . . 43
4.3.2. Implementation on large scale . . . 46
4.4. Results . . . 46
4.4.1. Admission control policy convergence . . . 46
4.4.2. Scaling agent's policy convergence and complexity . . . 48
4.4.3. Scaling profiles . . . 52
4.4.4. Service times . . . 57
4.4.5. CPU utilization . . . 60
4.4.6. Summary of results . . . 60
4.5. Discussion . . . 61

5. Container based provisioning **63**
5.1. Container provisioning . . . 64
5.2. Automatic provisioning system . . . 67
5.2.1. Admission controller . . . 67
5.2.2. Auto-scaler . . . 68
5.3. Experiment setup . . . 70
5.4. Results . . . 71
5.4.1. Docker experiments . . . 71

5.4.2. Kubernetes experiments . . . 73
5.5. Discussion . . . 74

II MEC Provisioning 77

6. Background and Related Work 79
6.1. Related work . . . 80
6.1.1. Controllers . . . 80
6.1.2. Service migration . . . 81

7. V2I platooning 83
7.1. Controller adaptations for V2I platooning . . . 84
7.1.1. Controller operation adaptations . . . 84
7.1.2. Latency compensation in the control law . . . 85
7.1.3. Slow down And spLiT (SALT) . . . 86
7.2. **Q**-learning agents for controller migration . . . 87
7.2.1. Data for state context definition . . . 88
7.2.2. Problem Formulation . . . 90
7.2.3. Q-learning . . . 91
7.2.4. Reward functions . . . 93
7.2.5. Asynchronous shared learning . . . 95
7.3. Performance evaluation . . . 97
7.3.1. Method . . . 97
7.3.2. Asynchronous shared learning extension . . . 100
7.3.3. Evaluation results . . . 101
7.4. Summary and discussion . . . 108

8. Conclusions 111

References 113

List of Tables

2.1. Summary of key related work 12

3.1. Transition probabilities of a busy VM Node 19

4.1. Key notation employed in the definition of SQLR 29
4.2. Summary of Results 61

7.1. Simulation Parameters 97
7.2. Parallel episodes until convergence 109

List of Figures

1.1. Schematic of Cloud provisioning 2
1.2. Contextualized Q-Table structure. 4

3.1. Block diagram of our transcoding service chain. The Admission Control (AC) ensures that requests are only admitted when there is sufficient capacity. The Load Optimizer (LO) ensures that the active Virtual Machines (VMs) are well utilised before new ones can be provisioned. . . 14
3.2. Relationship between CPU Utilization and concurrent transcoding processes in a VM. CPU usage is monitored by querying the number of ongoing transcoding requests as described in Section 3.2.2. 16
3.3. Schematic of large scale deployment. 17
3.4. Testbed setup. (1) Host PC (2) Client PCs (3) Gigabit per second (Gbps) capable switch. 18
3.5. Blocking probabilities under different values of γ. Between minute 10 and 20 the average traffic is 23 requests/min. Between minute 70 and 90 the average traffic is 20 requests/min. The rest of the time the average traffic is 8 requests/minute. $\eta = 0.8$. 20
3.6. Scaling actions by orchestrator. Given that only 3 VM nodes were available, scaling to a 4th VM node is included to show that if the capacity of the host had not been exceeded then there would have been another VM node added to the active set. $\eta = 0.8$. 20
3.7. CPU utilization. Between minute 10 and 20 the average traffic is 23 requests/min. Between minute 70 and 90 the average traffic is 20 requests/min. The rest of the time the average traffic is 8 requests/minute. Except for the “LB“ and “LO“ case VM_2 and VM_3 are shut down at certain periods, see Figure 3.6. $\eta = 0.8$. 21
3.8. Distribution of Service times for transcoding h264(1080p) to VP8(640x360). The other input and output formats mentioned in Section 3.2 show similar results. 21

4.1. Response time variation with load based on queuing theory results [49]. The variation is approximately linear just below the "knee" of the curve. If utilization levels remain below this value, response times are highly likely to be predictable and reliable. 31
4.2. Block diagram of training process. 32
4.3. SQLR block diagram. "LB" is the Load Balancer agent and "AC" is the Admission Control agent. 33
4.4. Influence of CPU utilization on service response times. 35
4.5. Q function table to train the AC. The gray area represents the ideal operating region at which resources are highly utilized and the service times are within Service Level Objectives (SLOs). The red-shaded area on the right represents the region where VM operation is likely to cause SLO violations. 36
4.6. SQLR action and state space. K is the current number of active VMs, N_Δ is the number of VMs that can be added and N_∇ is the number of VMs that can be removed. In general, $N_\Delta \neq N_\nabla$. The state space, whose parameters are prefixed by (*), comprises the number of active VMs and the quantized values of the average CPU utilization for the set of active VMs. 37
4.7. State space detail for a "card" in the action space. Each cell's index pair is given by the quantized level of average system-wide resource utilization in successive epochs. 39
4.8. Modified error function to estimate the blocking probability component of the initial Q values of card "0" (Figure 4.6) diagonals. 39
4.9. SQLR's horizontal scaling mechanism. We compare the Q-values of the grey-shaded cells in order to determine the best action according to Equation (4.6). Here, we choose a scale-out of +1 VM. After the scaling action a, the Q-value in the red cell receives the update as specified in Equation (4.4). One component of the update is the Q-value contained in the green cell of card "0" in bubble "$\{K+1\}$". 42
4.10. Testbed setup. (1) Dell T640 server: Hosts KVM hypervisor, VMs, Admission controllers and Scaling Agent. (2) Client PCs: Generate requests towards the server according to demand profile. (3) Gbps switch: Creates LAN between Clients and Server. 44
4.11. Pre-training workload profile. The red line is the moving average of the number of requests per minute, computed over windows of 30 samples. . . 45
4.12. Test workload profile. The red line is the moving average of the number of requests per minute, computed over windows of 30 samples. 45
4.13. Schematic of a modular large-scale deployment. (cf. Section 3.1.2.) 47

4.14. Admission Control training. Red curves: low utilization level between 30% and 45%. Blue curves: intermediate utilization levels between 45% and 53%. Cyan curves: high utilization levels of 60% and above. Dashed lines convey the Q-values of DROP decisions, solid lines of ADMIT decision. . 48
4.15. Scaling agent convergence behavior. 49
4.16. Cumulative Q-Values, i.e., the sum of Q-values for all states, taken at different snapshots in the course of the experiment. 51
4.17. Markov chain of possible actions from selected states. The numbers in curly brackets within each bubble, $\{\cdot\}$, point to the number of VMs. Right-pointing arrows from a state indicate scale-out, whereas left-pointing arrows indicate scale-in. The re-entrant arrows above each bubble indicate no scaling. 51
4.18. VM scaling for the EKF-based horizontal scaling scheme proposed in [25]. We represent resource savings with respect to static over-provisioning with 10 VMs via the gray-shaded area. 52
4.19. VM Scaling for RLPAS: the Q-Learning horizontal scaling scheme proposed in [22]. We represent resource savings with respect to static over-provisioning with 10 VMs via the gray-shaded area. 52
4.20. SQLR scaling behavior evolving with experience. For this training phase, we set $P_{\text{blk}} = 0.001$. 53
4.21. Blocking rates over two-minute intervals. Two SQLR configurations are shown: Case 1 ($\theta = 1, \beta = 0.01$) and Case 2 ($\theta = 10, \beta = 0.001$). For clarity, a moving average filter is applied with a window size of 30 samples. 55
4.22. Blocking rate distribution. Two SQLR configurations are shown: Case 1 ($\theta = 1, \beta = 0.01$) and Case 2 ($\theta = 10, \beta = 0.001$). 55
4.23. Service time distribution per job. Two SQLR configurations are shown: Case 1 ($\theta = 1, \beta = 0.01$) and Case 2 ($\theta = 10, \beta = 0.001$). The service time for each request is divided by the corresponding number of iterations it generates to obtain the time per job. 56
4.24. Moving averages of service times (taken over a window of 30 samples to smooth out switching overheads). Two SQLR configurations are shown: Case 1 ($\theta = 1, \beta = 0.01$) and Case 2 ($\theta = 10, \beta = 0.001$). 56
4.25. Soft Blocking Probability for the EKF scaler 58
4.26. Soft Blocking Probability for the RLPAS Scaler 58
4.27. Soft Blocking Probability for SQLR's Case 1 ($\theta = 1$, $\beta = 0.01$) 58
4.28. Soft Blocking Probability for SQLR's Case 2 ($\theta = 10$, $\beta = 0.001$) 58
4.29. Moving averages of CPU utilization (taken over a window of 10 samples to smooth out switching overheads). Two SQLR configurations are shown: Case 1 ($\theta = 1, \beta = 0.01$) and Case 2 ($\theta = 10, \beta = 0.001$). 59

5.1. Vertical vs. horizontal scaling: two containers running on distinct cores provide more predictable performance than one container running on two cores. 65
5.2. Proposed container auto-scaling architecture. 66
5.3. Relationship between CPU utilization and service time. When the reported utilization is $\leqslant 50\%$, the iteration time is predictable. 68
5.4. Short-Term Memory Q-Learning mechanism used for the auto-scaler, cf. Chapter 4. This schematic shows a scale out operation where in the epoch given by the interval $[t, t+1)$, action a increases the number of containers from N to $N+1$ and the state transitions from $s^{(t)}$ to $s^{(t+1)}$. 69
5.5. Traffic profile observed during the Docker experiments. Traffic rates are taken over two-minute windows. The red line is the moving average over 30 samples. 70
5.6. Scaling decisions taken by Auto-Scaler algorithm during the Docker experiment (using $\beta = 0.02$). 70
5.7. The blocking rate observed over the time during Docker experiments in terms of rejected requests per second. 72
5.8. Empirical CDF of the service time for Docker experiments. 72
5.9. Traffic profile observed during the Kubernetes experiments. Traffic rates are taken over two-minute windows. The red line is the moving average over 30 samples. 73
5.10. Scaling decisions taken by the AS algorithm during the Kubernetes experiment (with $\theta = 1.0$ and $\beta = 0.02$). 73
5.11. CDF of the service time for the Kubernetes experiments. 74
5.12. Blocking rate observed over time during Kubernetes experiments in terms of rejected requests per second. 75

7.1. V2I Platooning on the network edge 84
7.2. Controller delay compensation. At time t_i, car i generates the report packet with its own speed, acceleration and distance from vehicle $i-1$, and sends it to the MEC controller. Car $i+1$ does likewise at time t_{i+1}. At t_{ctrl}, the controller collates the data. 85
7.3. Platoon metrics as reported to the MEC Host. "PM" denotes a platoon member; d_λ is the required vehicle spacing; d_Δ is the spacing tolerance. . 88

7.4. Migrator State and Action Spaces. The relative migration delay and relative candidate MEC power tuple $\{\theta, \beta\}$, the change in processing time between successive epochs, T_Δ and the platoon topology given by the tuple $\{\Gamma^{(-)}, \Gamma^{(+)}\}$ constitute the state space. The actions "Remain in the current MEC host" and "Migrate to any MEC host characterized by $T_R < 1$" form the action space. The table $(\cdot)$ indicates migration options to MEC hosts with the same power as the current host; $(\uparrow)$ to more powerful MEC hosts; and $(\downarrow)$ to less powerful hosts than the current one. 93
7.5. The Q-learning update process of the migration agent. The gray cell represents the Q-value of the MEC currently hosting the controller. The red cell represents $Q(S, A)$, the Q-value of the MEC host chosen to host the controller from the context of the current host and platoon topology specified by $\{\Gamma_1^{(-)}, \Gamma_1^{(+)}\}$. The green cell represents $Q(S', a)$: the Q-value of the chosen MEC in the eventual context specified by the platoon topology $\{\Gamma_1^{(-)}, \Gamma_1^{(+)}\}$. The blue cells represent Q-values of the other alternative MEC hosts not chosen for the migration. 94
7.6. Simulation frameworks 96
7.7. Road and MEC Network for evaluating SALT. The scale corresponds to the road network, eNodeBs and the MEC host are exaggerated to make them discernible. 98
7.8. Road and MEC Network. The scale corresponds to the road network, eNodeBs and MEC hosts are exaggerated to make them discernible. . . . 99
7.9. MEC host load due to background traffic. 99
7.10. Spacing errors of platoon members (negative means farther). 101
7.11. Platoon effectiveness (spacing fairness $\times$ average speed) for different SALT triggers. 102
7.12. Filtered overall delay (interval between a vehicle sending packet to receiving a control directive from the controller) for the SALT scenario. 103
7.13. SALT performance for varying degrees of delay with $\Psi = 0.3$ 104
7.14. Sample Q-migration policies 105
7.15. Follow ME migration policy [74] 105
7.16. AUSP migration policy [85] 105
7.17. Speed profiles of the first follower. 107
7.18. Distribution of spacing between platoon members. Each box plot represents data taken over 20 s windows. The last box plot represents data in the last 10 s. 108

List of Acronyms

5G fifth-generation

AC Admission Control

ACC Adaptive Cruise Control

ASP Application Service Provider

AUSP Adaptive User-managed Service Placement

AWS Amazon Web Services

CACC Cooperative Adaptive Cruise Control

CAPEX capital expenditure

CFS Completely Fair Scheduler

CSP Cloud Service Provider

DTMC Discrete-Time Markov Chain

EKF Extended Kalman Filter

ETSI European Telecommunications Standards Institute

FaaS Function-as-a-Service

GCP Google Cloud Platform

HPA Horizontal Pod Autoscaler

IaaS Infrastructure-as-a-Service

ITS Intelligent Transportation Systems

LB load balancer

LO Load Optimizer

MDP Markov Decision Process

MEC Multi-access Edge Computing

NFV Network Function Virtualization

NUMA non-uniform memory allocation

OPEX OPerating EXpenditure

PaaS Platform-as-a-Service

QoS Quality of Experience

QoS Quality of Service

RL Reinforcement Learning

RLPAS Reinforcement Learning-based Proactive Auto-Scaler

SaaS Software-as-a-Service

SALT Slow down And spLiT

SDN Software Defined Networking

SFC Service Function Chain

SHA-256 256-bit Secure Hash Algorithm

SLA Service Level Agreement

SLO Service Level Objective

SQLR Short-term memory Q-Learning pRovisioning

SUMO Simulation of Urban MObility

V2I Vehicle-To-Infrastructure

VANET Vehicular Ad hoc NETwork

VeINS Vehicles in Network Simulation

VM Virtual Machine

1 Introduction

Cloud networks are increasingly becoming the preferred medium through which services are delivered to end users. Cloud Service Providers (CSPs) may offer Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) or Software-as-a-Service (SaaS) [6]. In IaaS, the tenants enjoy greater flexibility on deploying their own network management tools. In PaaS, tenants instead rely on the management tools supplied by the CSPs. In SaaS, all network resource management operations are deferred to the CSP and the end users simply consume the application offered. In many deployments, Application Service Providers (ASPs) are either IaaS or PaaS tenants of CSPs while they in turn offer their services on a SaaS basis to their own end users.

The main reasons for this is ASPs can save on capital expenditure (CAPEX) as they do not have to invest large sums of money on building up their own infrastructure to deliver their services. They also reduce OPerating EXpenditure (OPEX) given that the maintenance of these networks is not part of their running costs. An attractive feature of cloud networks is that they can scale to increasing demands given the substantial footprint of data center infrastructure that CSPs have.

However, cloud resources, though vast, are finite. Owing to this, provisioning (the process by which these resources are allocated to tenants of the CSPs) is a crucial part of cloud network operations. Prudent provisioning is indispensable if CSPs are to extract maximum return on investment by accommodating more ASPs without jeopardising the quality of service for their existing tenants or the quality of experience for the ASP end users. These objectives are often counter-posed, given that minimizing resource use of a given ASP increases the likelihood of hosted applications performing poorly. Such undesirable outcomes may result in considerable penalties for the CSP [7].

1.1. Enabling technologies in cloud provisioning

Elastic provisioning or the dynamic allocation of resources is therefore crucial to achieving these objectives. Resource virtualization technologies, such as Network Function

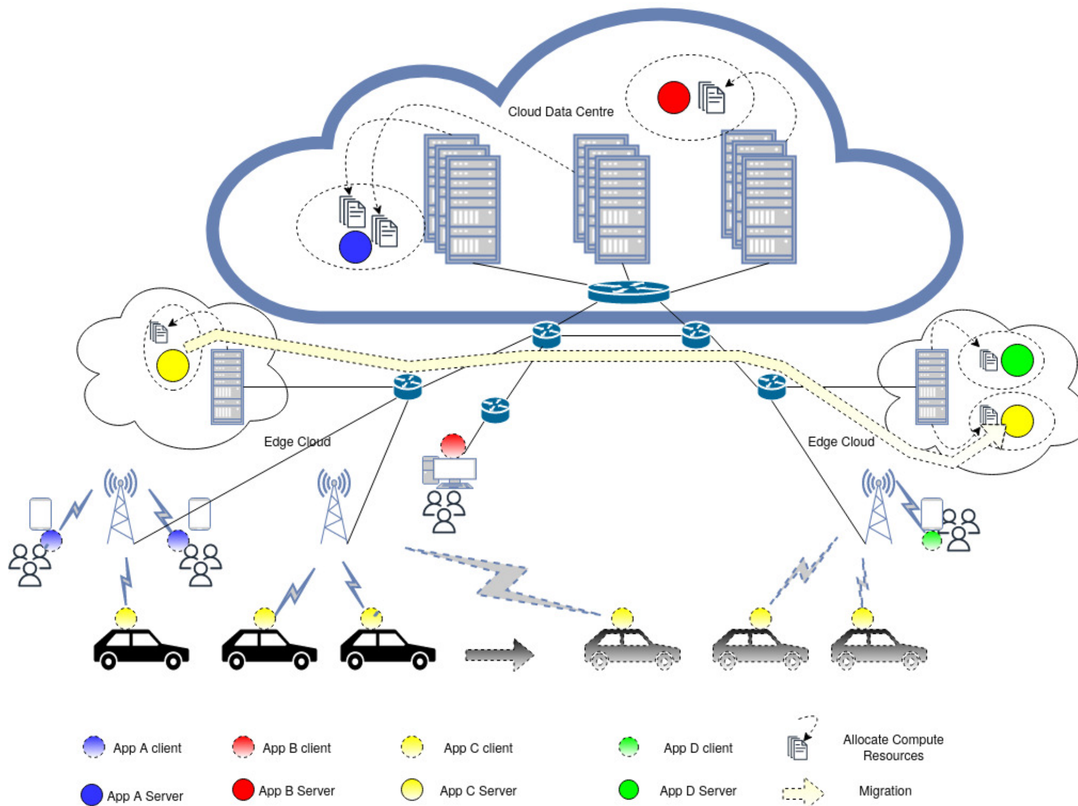


Figure 1.1: Schematic of Cloud provisioning

Virtualization (NFV) and Software Defined Networking (SDN), under-pinning cloud infrastructure make dynamic resource allocation possible. Extra resources can be brought online when demand increases and released when demand decreases. However, the latter is a non-trivial operation given that it requires fore-knowledge of future workloads to pro-actively service the dynamic demand.

Recently, the concept of edge clouds has also emerged as a compelling paradigm. In this scheme, latency sensitive applications are hosted in smaller data centres close to the user so as to curtail long round trip times. Taking user mobility into account, the challenge here is to ensure that the application keeps pace with the users to maintain the low latency connection. Provisioning in this case not only requires allocating resources but also migrating the application and its state to the most suitable edge location in the network.

An illustration of the various provisioning scenarios we consider in this thesis is shown in Figure 1.1. Application "A" here may require more cloud resources than Application "B", given that it is experiencing greater demand in comparison at the moment. Moreover, Application "C", which is hosted at the edge of the network with stringent latency requirements, has users with high mobility. As the users move into the coverage area of another edge cloud, it is expedient to migrate the application taking into account the

presence of Application "D" which is also competing for resources on the edge.

In the first part of this thesis we consider provisioning as scaling of cloud resources to match application demand. In the second part of the thesis we consider provisioning in edge cloud environments where application migration is indispensable as part of the provisioning process.

1.2. Provisioning as a sequential decision process

If the cloud application or service is known, it is possible to calibrate operation support systems such that appropriate thresholds can be used to trigger the addition or removal of network resources. This method enjoys wide usage in many commercial deployments given its simplicity but is limited in its applicability to diverse cloud applications. While it might be possible to predict demand by learning patterns based on vast amounts of historical data using deep neural networks, these models would have to be retrained to fit each cloud application with its own traffic pattern.

It is however quite feasible to exploit the time-series nature of demand to make sequential decisions on the evolution of allocated resources and subsequently evaluate their quality. In this way, should similar conditions be encountered in future, better decisions will be made based on previous experiences. This is achieved by treating the cloud infrastructure as a state machine given that the addition or removal of resources influences certain performance indicators which if properly chosen can define the state. The ability to learn online means that we can dispense with requirements for large datasets and the massive training overhead for each different application.

Markov Decision Process (MDP) can be used to describe scenarios involving sequential decisions. With the environment in a given state S, an agent takes an action A which results in the state transitioning to S' with a probability T and elicits a reward R. For cases where the environment is suitably bounded, with known transition probabilities, the environment can be suitably modelled as an MDP and dynamic programming methods or other such heuristics used to learn the best action to take when in a given state.

However, when the complexity of the underlying environment is such that an MDP is infeasible to obtain, Reinforcement Learning (RL) techniques can be employed which develop suitable action policies to apply in a given state [8]. In this thesis, we enhance one of these techniques, Q-Learning, to cover myriad applications and show that they outperform other state of the art approaches. We also present novel improvements that ensure that the agents have good convergence properties.

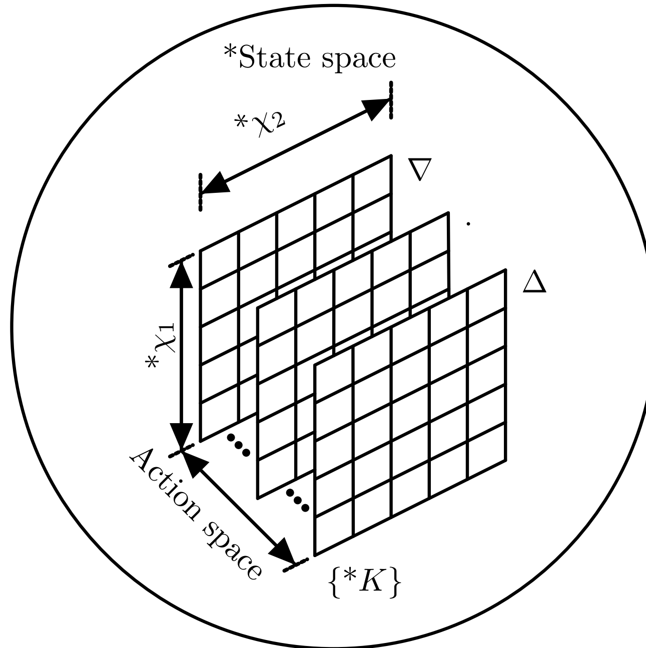


Figure 1.2: Contextualized Q-Table structure.

1.3. Design of Reinforcement Learning Agents

In this thesis, we use application performance indicators to define the state in designing the resource allocation agents. To handle the stochastic nature of the compute environment, the states are contextualized so that provisioning decisions take into account the evolution of application performance as shown in Figure 1.2. Discretising the state space and organising the consequent Q-tables in this way ameliorates the 'curse of dimensionality' [8] and makes it possible to learn on a tight data budget.

From the figure, χ_1 and χ_2 represent utilization metrics of the compute resource. These are usually quite noisy and are as such averaged between decision intervals. The element K represents the application related aspects of the state space e.g. how many resources are currently assigned to it or what its response is over the decision intervals. The ∇, Δ and $\cdot$ label the provisioning actions available to the agent. If it is a scaling agent the actions are: remove resources from the assigned pool, add resources to the assigned pool and maintain the resource pool as it currently is. If it is a migration agent, these actions are: move the application to a host of lower capacity, move the application to a host with a higher capacity and move the application to a host with similar capacity or maintain it on the current one. The most rewarding action settled on by a learned policy is dependent on the current overall state of the environment determined by the interaction of: resources assigned, the application response and incident workload.

1.4. Contributions

This thesis investigates machine learning techniques for adaptive provisioning with its main contributions appearing in 5 publications: 1 in *IEEE ICNC*, 1 in *IEEE Transactions on Network and Service Management* (indexed in Journal Citation Reports (JCR)), 1 article is under review in *Elsevier Computer Communications* (indexed in JCR), 1 in *IEEE INFOCOM* workshops and 1 in tier-1 conference *ACM MSWiM* as reported by CORE2021[1] or ERA2010[2]. Concretely:

Contribution 1: *Adaptive resource scaler based on Discrete Time Markov approximation*
Leveraging the highly correlated relationship between the workload and the resource utilization in defining the state, an adaptive provisioning system is developed. By profiling the transition probabilities between states, patterns emerge which we exploit to carry out adaptive resource scaling. We consider the case of transcoding video segments, a resource hungry, delay tolerant application. Further, since the application is delay tolerant, we design a load optimizer function that endeavours to extract the most feasible utility from compute resources before deploying others.

- **C. Ayimba**, P. Casari, and V. Mancuso, "Adaptive Resource Provisioning based on Application State". Published in *IEEE International Conference on Computing, Networking and Communications (ICNC)*, 2019, pp. 663–668.

Contribution 2 *Admission Control based on Q-Learning*
Given the stochastic nature of instantaneous demand, it is not always feasible to accommodate all incident requests to a cloud resident application with the resources on hand. Therefore before a scaling operation can be successfully completed, it is important to admit only those requests that can be reliably served using an Admission Control (AC) function. By using Q-Learning to determine the appropriate thresholds for resource use, an application and resource configuration agnostic admission control function is presented.
Contribution 3 *Short Term Memory Q-learning resource scaler*
Given that a universal application and workload model suitable for all cloud applications would be impractical, we present a model-free Q-Learning agent capable of scaling resources for myriad cloud applications with distinct workload patterns. It uses the variation of resources in successive temporal intervals contextualized by the amount of compute resources to define system state.

- G. Somma, **C. Ayimba**, P. Casari, S. P. Romano and V. Mancuso, "When Less is More: Core-Restricted Container Provisioning for Serverless Computing," Published in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2020, pp. 1153-1159.*

[1] CORE2021: http://portal.core.edu.au/conf-ranks
[2] ERA2010: http://www.conferenceranks.com/

- **C. Ayimba**, P. Casari, and V. Mancuso. SQLR: Short-term memory Q-learning for elastic provisioning. Published in *IEEE Transactions on Network and Service Management, Vol. 18, no. 2, pp. 1850–1869*

Contribution 4 *Context Aware Q-Learning edge application placement algorithm*
In edge computing scenarios, latency is a crucial consideration so that not only the amount of compute resources availed to an application but also where these resources are located on the network have to be considered. Taking platooning as a quintessential use case, we present a context aware migration agent based on Q-learning that considers the available edge resources and moves the platooning controller to the most suitable position on the network. Owing to the commercial and research interest in autonomous driving and the growing field of edge computing, we show how these two technologies can be fused to enhance coordinated driving and its attendant benefits.

- **C. Ayimba**, M. Segata, P. Casari, and V. Mancuso. 2021. "Closer than Close: MEC-Assisted Platooning with Intelligent Controller Migration," Published *In Proceedings of the 24th International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '21).*

- **C. Ayimba**, M. Segata, P. Casari, and V. Mancuso. "Driving Under Influence: MEC-Enabled Platooning," Under review *In Elsevier Computer Communications.*

1.5. Outline of the thesis

Each chapter of this work presents a cloud/edge application with unique provisioning challenges. In Part I of the thesis we present algorithms that dynamically adjust compute resources for cloud applications. Part II of the thesis considers the case of provisioning for Multi-access Edge Computing (MEC) applications with strict low latency requirements whose users are highly mobile.

In Chapter 3, we present transcoding as a delay tolerant cloud application. We exploit this property to develop a novel provisioning scheme which focuses on improving the utility of resources. Our scheme leverages admission control to create a Finite State Machine with workload dependent transition probabilities.

In Chapter 4, we consider more generic applications whereby service availability and reliability are key performance indicators. In this case, leveraging the evolution of utilization between scaling decisions, we present a novel short term memory Q-Learning provisioning scheme. Further in Chapter 5, we show how this scheme can be extended for use in serverless (container based) Function-as-a-Service (FaaS) environments.

In Part II, we present a Q-Learning based application migration algorithm which uses the spacing between vehicles in a platoon and the available cloud resource utilization as context to decide on the best location to host the controller application.

Part I

Cloud Provisioning

2 Background and Related Work

Legacy provisioning involved the dimensioning of resources to match peak loads. This over-provisioning was particularly prudent given that resources were mainly physical machines. The downside to this approach was that during off-peak periods, the excess resources became under-utilised. With the advent of virtualization and cloud computing, dynamic provisioning is now commonplace. The delicate balance between allocating sufficient resources for an application and avoiding wastage of said resources is an active field of research and innovation. In the following, I present some of the recent work aimed at striking this balance.

2.1. Threshold setting

One of the most pervasive methods in dynamic scaling involves setting a threshold on utilization, at which point additional resources can be brought online or released. A proportional thresholding policy that adapts to workloads so that a tenant can carry out horizontal scaling has been proposed in [9]. In [10] a control theory-based approach to threshold setting is proposed to counteract the instability caused by oscillations due to improper thresholds. In [11–13], fuzzy logic thresholds are used to ameliorate the oscillation problem.

Threshold setting is also quite popular in commercial offerings such as [14, 15] given its simplicity. The Kubernetes Horizontal Pod Autoscaler (HPA) for instance, obtains the utilization metrics of all containers in a given deployment, gets their mean and based on their ratio with the desired (configured) value adds or removes containers from the cluster. To avoid the oscillation problem, removals are only performed after a set timeout from the time they were triggered. Proper threshold setting however requires good knowledge of the application as well as knowledge of the CSP infrastructure configuration.

2.2. Reactive methods

Heuristic based methods such as [16, 17] rely on Monitoring, Analysis, Planning and Execution (MAPE) pipelines to make scaling decisions. These techniques extract instantaneous workloads and react to allocate sufficient resources to meet demands as specified in Service Level Agreements (SLAs). In particular, the authors of [16] focus on response time as a key metric which is challenging to guarantee given the complex distributed nature of cloud applications and unpredictable workloads. They propose a provisioning system that identifies bottlenecks in multi-tier web applications, in order to maintain reasonable response times. On the other hand, the costs for the Cloud Service Provider (CSP) are the primary focus of the scheme proposed by [17]. Their proposal is a workload model that exploits cloud heterogeneity assigning applications with complementary resource requirements to the same physical server. In this way, both high resource usage is achieved as well as reasonable application performance.

Other approaches further explore this idea. The authors of [18] for instance propose leveraging admission control to mix heterogeneous workloads so as to better utilize resources. Additionally, they apply scheduling so that the overall system achieves SLA enforcement by re-configuring and scaling resources as required depending on the demand. These methods require workload profiling and as such only react to demand which may not be sustained but short lived. They are thus quite prone to SLA violations.

2.3. Proactive methods

In order to address the shortfalls of reactive methods, recent research has focused on using Machine Learning techniques to predict demand. The approach presented in [19] employs a neural network trained on data from the TPC-W [20] benchmarking tool and a sliding window for CPU utilization predictions. These are then used to decide when scaling is required.

Assuming high predictability in arrival rates and system responses, the authors of [21], propose a theoretical, model based, Reinforcement Learning (RL) approach to cloud resource allocation which factors in both Service Level Objective (SLO) violations and net gains for the CSP. They propose finding the optimal policy by optimizing for the highest accumulated reward. They also carry out admission control to guarantee Quality of Service (QoS) for the Application Service Providers (ASPs).

In [22] an on-policy Reinforcement Learning-based Proactive Auto-Scaler (RLPAS) technique is proposed. This scheme uses single step temporal difference RL with multiple coordinating agents which communicate constantly to update the Q-value table. They also employ function approximation to optimize the value function. Their reward functions are based on application-specific targets for throughput and response times. They propose

adjusting both the number and type of Virtual Machines (VMs) i.e. hybrid horizontal and vertical scaling.

A vertical scaling agent based on Q-learning is proposed in [23]. Multiple RL agents re-apportion virtual CPUs, memory and bandwidth to active VMs hosting applications. Response times and throughput of the applications act as the basis for the reward functions that influence the policies learned by the agents. In order to accelerate learning, a the Q-table is stored in a Cerebellar Model Articulation Controller (CMAC) structure so that multiple state action pairs map on to a limited set of indices. The indices are then used to calculate the Q-values. This technique reduces the combined state and action space considerably. The authors of [24] propose an RL-based technique which causes the migration of VMs from servers with low utilization and subsequently shuts the servers down.

An Extended Kalman Filter (EKF) based scaler is proposed by the authors of [25] who employ a queue model as the observation model. A 3-tier cloud application with 3 request classes forms the basis of their measurement model. With these models response times are estimated according to the incident workload to trigger the corresponding scaling operations. In [26], a modified Q-Learning technique is used to adjust the resources of running VMs. Multiple cooperating agents, whose state space is a fuzzy logic combination of response times and utilization levels, are employed to expedite convergence by exploring different areas of the state space concurrently.

It may happen that in the intervening period between a sudden workload increase and a resource scale out operation, SLA violations occur. To address this issue, the authors of [27] use a Q-Learning scheme that allocates more resources than needed when the workload increases. Any superfluous resources are then gradually released as needed. In another recent work, a resource profiling system that predicts the near-term demand is proposed by the authors of [28]. This prediction is then used to adjust system resources appropriately trading off scaling costs and SLOs.

In [29], the authors consider a provisioning system that schedules resources in order to minimize costs incurred as a result of SLO violations and those arising from leasing cloud resources. A summary of key related proactive state of the art proposals is given in Table 2.1

Table 2.1: Summary of key related work

Reference	Parameters Considered	Scaling type	Technique used	Limitations
Rightscale [14]	Configurable triggers (e.g., memory, utilization, etc.)	Horizontal	Application dependent rule-based threshold setting	Requires knowledge of cloud application and underlying configuration.
Alsarhan et al. [21]	Theoretical: arrival rates, service times	Horizontal	Theoretical model-based reinforcement learning.	Dependent on workload profiling.
Benifa et al. [22]	Measured arrival rates, response times, throughput	Horizontal	Q-Learning with function approximation.	Too sensitive to transients in workload and CPU utilization.
Rao et al. [23]	CPU, memory and I/O utilization	Vertical	Model-free reinforcement learning with distributed agents.	VMs require autonomous control of host resources.
Gandhi et al. [25]	Measured arrival rates, response times	Vertical and Horizontal	Application modeling and extended Kalman filter (EKF).	Produces stiff scaling response, requires knowledge of expected response times.
Vasić et al. [30]	Measured arrival rate	Horizontal	Workload profiling, classification and pattern matching.	Unexpected workloads may cause erratic behavior.
Ibidunmoye et al. [26]	CPU utilization, response times	Vertical	Model-free reinforcement learning based on fuzzy logic with multiple agents.	Fuzzy state classification requires knowledge of cloud application and configuration.
Liu et al. [27]	CPU and memory utilization	Horizontal	Standard reinforcement learning with aggressive rewards for over-provisioning.	Prone to wasteful over-provisioning.
Fernandez et al. [28]	Measured arrival rates, CPU utilization and throughput	Horizontal	Threshold-based technique based on short-term capacity forecasts.	Vulnerable to over/under provisioning when faced with unpredictable workloads.
Xu et al. [31]	Pricing and availability of transient servers	Horizontal	Long-short term memory price prediction of transient servers.	Relies on short-lived virtual instances, possibly yielding inconsistent application performance.

3 Provisioning for a delay tolerant application

Many approaches in the literature seek to model the workload so as to determine the appropriate amount of resources to provision. We argue that when the cloud application is known, it is more apt to model how the resources respond to workloads, given that the resource capabilities and hence their responses are bounded. In this way resources can be better adapted to the workloads.

To illustrate the operation of the proposed provisioning scheme, we use transcoding (the conversion from one video format to another) as a case study of a cloud service. Transcoding is a resource-demanding process and is therefore a suitable application to be run as a service in the cloud. This is particularly important when the device on which the media is consumed has limited computational resources and power, as would be the case for a mobile phone. Given the use of buffers in streaming applications, some delay can be tolerated in provisioning for this service. We take advantage of this delay allowance in designing our provisioning scheme.

3.1. System response modelling

Adaptive provisioning can be considered a special case of sequential decision making [32]. An agent monitors the environment in order to obtain its state. It then decides on an action to take. Feedback from the environment, in the form of an immediate positive or negative reward signal or an eventual outcome, indicates how good or bad the action or a sequence of actions was. Such an action or sequence of actions results in a transition of the environment from one state to another.

If we let T be the set of decision epochs, S be the set of states the environment can be in, A_s be the set of actions that the agent can take when the environment is in state s, $p_t(\cdot|s,a)$ be the distribution of state changes given the state and action at the current decision epoch, following [33], the sequential decision process can be described by the set

$$\{T, S, A_s, p_t(\cdot|s,a)\}. \tag{3.1}$$

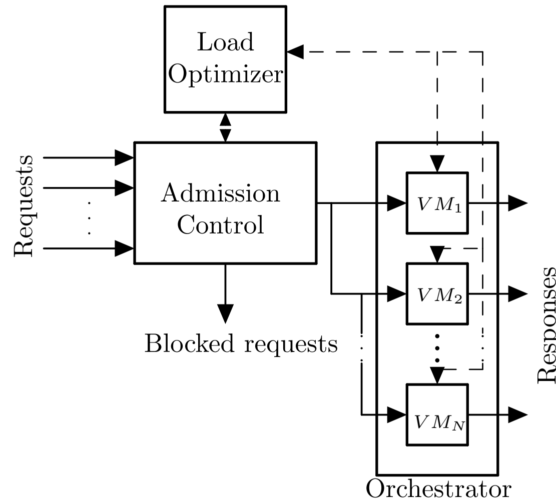


Figure 3.1: Block diagram of our transcoding service chain. The AC ensures that requests are only admitted when there is sufficient capacity. The LO ensures that the active VMs are well utilised before new ones can be provisioned.

3.1.1. System functions

Our system uses discrete time epochs to make scaling decisions based on the current state of the resource environment. A state refers to the number of requests being served concurrently by a Virtual Machine (VM) node and will be used interchangeably throughout this chapter. The actions to be taken are predefined based on an average system-wide value obtained by monitoring the set of active VM nodes for a finite time and counting the number of times each was servicing a specified number of requests.

A block diagram of the transcoding Service Function Chain (SFC) is shown in Figure 3.1. The admission control function carries out the instantaneous monitoring of individual nodes: it has the effect of converting each VM node into a finite-state machine, given that it conditionally admits requests based on the current state of the node, thereby limiting the number of possible states. An admitted request increases the state value of a VM node while a request that is fully served decreases the state value. Each of these events occurs with a probability that depends on the workload and computing power of the VM node respectively. These managed transitions can be modelled via a Discrete-Time Markov Chain (DTMC). Systemic environment monitoring is carried out by the Orchestrator function.

3.1.1.1. Orchestrator

The orchestrator takes action in response to perceived resource state as gleaned from polling the set of active VM Nodes. The actions that can be taken are:

- Scale out: Increase the number of active VM nodes to handle increased

demand.

- Scale in: Reduce the number of active VM nodes when demand is low.
- No action: If the current number of active nodes can handle the traffic satisfactorily.

Scaling Out: The orchestrator employs a provisioning algorithm that calculates the average number of concurrent requests being served by the entire set of active VM nodes using reports they send. These are obtained over a time window that is at least an order of magnitude longer than the service time of a single request, and are sent at intervals slightly shorter than the observation window. This setting preserves the memory of traffic events in previous epochs, and reduces the possibility of premature system scaling, which may result in oscillations (scaling in shortly after scaling out, or vice-versa) [10].

The average number of concurrent requests being served, K_σ can be calculated as

$$K_\sigma = \sum_{i=0}^{k} \left(i \sum_{j=1}^{N(t)} w_j(t) p_{ij}(t) \right) , \tag{3.2}$$

where k is the size of the state space, $N(t)$ is the number of active VM nodes in observation window t, $w_j(t)$ is the fraction of counts in node j w.r.t. the total counts from all active nodes taken over observation window t and $p_{ij}(t)$ is the probability a given VM node j is running i concurrent transcoding jobs over observation window t.

The scale-up condition compares the average number of concurrent requests the system is serving to the average calculated experimentally when the system is close to saturation. Specifically, the orchestrator checks if

$$K_\sigma \geq K_\Delta \ , \ K_\Delta = \frac{\sum_{i=0}^{k} i\pi_i}{\gamma} \ , \tag{3.3}$$

where K_Δ is the upper bound metric, γ is a tunable parameter, and π_i is the long-term probability of being in state i (the long-term probabilities, $\boldsymbol{\pi}$, are derived from the state transition matrix of a VM node operating under saturation conditions). If the condition in (3.3) is verified, a new VM node is added to the set of VM nodes.

The tradeoff between the blocking probability and resource utilization is not trivial. The choice of γ should be such that the orchestrator only adds to the set of VM nodes when the joint utilization of the current set is high enough. Similarly, γ should be chosen such that the orchestrator is responsive to perceptible changes in demand that may lead to reduced service availability, owing to an increase in the likelihood that a new request finds all active VM nodes too busy to admit it (blocking probability). In a scenario with known rates of arrival and departures the blocking probability can be computed using a queue model for instance, however we use observations to make the system more robust

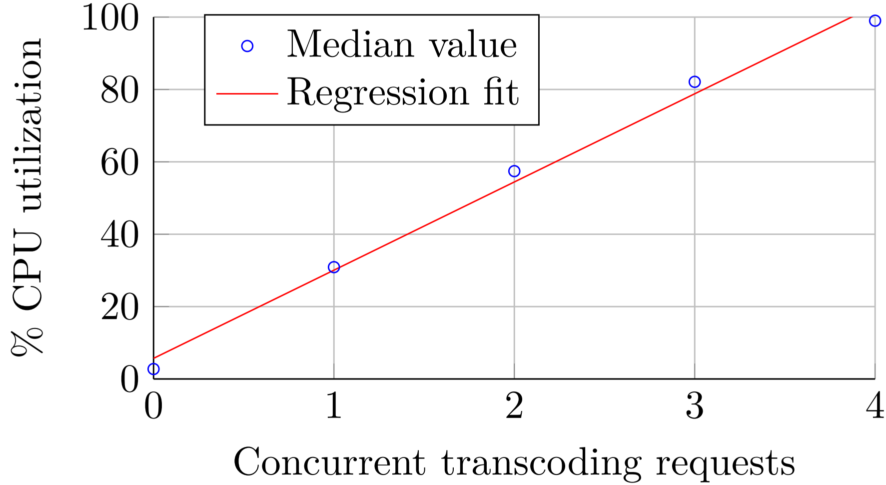


Figure 3.2: Relationship between CPU Utilization and concurrent transcoding processes in a VM. CPU usage is monitored by querying the number of ongoing transcoding requests as described in Section 3.2.2.

to dynamic demands.

Scaling In: A lower bound metric, $K_\bigtriangledown$ of system utilization also needs to be established, which specifies the lowest average number of concurrent processes that can justify the use of extra resources. It is fittingly chosen in relation to $K_\bigtriangleup$ as

$$K_\bigtriangledown = \eta K_\bigtriangleup \ , \quad 0 < \eta < 1 \ , \tag{3.4}$$

where η is a tuning parameter specifying the tolerance of the system.

Should K_σ become lower than $K_\bigtriangledown$, we determine which node should be shut down by computing the average number of concurrent processes, $K_\epsilon(n)$, for each node:

$$K_\epsilon(n) = \min_{1 \leq n \leq N} \sum_{i=0}^{k} i p_{in} \ , \tag{3.5}$$

where p_{in} is the probability of being in state i for VM node n, N is the total number of active VM nodes and k is the size of the state space.

No Action: If the state of the system is such that: $K_\bigtriangledown < K_\sigma < K_\bigtriangleup$, no scaling action is undertaken.

3.1.1.2. Load Optimizer and Admission Control

Our system includes a load-optimizer network function which informs the decision of the admission controller to admit or drop a request. By monitoring CPU utilization during the system calibration process described in Section 3.2.2, we obtain the relationship shown in Figure 3.2. Given the strong correlation depicted in Figure 3.2, the optimizer obtains

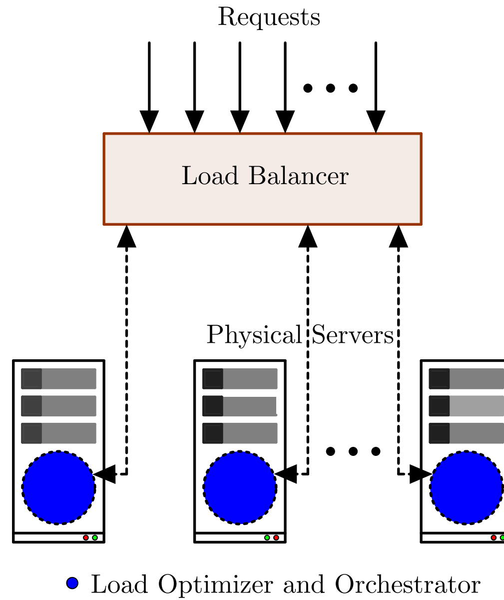


Figure 3.3: Schematic of large scale deployment.

the utilization level of each of the active VM nodes in turn by querying the number of concurrent requests in service. If the utilization of a VM node is such that a new request will not cause it to saturate, the optimizer directs the admission controller to admit the new request to the given VM node. If all the active VM nodes are saturated, the optimizer instructs the admission controller to drop the new request. Therefore, unlike a fair load balancer which channels traffic uniformly among nodes, the load optimizer only redirects requests to an alternative node when the ones under consideration are saturated or near saturation.

3.1.2. Large scale deployments

Though the experiments outlined in this chapter were carried out on a single server, large scale deployments can be handled in a modular fashion as shown in Figure 3.3. Here, a set of VNFs is employed in each physical server, and a standard load balancer mediates the channeling of the traffic to each server.

A simple round robin policy can be used for the standard load balancer given that the LO and AC, resident in each server, will ensure that these individual resources are not overloaded thereby ensuring that system as a whole is not saturated either.

3.2. Experimental results

In this section, we describe the operation of the transcoding experimental testbed and our scaling scheme. We first present the calibration of the VMs that are deployed to serve

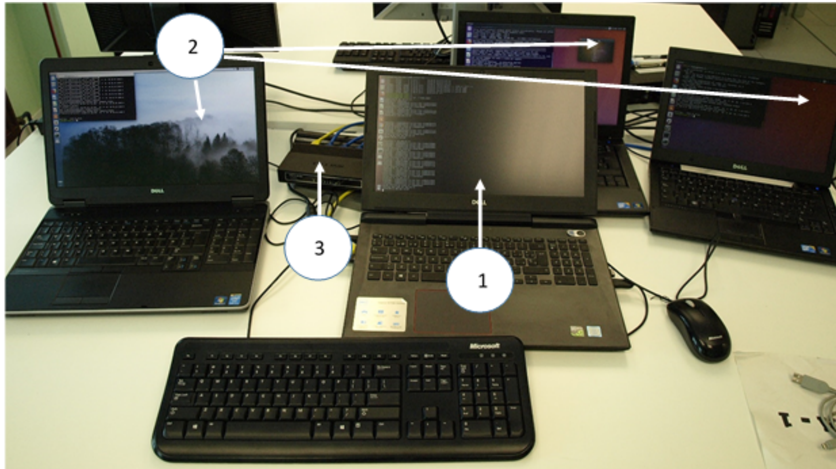


Figure 3.4: Testbed setup. (1) Host PC (2) Client PCs (3) Gigabit per second (Gbps) capable switch.

the transcoding requests. We then show how our scaler responds to dynamic demand by adapting the number of deployed VMs by leveraging the metrics obtained by calibration.

3.2.1. Testbed setup

We setup the experiment as shown in Figure 3.4. The Host PC has 4 hyperthreaded cores (8 logical CPUs) and 16 GB RAM. Each of the client PCs has at least 2 hyperthreaded cores (4 logical CPUs) and 8 GB RAM. The host uses KVM [34] as the hypervisor and libvirt [35] to manage the deployment of the VM nodes. Each VM Node is configured with 4vCPUs and 2GB of RAM.

The host launches VM nodes running G-Streamer [36] as the transcoder. It also runs ancillary Python and Shell scripts which handle the signalling and control aspects ensuring that each response is correctly mapped to the requesting process running on the client PCs. The Admission Control, Load Optimizer and the Orchestrator Network Functions are located in the host. All physical and virtual machines run on Ubuntu Linux 16.04 LTS.

We obtained the source video files from the official “Big Buck Bunny” repository [37] in 3 formats/resolutions: avi (720p), h264 (1080p), h264 (2160p). These formats represent mature standards widely adopted by content providers for video encoding [38]. We subsequently carried out scene selection and length splitting to create short video segments of 5 seconds for each file, using FFMPEG [39], without changing the source formats or other properties. This is common practice in adaptive bit rate streaming, whereby the same video content is encoded in different resolutions and file formats that support fragmentation. Each short fragment can be streamed interchangeably with others (bearing

Table 3.1: Transition probabilities of a busy VM Node

Source State \ Target State	0	1	2	3	4
0	0.259	0.441	0.218	0.076	0.006
1	0.025	0.345	0.409	0.209	0.012
2	0.004	0.068	0.415	0.467	0.046
3	0	0.018	0.186	0.791	0.005
4	0	0.002	0.030	0.248	0.720

the same content but of a different format/resolution) depending on available network bandwidth [40].

Each request specifies the source file to transcode from the three formats/resolutions. The format of the output stream is selected randomly by the control script as VP8 with either 320x180 or 640x360 resolution. The audio stream is transcoded from 448 kbps AC-3 format with 6 channels (5.1 surround) to 128 kbps mp3 audio with 2 channels (stereo). These formats were chosen for their popularity in video streaming [41]. For each request, the time taken to complete the simultaneous transcoding and streaming operation (as reported by G-Streamer) is logged with a timestamp indicating when it was received.

3.2.2. System calibration

In order to establish the upper-bound metric, K_Δ, the SFC was constrained to use only one VM node. This threshold was obtained by running transcoding requests at random intervals between 0 and 5 seconds of each other for a sustained period of 16 hours. This rate ensured that the VM node was operating close to saturation for the entire duration. The admission control was set to only admit new requests if CPU utilization was below saturation.

A monitoring script, running at 1 second intervals, keeps track of the process IDs of ongoing transcoding streams, in order to obtain the probabilities of transitioning from one state to another. The script checks the process IDs of running transcoder threads and compares the set of current IDs with the previous set. The intersection of the two sets indicates the number of transcoding processes that were ongoing in the system, the number of IDs present in the current set but absent in the previous set indicates the new requests. The number of IDs absent in the new but present in the previous set indicates completed streams. The total number of parallel transcodings in progress define the state.

Table 3.1 shows the transition probabilities obtained with our testbed. Using (3.3) and the stationary distribution derived from Table 3.1, we computed $K_\Delta = \frac{2.709}{\gamma}$. State occupancy reports were obtained over a duration of 5 minutes and sent to the orchestrator every 3 minutes to provide some filtering to spurious traffic events which may result in premature scaling [10].

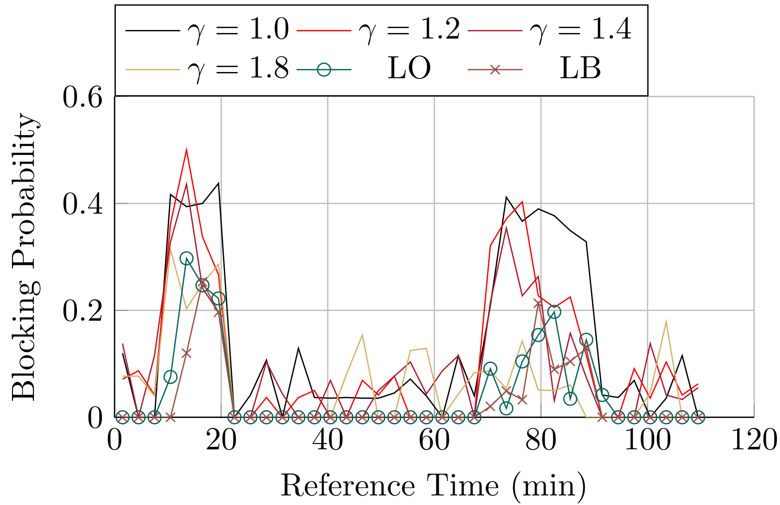


Figure 3.5: Blocking probabilities under different values of γ. Between minute 10 and 20 the average traffic is 23 requests/min. Between minute 70 and 90 the average traffic is 20 requests/min. The rest of the time the average traffic is 8 requests/minute. $\eta = 0.8$.

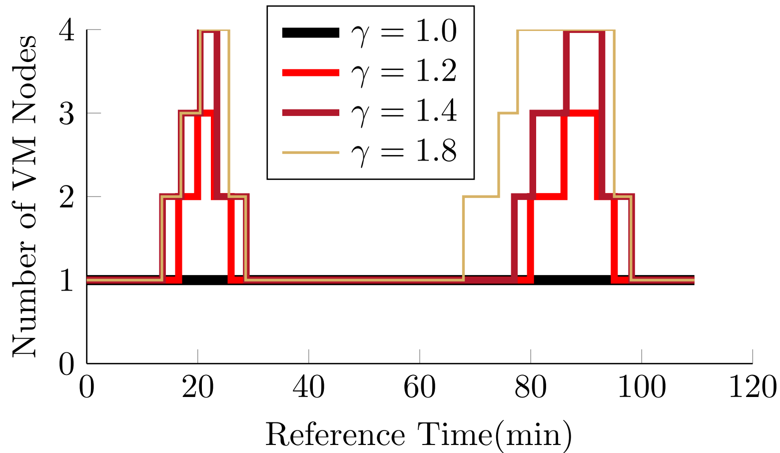


Figure 3.6: Scaling actions by orchestrator. Given that only 3 VM nodes were available, scaling to a 4th VM node is included to show that if the capacity of the host had not been exceeded then there would have been another VM node added to the active set. $\eta = 0.8$.

The blocking probability is calculated as the proportion of requests that are not admitted over the 3 minute epoch to the total number of requests received in that epoch.

3.2.3. Evaluation

In the referenced figures, “LO” refers to the case where only the Optimizer (with all three nodes active) is used and not the Orchestrator whilst “LB” refers to the case where Load Balancing (round robin) is applied with all three VM nodes active.

As depicted in Figure 3.5, a higher value of γ results in lower blocking probability as the SFC becomes more sensitive to smaller increases in traffic. When $\gamma = 1.0$ high

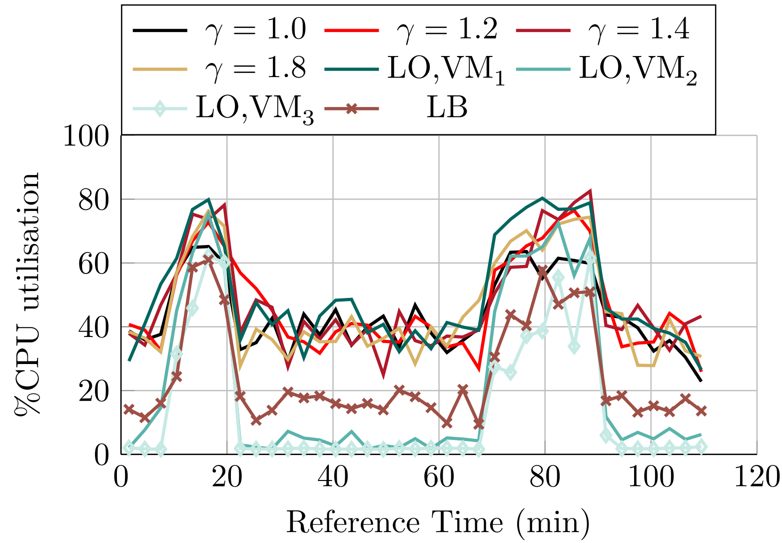


Figure 3.7: CPU utilization. Between minute 10 and 20 the average traffic is 23 requests/min. Between minute 70 and 90 the average traffic is 20 requests/min. The rest of the time the average traffic is 8 requests/minute. Except for the “LB“ and “LO“ case VM_2 and VM_3 are shut down at certain periods, see Figure 3.6. $\eta = 0.8$.

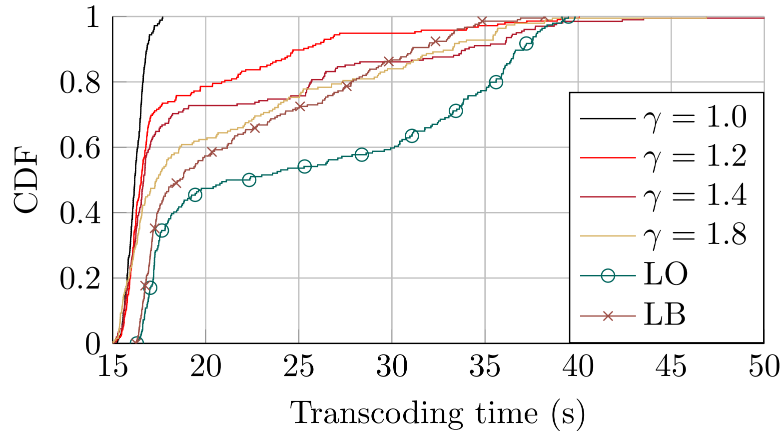


Figure 3.8: Distribution of Service times for transcoding h264(1080p) to VP8(640x360). The other input and output formats mentioned in Section 3.2 show similar results.

blocking probabilities are sustained for longer periods. If all the VM nodes are active and either “LB” or “LO” is employed, the lowest blocking probability is experienced as an alternate VM node is immediately available to handle the increase in demand.

Figure 3.6 shows the number of VM nodes dynamically provisioned by the orchestrator. A higher setting of γ increases the sensitivity of the orchestrator resulting in more scaling actions as the SFC adjusts to subtler changes in demand. When $\gamma = 1.0$, the orchestrator exhibits the smallest sensitivity and no scaling action is made leaving only the primary VM node to handle all the traffic.

Figure 3.7 shows the average CPU utilization of the primary node (VM_1) for different

values of γ, and that of all VMs for the two cases in which the orchestrator is not used. At low demand, when only the LO is used, VM_1 experiences most of the traffic while VM_2 and VM_3 are mostly idle. In the case where the orchestrator is used together with the optimizer, it shuts down the least busy VM nodes thereby eliminating superfluous resources. When load balancer (LB) is used, the levels of CPU utilization at low demand are about 50% less than those experienced when the load optimization is used together with the orchestrator.

Figure 3.8 shows the distribution of the service times of transcoding requests. The greater the number of active nodes, the worse the service time, given that more context switching [42] has to be performed (whereby the hypervisor has to stop, save state, handle interrupts and resume processing on a VM Node), which results in less CPU time for performing the transcoding. Context switching compounds when the hypervisor manages multiple VM nodes. The orchestrator ameliorates this effect by shutting down idle VM nodes resulting in improved performance.

The median transcoding time with $\gamma = 1.8$ (which provides a good compromise between service times and service availability) is about 1.3 s shorter than when round-robin LB is used. When extra VM nodes are not added, as is the case with $\gamma = 1.0$, the best response is observed.

From the results it is clear that a trade-off between system sensitivity to changes in demand and service performance has to be made. The settings for the orchestrator can be customized with reference to Service Level Agreement (SLA) terms agreed with tenants.

3.3. Summary and discussion

In this chapter, we presented a load optimizing and an orchestrator network function. When used together in a SFC, they improve the utility of VM nodes involved in servicing requests and limit the use of resources to those strictly necessary to meet SLAs.

We demonstrated that by learning the response of virtual machines handling a cloud service, decision bounds can be obtained for scaling up or down. We also showed that shutting down under-utilized nodes improves service times by reducing hypervisor overheads involved in managing multiple virtual machines.

The approach we presented in this chapter however does not scale well given the need for empirical calibration of system resources with regard to the cloud application. In the following chapter we address this limitation by presenting a more robust provisioning system.

4 Provisioning without application knowledge

Cloud applications vary widely in the services they offer, their acceptable Quality of Experience (QoS), their demand profiles among others [43]. Service Level Objective (SLO) violations can have grave consequences for an Application Service Provider (ASP), including loss of users and revenue [7]. Depending on the SLA contract signed, the ASP may even seek compensation from the Cloud Service Provider (CSP) for such violations [44]. For CSPs, this creates the need for dynamic provisioning/scaling tools. Such tools increase the resources allocated to an application when sudden increases in demand occur (or are foreseen), and release resources when they are no longer needed. The adaptations of allocated resources (*i*) save costs for ASPs and (*ii*) free capacity for other CSP tenants.

A significant number of state-of-the-art solutions to this problem guide scaling by continuously monitoring application and system metrics [45]. Other recent proposals addressing similar resource allocation problems, such as those presented in [46], leverage model-based machine learning. Though computationally efficient, they rely on extensive modelling and simulations, which may not always correspond to real-world demand and cloud application dynamics. Significantly different applications may require very different scaling configurations. Therefore, the challenge is to minimize the resources assigned to any application, while guaranteeing service quality in the face of variable demand.

To address this challenge, in this chapter we develop an application-agnostic Short-term memory Q-Learning pRovisioning (SQLR) system. Our scheme leverages two distinct and co-operating model-free Reinforcement Learning (RL) agents. The first agent uses conventional Q-Learning to make admission control decisions. Specifically, it learns the utilization threshold of compute resources that would make further task admissions inconvenient. The second RL agent uses a customized, context-aware Q-Learning algorithm to make resource scaling decisions when the system is exposed to dynamic and stochastic workloads. Since RL works by learning from experience rather than training on static data-sets, RL offers a practical and adaptive solution to the problem. Concretely, our main contributions are:

1. A configuration-agnostic admission control agent based on Q-Learning, that learns the most appropriate action to take given the level of resource utilization reported by a VM instance.

2. A flexible RL scaling agent that

    - is horizontal, i.e., it adapts resources in terms of the number of VMs allocated to a service, without resizing them by changing, e.g., the number of virtual CPUs, or the amount of memory allotted;
    - given high-level objectives, learns and enforces the best tradeoff between service availability and resource costs, even in the presence of challenging workloads;
    - quickly adapts to previously unexplored workloads by learning from *observed* resource utilization patterns rather than from the workloads *per se*;
    - progressively improves with every scaling decision, resulting in better service reliability and availability as time passes by.

3. A weighted fair learning mechanism to scale resources horizontally. This scheme encourages the exploration of new system states, while consistently exploiting better known states; this increases the likelihood of selecting near-optimal actions prior to completing the exploration of all possible states, i.e., much before reaching full policy convergence.

Our work is also relevant for modern cloud deployments such as Multi-access Edge Computing (MEC) for fifth-generation (5G) networks. Here, vertical scaling (which entails adjusting the capacity of running VMs) is not preferred given the constraints on MEC deployments, and horizontal scaling represents a more viable solution [47, 48].

The remainder of this chapter is organized as follows: in Section 4.1, we discuss the conventional Q-Learning algorithm and some ancillary modifications we made to it. In Section 4.2, we present the design of our system. In Section 4.3, we outline the experiments made to test the scaling and admission control algorithms. We present the results of our approach compared to other methods in Section 4.4. In Section 4.5, we reiterate the key contributions of this chapter.

4.1. RL modeling and modified Q-Learning approximation

4.1.1. Key idea

We assume a resource scaling system that allocates tasks to available VMs in a cloud computing environment, and that horizontally scales the number of VMs in face of time-varying workload. The scaling system is composed of three agents: a LB agent, an AC

agent, and a scaling agent. The LB assigns each incoming tasks to a VM, then the AC decides whether or not to admit the task to that VM. Periodically, the scaler oversees the utilization of active VMs and adapts their number to match workload requirements.

By design, our system is oblivious to the particular application that runs in the cloud: therefore, we only rely on system-level metrics (e.g., CPU utilization and time evolution thereof) in order to make admission and scaling decisions. We tackle the complexity of this scenario while keeping the system responsive to changes in stochastic workload patterns. To do so, we design our AC and scaling agents as sequential decision processes, where optimal decisions are identified thanks to a Q-learning approach.

In the following, we detail our system model for decision making (Section 4.1.2), discuss the admissible actions and state space of the AC and scaling agents (Section 4.1.3), explain our decoupled learning mechanisms (Section 4.1.4 and Section 4.1.5) and how we drive the exploration of new decisions against the exploitation of best decisions found up to a certain time (Section 4.1.6).

4.1.2. System model for decision making

Like in [21], we treat AC and horizontal scaling as sequential decisions, and assume that load balancing policies exist to evenly share workload among active VMs in the long run. Specifically, we consider a simple load balancer that dispatches incoming tasks to VMs by privileging the least utilized VM. We can then approximate the admission control and scaling processes as two separate classes of Markov Decision Processs (MDPs) with distinct sets of actions A, states S, state transition probabilities T and reward functions R.

We consider separate admission control and scaling decisions because: (*i*) the AC needs to operate at a much shorter time scale than resource scaling, and (*ii*) resource scaling cannot instantaneously obviate a lack of resources, because starting VMs up takes a non-negligible time. Moreover, the AC dropping rate for a given workload offered to a VM depends on the utilization of that VM and not on the number of deployed VMs. Therefore we can make AC decisions locally at each VM, in contrast with the necessarily global scope of scaling decisions.

We remark that treating our decision processes as MDPs requires a necessary approximation. The transition probabilities of a standard MDP are well defined: by way of contrast, our transition probabilities depend on the input workload, which is not necessarily stationary or known. Moreover, instantaneous fluctuations in the workload patterns are possible, which may lead to unexpected transitions. For such events, a plain memory-less decision process is inadequate [8, §17.3].

Yet, approximating a decision process through the MDP framework is still feasible, because we can track stochastic workload fluctuations by incorporating some memory of the past in the state definition, as will become clear later.

4.1.3. RL short-memory decision agents

Our system encompasses a load balancing component, a scaling component, and an AC component. We implement AC and scaling as reinforcement learning agents, whereas the load balancing component does not require learning in our setup.

A learning step, [1] or epoch, consists of the agent observing its state, taking some action allowed in that state, and monitoring the environment to compute and accrue some positive or negative reward as the environment transitions to a new state.

For an AC agent, the permissible actions are to either (*i*) pass an incoming request on to the VM that will serve the request, or (*ii*) refuse service to an incoming request. Either action results in a VM transitioning from one level of utilization to another with some probability. By defining the state as the utilization level of the VM handling the request, we are able to formulate this process as a per-VM reinforcement learning problem. We structure the reward values of the AC agent based on the predictability of job service times, which in turn relates to VM utilization level. For a given machine, the service time increases exponentially with its load [49]. The role of the AC agent is therefore to infer the next VM utilization level after an AC decision.

For the horizontal scaling agent, the actions are: (*i*) increasing, (*ii*) decreasing, or (*iii*) maintaining the same number of VMs. The state, in this case, is defined by three values:

1. the average system-wide utilization over the previous epoch;
2. the average system-wide utilization in the current epoch;
3. and the number of active VMs.

The action of the scaling agent changes this set of values, hence the system state. Therefore, unlike conventional RL, decisions depend not only on the current state, but also on the transition that led to the current state. This means that decisions are not memory-less. Rather, the agents have to embed some short-term memory in their decisions. To make this setting compatible with an MDP model, we define the system state such that it includes the average utilization level in the previous state, along with the previous number of active VMs. Thus, our modified RL algorithm retains a memory of the immediate past in the present state.

Thanks to this definition of state, our scaling agent exploits the rate of change in global average utilization levels, and can thus distinguish different workload patterns. This allows the agent to cumulatively learn different VM scaling policies for different workload profiles without overriding previously learned policies. We structure the reward function of the scaling agent to include both the blocking rate resulting from AC and the number of VMs used.

[1] Also called time-step in RL literature.

4.1.4. Decoupled learning of agents

Given that the AC decision is local to each VM, the AC agent must learn admission policies before the scaling agent can refine scaling policies. AC policies can be learned offline for a single VM, and do not need later refinements. Once AC policies have stabilized, the scaling agent can continuously learn its policies as new workloads are observed. For this purpose, the scaling agent simply needs to use a reward function that includes the blocking rate resulting from the use of the AC.

4.1.5. Modified Q-learning approximation

Since our RLs can be described by MDPs, the optimal AC and scaling decision policies could be determined by evaluating the corresponding MDPs. This entails tracking how much reward an action receives and obtaining the state transition probabilities that yield the highest accumulated reward given a particular system state. Then, we can program an agent via either value or policy iteration, in order to execute the resulting policies.

However, policy evaluations of the MDPs are impractical given that transition probabilities can vary widely depending on the workload and configuration of the system. A practical solution that applies to this case is Q-Learning [50]. Here, the agent develops a mapping of states to actions (known as the Q function) by tracking the accumulated reward (or "Q-value") for each state-action pair. With reference to Table 4.1, which summarizes the key notations used in this chapter, we now explain the design principles and behavior of the scaling and admission control agents. From [8], at learning step t, the optimal action-value function q^* is approximated as:

$$Q(S^{(t)}, A^{(t)}) \leftarrow \alpha R' + (1-\alpha) Q(S^{(t)}, A^{(t)}), \tag{4.1}$$

where

$$R' = R^{(t+1)} + \gamma \max_{\mathrm{a}} Q(S^{(t+1)}, a), \tag{4.2}$$

$Q(S^{(t)}, A^{(t)})$ is the action-value,[2] and $R^{(t+1)}$ is the immediate reward the agent receives after taking action a and ending up in state $S^{(t+1)}$, whose action-value is $Q(S^{(t+1)}, a)$. The factor γ anticipates the contribution of future rewards towards the immediate reward [50].

However, the use of a fixed learning rate α in Equation (4.1) assumes that all states are visited evenly during training [8]. Given the formulation of the state space, this may not always be the case for our AC and scaling agents. Further, the update process in Equation (4.1) typically leads to a stochastic policy, with Q function values oscillating slightly about an estimated expected value. To ameliorate this effect, we employ a modified reward mechanism, which takes into account the number of times that the agent visited the given state. This method follows closely the algorithm for the online

[2] In this chapter, we use the terms action-value and Q-value interchangeably.

computation of the mean:

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k}(X_j) = \frac{1}{k}\left(X_k + (k-1)\mu_{k-1}\right). \tag{4.3}$$

The Q function update then becomes:

$$Q(S^{(t)}, A^{(t)}) \leftarrow \frac{1}{k}\left[\Delta + (k-1)Q(S^{(t)}, A^{(t)})\right], \tag{4.4}$$

where $\Delta = R' - Q(S^{(t)}, A^{(t)})$, and k is the number of learning steps (prior to the current action) when the agent found itself in state $S^{(t)}$ and acted with action $A^{(t)}$.

Note that we modify the commonly adopted update mechanism by using the discounted reward Δ instead of the immediate reward R' in Equation (4.4). This reduces the chance of wrongly estimating the mean action value at the initial learning phases. The update equation in Equation (4.4) also guarantees that the policy will eventually converge, since the update value on the right-hand side of Equation (4.4) becomes progressively smaller as the number of learning steps increases.

4.1.6. Exploration/exploitation tradeoff mechanism

For both AC and resource scaling, we train the agents by initially encouraging random actions (exploration phase). As the agent develops a policy, it progressively acts less randomly, i.e., it chooses those actions that are known to yield the highest reward (exploitation phase). We accomplish this by employing ϵ-greedy action selection [8]. In this scheme, the agent selects the action that yields the highest reward with probability $1 - \epsilon(s)$, and a random action with probability $\epsilon(s)$.

We remark that the agent does not visit all states with the same frequency. Therefore, a global assignment and decrease of ϵ may bias the learned policy towards the most visited states. To avoid this, we employ a scheme that reduces ϵ independently for each state, proportional to the number of times $i(s)$ that state s is visited. This accelerates the learning process by encouraging exploration for the least visited states, while exploiting optimal actions for the most visited states. Specifically, we set:

$$\epsilon(s) = \begin{cases} 1 - \dfrac{i(s)}{M}, & \text{if } i(s) < M \\ \epsilon_{\min}, & \text{if } i(s) \geqslant M, \end{cases} \tag{4.5}$$

where M is a design parameter representing the number of statistically significant visits that should result in convergence to a stable policy. We consider that state s has achieved convergence when $\epsilon(s)$ equals $\epsilon_{\min} \geqslant 0$. For clarity, in what follow we drop the dependence of ϵ and i on the state s.

In order for the system to perform satisfactorily prior to convergence, we devise

Table 4.1: Key notation employed in the definition of SQLR

Variable	Meaning	Description
$Q(S^{(t)},A^{(t)})$	Action-value	The value of the Q function at time t.
$R^{(t+1)}$	Immediate reward	The reward the agent receives after taking action a and ending up in state $S^{(t+1)}$.
α	Learning rate	A fraction that modifies the reward update and influences the speed of convergence.
ϵ	Randomness factor	The probability of selecting an exploratory action prior to convergence.
$\epsilon_{\min}$	Minimum randomness factor	The minimum probability of selecting an off-policy action after convergence. We set this at 0.
k	State visits/action counter	The state- and action-dependent number of times the system was in state s and selected action a.
M	Visits to state s after which $\epsilon(s) = \epsilon_{\min}$	A statistically significant number of visits to achieve a stable policy for a given state. For the AC, we set $M = 1000$. For the scaling agent, we set it at ten times the number of actions allowed in that state.
γ	Discount rate $\in (0, 1]$	Expresses the current value of a future reward due to the present action. We set $\gamma = 0.8$.
x_{bnd}	Utilization upper bound	The utilization level above which response times become unpredictable [49]. We set $x_{\text{bnd}} = 60\%$.
x_n	Highest quantized utilization	The quantized utilization level closest to x_{bnd} used in the AC policy (see Fig. 4.5). We set $n = 3$ in (4.9).
$x_{\lim}$	Utilization admission limit	The practical limit of resource utilization obtained after training the AC.
θ	Resource cost modifier	Multiplier that weighs the cost of deploying resources in the reward function.
β	Blocking probability modifier	Multiplier that weighs the blocking rate in the reward function.
P_{blk}	Target blocking probability	We set $P_{\text{blk}} = 0.001$, corresponding to service availability of 99.9%.
$R_{\min}$	Minimum reward	Small, positive reward for maintaining the blocking probability lower than P_{blk}. We set $R_{\min} = 0.001$.

a *weighted fair guided exploration scheme*. Consider learning instance i. If the most rewarding action is not chosen (which occurs with probability ϵ) the conditional probability $P_a^{(i)}$ of selecting any of the L possible actions depends on its present action-value $Q^{(i)}(s, a)$, and on the number of times $k^{(i)}$ that action a has previously been selected when the system was in state s:

$$P_a^{(i)} = \begin{cases} \dfrac{1}{L}, & \text{for } \Psi_a^{(i)} = 0 \\ \dfrac{\Psi_a^{(i)}(1 - \tanh \phi_a^{(i)})}{\sum_{j=1}^{L} \Psi_j^{(i)}(1 - \tanh \phi_j^{(i)})}, & \text{for } \Psi_a^{(i)} > 0 \end{cases} \tag{4.6}$$

where

$$\begin{aligned} \Psi_a^{(i)} &= Q^{(i)}(s, a) + \sum_{j=1}^{L} \left|Q_j^{(i)}(s, a)\right|, \\ \phi_a^{(i)} &= \frac{k^{(i)}}{i}. \end{aligned} \tag{4.7}$$

Note that, in Equation (4.6), $\Psi^{(i)} \geqslant 0$ is used in place of the action value $Q^{(i)}(s,a)$ which, if negative, would result in unfeasible probabilities. We choose the hyperbolic tangent as a weighting function since $0 \leqslant \tanh(\phi) \leqslant 1$ for $\phi \geqslant 0$.

The above strategy achieves a tradeoff between exploration and exploitation, and curtails the detrimental effects of unguided exploration on performance.

4.2. SQLR design

4.2.1. Key idea

In Section 4.1, we have described a system that performs load balancing, admission control and horizontal scaling. The load balancer is simple and fair to active VMs, and optimizes resource utilization while minimizing AC blocking events [51]. The AC agent runs based on a Quality of Service (QoS) consideration: accepted tasks should receive a reasonably predictable service time. Another learning agent runs the third and most central component of our algorithm: horizontal resource scaling. This agent decides based on a cost-benefit tradeoff: utilize the least number of VMs so that the AC blocking rate be under a target threshold. With the above, we can formalize an optimization problem and design a framework to dynamically optimize the resources allotted to service a workload, as shown hereon.

4.2.2. Problem formalization

With the learning model described in Section 4.1, the resource adaptation problem we tackle in this chapter can be stated as follows: *Maximize the number of served tasks by horizontally scaling the number of VMs as workload evolves over time. Minimize the number of instantiated VMs that run a given service, under the constraint that the probability to block a service request remains below a predefined threshold.* Formally, let $X_{ji}(t) = 1$ if task j arrives at time t and is assigned to VM i, and $X_{ji}(t) = 0$ otherwise. Also, let $Y_{ji}(t) = 1$ if task j is running on VM i at time t, and 0 otherwise. Call $V(t)$ the number of VMs activated at time t, $V_{\max}$ the maximum number of VMs reserved for an ASP, $A(t)$ the set of tasks arriving at time t, and $\mathcal{J}(t)$ the set of tasks to be served at time t. Let P_{blk} be the ideal blocking probability set out in the SLA, and ρ_j be the contribution of task j to the utilization level of a given VM. Finally, call x_{lim} the utilization level above which response times become unpredictable.

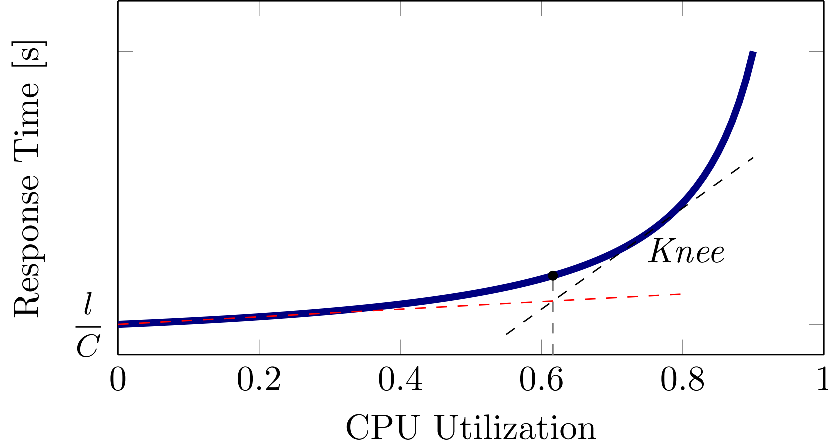


Figure 4.1: Response time variation with load based on queuing theory results [49]. The variation is approximately linear just below the “knee” of the curve. If utilization levels remain below this value, response times are highly likely to be predictable and reliable.

We can express our problem formally as:

$$\min \frac{1}{T} \int_0^T V(t)\, \mathrm{d}t \tag{4.8a}$$

$$\text{s.t.:} \; \frac{\int_0^T \sum_{i=1}^{V(t)} \sum_{j=1}^{A(t)} X_{ji}(t)\, \mathrm{d}t}{\int_0^T A(t)\, \mathrm{d}t} \geqslant 1 - P_{\text{blk}}\,, \tag{4.8b}$$

$$1 \leqslant V(t) \leqslant V_{\max} \quad \forall t\,, \tag{4.8c}$$

$$\textstyle\sum_{i=1}^{V(t)} X_{ji}(t) \leqslant 1\,, \tag{4.8d}$$

$$\textstyle\sum_{i=1}^{V(t)} Y_{ji}(t) \leqslant 1\,, \tag{4.8e}$$

$$\textstyle\sum_{j \in \mathcal{J}(t)} \rho_j Y_{ji}(t) \leqslant x_{\text{lim}}, \quad 1 \leqslant i \leqslant V(t)\,. \tag{4.8f}$$

Constraint (4.8b) ensures that the number of tasks dropped remain within SLA bounds for service unavailability. Constraint (4.8c) ensures that the number of VMs reserved for an ASP is bounded. Constraint (4.8d) mandates that each task be assigned to a single VM, and (4.8e) indicates that each task can only be running on one VM at a time. Constraint (4.8f) avoids driving the utilization of the VM above an allowable level x_{lim}, which is the threshold learned by the AC agents. This ensures that tasks admitted to a VM will not suffer from unpredictable response times. Figure 4.1, extracted from the extensive analysis in [49], illustrates what AC agents typically observe. Section 4.2.4 details how the AC agents detect and learn the knee of the curve, so as not to drive a VM's CPU utilization beyond limits that would make the response time unpredictably high.

Besides $A(t)$ and ρ_j being unknown functions, the problem presented in Equation (4.8a) is a variant of the knapsack problem, which is NP-hard and

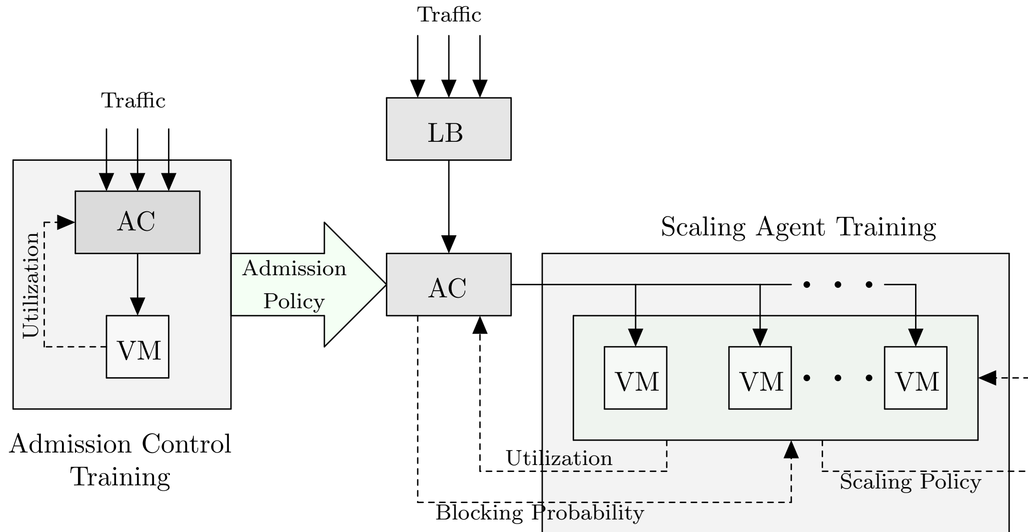


Figure 4.2: Block diagram of training process.

cannot be solved exactly in polynomial time. Furthermore, the analytical modelling of ρ_j is impractical in real environments, because of the excessively high number of concurrent factors that affect it. These include the complexity of an incoming task, the architecture of computing hardware, operating system scheduling and thread handling, the presence of ongoing background processes, etc. Instead, the Q-Learning approximation described in Section 4.1 solves the NP-hard problem near-optimally, under the uncertainty of operational system conditions. Indeed, Q-Learning is known for its versatility in finding near optimal solutions in uncertain settings [8]. We remark that problem (4.8a) optimizes VM provisioning. This means that the horizontal scaler should learn the behavior of the AC agent so as to block the least number of tasks with the least possible number of VMs. Therefore, we chain two separate learning processes: first the AC agent learns how to accept or reject workload to avoid unpredictable service times; then the scaler learns how to act around the AC's behavior to avoid blocking with the least number of VMs. A schematic of this learning process is shown in Figure 4.2

For a practical implementation of the optimization, we use the block diagram shown in Figure 4.3. We call the resulting system SQLR, read as "scaler," because we have crafted a definition of state that embeds short-term memory, as described in Section 4.1.3, for the scaling agent, and because SQLR can be classified as a dynamic resource-provisioning scheme. SQLR comprises: a LB, AC and a scaling agent. We describe each component in the following subsections.

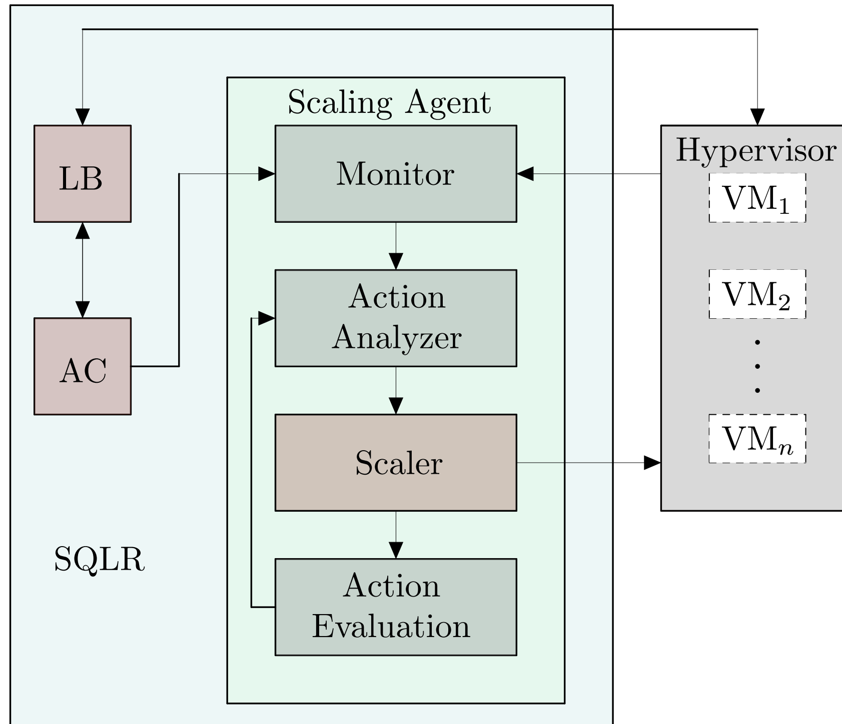


Figure 4.3: SQLR block diagram. "LB" is the Load Balancer agent and "AC" is the Admission Control agent.

4.2.3. Load Balancer

As this component does not constitute a contribution of our work, we only mention it briefly here. We log CPU utilization at 1 s intervals. Our LB works by delivering an incoming task to the VM with the lowest, most recently logged CPU utilization at the time such task arrives. This policy is similar to server state-based strategies used for classic web traffic [52], and yields a high probability that the available resources are evenly loaded in the long run. Hence, the disparity in overall response times is also reduced.

4.2.4. Admission Control

In a resource-constrained system, admission control ensures that the system does not take on more tasks than it can satisfactorily handle. According to [49], it is possible to use the theoretical utilization bound (x_{bnd}) to make the admission decision. However, to obtain the best results for an actual system, the admission control agent must learn an appropriate admission limit (x_{lim}) that takes the system configuration into account. As mentioned in Section 4.1, we do this by treating admission control as an MDP.

AC action and state spaces—The action space of the AC agent consists of the mutually exclusive options:

1. ADMIT: allow an incoming task to be served by a VM;

2. DROP: refuse service to an incoming task.

The state space derives from the quantized levels of resource utilization on the VM serving the task. The resource we consider in this work is CPU utilization. This low-level metric correlates well with the workload, and does not require any domain-specific knowledge of the deployed application [27]. Bearing in mind that CPU utilization greatly impacts response times, we choose the upper utilization threshold as the one beyond which the service times will likely violate the agreed SLO. We use this threshold as a target to determine the rewards/penalties the admission controller will accrue as it builds a policy using Q-Learning. Building an AC policy therefore consists in identifying the most rewarding action (ADMIT or DROP) for each state.

Discretized AC state space—In order to obtain a discretized state space, we partition the utilization values corresponding to predictable response times into regions. To this end, we employ the geometric quantizing function:

$$x_j = \left\lfloor \left(1 - \left(\frac{1}{2}\right)^j\right) x_{\mathrm{bnd}} \right\rfloor, \quad j = 0, 1, \ldots, n, \tag{4.9}$$

All values above x_n (which correspond to a CPU utilization greater than $(1-(1/2)^n)x_{\mathrm{bnd}}$) can be regarded as a single, undesired state, and need no further quantization. Note that x_n is the quantization level closest to the ideal utilization. Therefore, operating a VM beyond x_n likely leads to service times that violate SLOs.

By using the geometric quantizer provided in Equation (4.9), we achieve both coarse and fine adjustment. The quantizer is coarse and reduces the state space (and hence the time needed to train the agent) by sparsely quantizing the load levels just below x_{bnd}. At loads closer to x_{bnd}, the quantizer becomes fine-grained. Therefore, the AC agent learns how the VM responds to such high loads with a small quantization error. Indeed, it is fundamental to accurately learn which load value $x_{\mathrm{lim}} < x_{\mathrm{bnd}}$ ensures predictable service times in a real system. In fact, the value of x_{bnd} is inferred from ideal theoretical analysis, hence it is practically too high and may lead to undesirable service times. Therefore the AC agent employs $x_{\mathrm{lim}} < x_{\mathrm{bnd}}$ to make admission choices.

We now explain the details of load limit calculations in theory (x_{bnd}) and in practice (x_{lim}).

Theoretical AC admission bound—We choose the CPU utilization threshold x_{bnd} based on the analytical results described in [49], and relating response times to occupancy in a processor sharing queue. The time T, taken by a processor with capacity C operations per second to serve a request requiring ℓ operations is given by:

$$T(\rho) = \frac{\ell}{C - \rho}, \qquad \rho = \frac{\lambda}{\mu}, \tag{4.10}$$

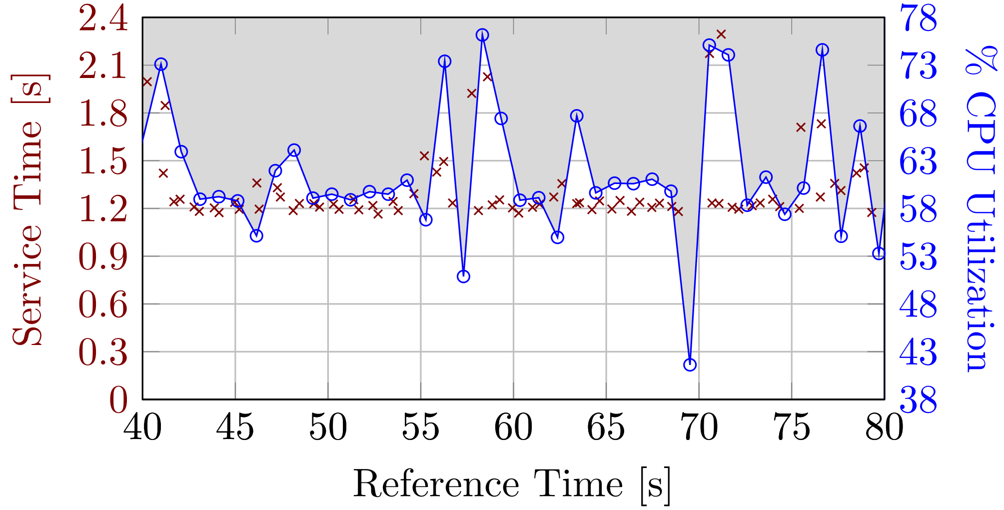


Figure 4.4: Influence of CPU utilization on service response times.

where λ is the arrival rate (i.e., the workload) and μ is the departure rate. The occupancy of the processor ρ is here considered as its utilization level.

We refer the reader back to Figure 4.1, which plots Equation (4.10). The point at which the gradient of the curve changes from an almost constant value to an exponential rise is chosen as the threshold beyond which service times become unpredictable and unreliable. We compute this value by taking the intersection of the tangent to the curve at the initial point with 0% utilization with the tangent at the point where the gradient is approximately 0.5s per 1% rise in utilization. This queuing theory result for x_{bnd} assumes Poisson arrivals, but it fits well our experimental observations. An example of such an observation is depicted in Figure 4.4, for the hardware/software configuration of a VM in our testbed. Considering tasks that require about 1.2 s to complete, we observe that service times are relatively constant around 1.2 s for utilization values lower than 62%. Instead, service times vary wildly for higher utilization levels. Accordingly, we set $x_{\text{bnd}} = 60\%$ to ensure a safety margin when building the discretized state space.

AC admission bound based on learned rewards—The immediate reward R for the action taken by the AC agent is the load x, discretized to the nearest quantized level boundary (downwards for a DROP decision or upwards for an ADMIT decision), *after* an AC decision is made. Therefore, with reference to Figure 4.5, we calculate the reward for making a decision while the quantized load is x_i and observing a resulting level of utilization x as:

$$R(x \mid x_i) = \begin{cases} x_k, & \text{if DROP;} \\ x_{k+1}, & \text{if ADMIT,} \end{cases} \tag{4.11}$$

where $k = \arg\max_j (x_j < x)$. At the boundary, $x_k = x_n$, and $x_{k+1} = x_{\text{bnd}}$. Beyond the

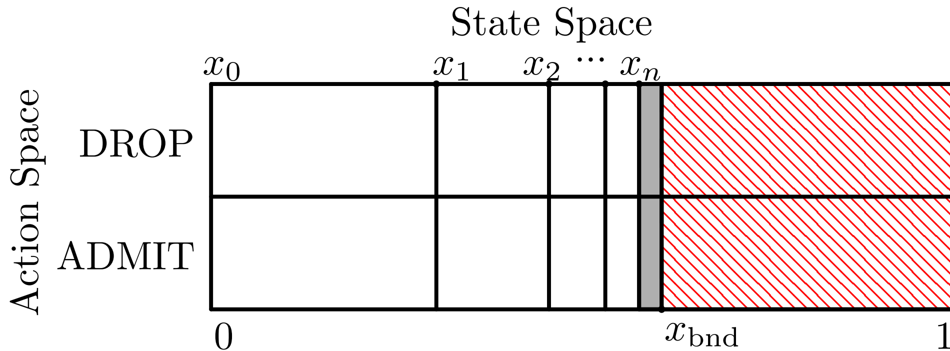


Figure 4.5: Q function table to train the AC. The gray area represents the ideal operating region at which resources are highly utilized and the service times are within SLOs. The red-shaded area on the right represents the region where VM operation is likely to cause SLO violations.

boundary, when $x > x_{\text{bnd}}$, $R(x)$ is defined as

$$R(x \mid x_i) = \begin{cases} x_{\text{bnd}}, & \text{if DROP} \\ \frac{1}{2}(x_{\text{bnd}} - 1), & \text{if ADMIT.} \end{cases} \tag{4.12}$$

As $x_{\text{bnd}} < 1$, Equation (4.12) states that the reward for an ADMIT decision beyond the boundary is negative. This represents a penalty for violating the allowable CPU utilization limit.

For each x_i, the AC agent learns the optimal ADMIT/DROP policy by using the weighted fair exploration mechanism detailed in Section 4.1.6. Initially, the agent drops the tasks with probability 0.5, and subsequently it drops or accepts them according to the action that corresponds to the highest Q-value (as computed with Equation (4.4)) with probability $1-\epsilon$. The training continues until each state (i.e., each quantized load region) is eventually marked as either ADMIT or DROP. This is when the AC policy *converges*. To do so, the final ADMIT/DROP marking of a region is determined after a minimum number of visits. In our case, ϵ goes to 0 after 1000 learning steps per load interval, so that if the accumulated Q-value for ADMIT is higher than the one for DROP, the load region will be marked as ADMIT, and DROP otherwise. Therefore, by training the AC agent with stochastic load variations, we can identify x_{lim} as the highest quantized value x_i for which the AC agent admits tasks.

Given the structure of the chosen reward function, x_{lim} prudently aims at maximizing the utilization of resources at a VM without violating response time requirements. Therefore, the scaling agent can use x_{lim} to make optimal scaling decisions, as shown in the next Section 4.2.5.

Training the AC agent in practice—Having determined x_{bnd} to be 60%, the geometric

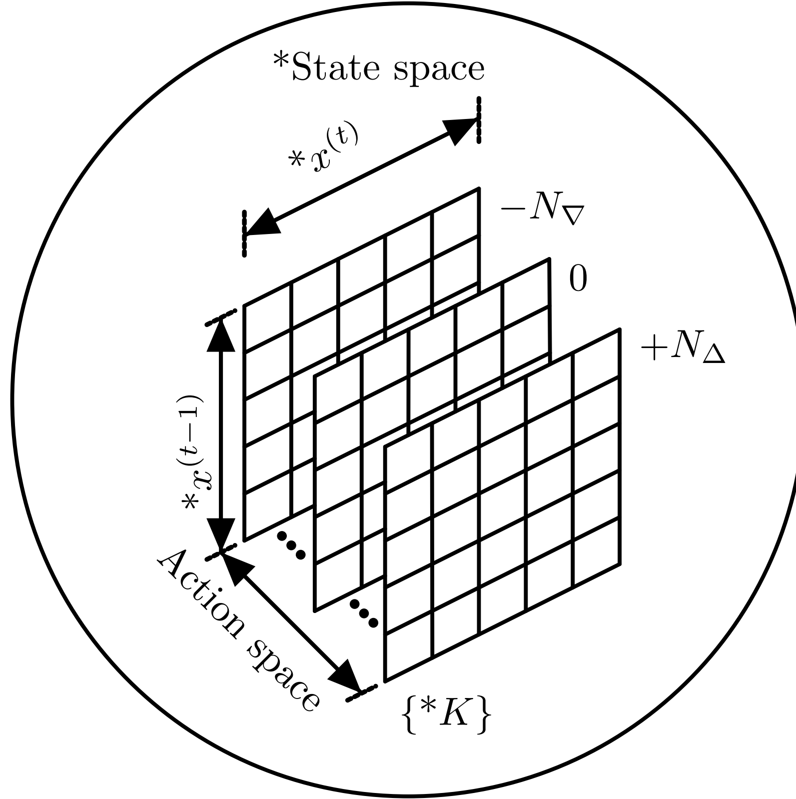


Figure 4.6: SQLR action and state space. K is the current number of active VMs, N_Δ is the number of VMs that can be added and N_∇ is the number of VMs that can be removed. In general, $N_\Delta \neq N_\nabla$. The state space, whose parameters are prefixed by (*), comprises the number of active VMs and the quantized values of the average CPU utilization for the set of active VMs.

quantizer is fully defined and the AC training phase can start. In practice, we only need to train one VM, because we have assumed homogeneity across VMs. Thus, we send tasks towards one VM and start updating the Q-values for ADMIT an DROP actions in each state. To explore all quantized levels of CPU utilization, we generate workload with high variability in inter-arrival times. When we visit a state for the number of times prescribed in Equation (4.5), this state converges, and we lock the policy for this state into either an ADMIT or DROP decision, according to which one has the highest accumulated Q-value Equation (4.4).

4.2.5. Scaling agent

We design and implement a Q-Learning scaling agent whose objective is to achieve as low a blocking rate as possible with as few resources as possible, according to the task admission policy AC agents previously learned.

Action and state spaces for horizontal scaling—The scaling agent adds VMs (scale-out) or removes VMs (scale-in) as appropriate, given the recent history of utilization of active VMs. Therefore the action space for the scaling agent is given by the range of VMs that can be added or removed. The state space consists of three values (*i*) the current number of VMs, (*ii*) the quantized values of the average CPU utilization for the set of active VMs in the *previous* epoch, and (*iii*) the quantized values of the average CPU utilization for the set of active VMs in the *current* epoch.

We show the state and action spaces for our horizontal scaling agent in Figure 4.6. We represent each permissible action as a "card" indicating the number of VMs that must be added or removed when taking the action associated with the card. Moreover, each card consists of a grid (see Figure 4.7) whose rows and columns are indexed with load levels. These levels represent the immediate past load and the current quantized load, respectively, thus expressing the short-term memory hidden in the state. The cells contain the cumulative reward obtained by a given state-action pair. Here, we quantize VM loads uniformly, so as to obtain a more granular view of system-wide resource utilization than a geometric quantizer would achieve. We choose uniform steps of 2% in the region between 0 and 20% of utilization, and steps of 5% in the region between 20% and x_{lim}. The finer sampling between 0 and 20% utilization yields better control when the workload is low: in these cases, only a marginal change is observed when a VM is added or removed. By way of contrast, if utilization is already high, adding or removing a VM causes significant utilization changes. Thus, quantizing utilization more coarsely already enables the detection of such changes, while reducing the state space. Finally, the region above x_{lim} conglomerates into a single, large level. In fact, at this region of utilization, a coarse scale-out decision is most likely, and does not require a fine resolution in the state space representation.

Scaling rewards—The scaling reward function (R_{sqlr}) consists of two components: (i) R_{blk}, computed by comparing the blocking probability P observed after a scaling to the maximum allowable blocking rate P_{blk}, and (ii) $R_{\text{res}} \leqslant 0$, which expresses the cost of resources, and depends on the number K of active VMs after the scaling decision). Specifically:

$$
\begin{aligned}
R_{\text{sqlr}} &= R_{\text{blk}} + R_{\text{res}}; \\
R_{\text{blk}} &= \begin{cases} R_{\text{min}}, & \text{if } P \leqslant P_{\text{blk}}; \\ \theta\left(P_{\text{blk}} - P\right), & \text{if } P > P_{\text{blk}}; \end{cases} \\
R_{\text{res}} &= \beta(1 - K),
\end{aligned}
\tag{4.13}
$$

where R_{min} is a small positive reward that the agent accrues as an incentive for keeping the system within the allowable service outage limits. The training parameters θ and β act as modifiers, so that blocking probability violations receive a different penalty than the

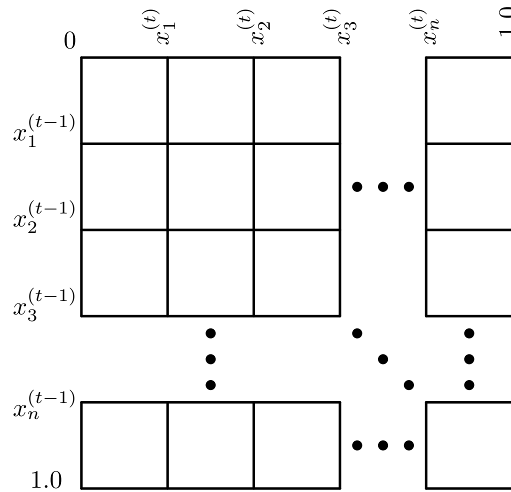


Figure 4.7: State space detail for a "card" in the action space. Each cell's index pair is given by the quantized level of average system-wide resource utilization in successive epochs.

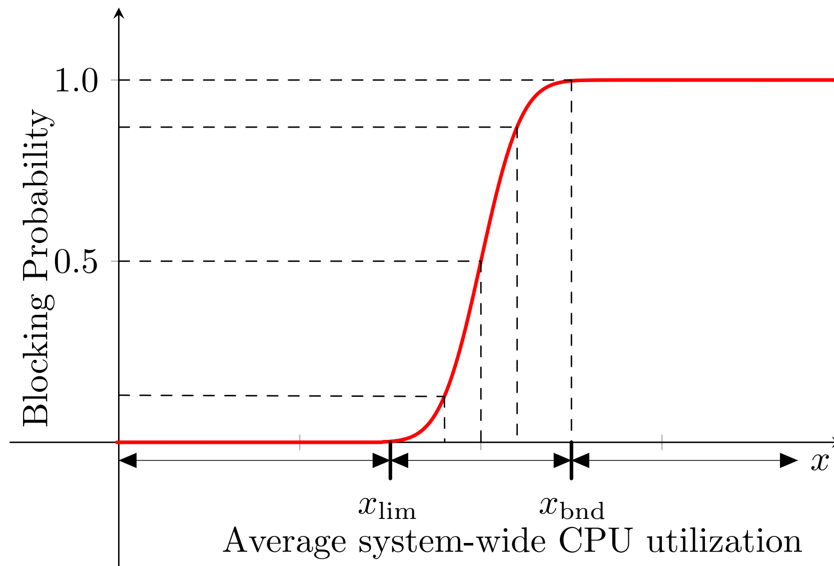


Figure 4.8: Modified error function to estimate the blocking probability component of the initial Q values of card "0" (Figure 4.6) diagonals.

use of unneeded extra resources. This makes the scaling agent flexible. In fact, different CSPs may give different weight to SLOs violation penalties and cost savings achieved by reducing resource usage.

Initialization—As stated earlier, each card in the bubble shown in Figure 4.6 consists of a grid, whose cell indices correspond to the average level of utilization of the active VMs over the previous epoch and the current epoch (cf. Figure 4.7). We initialize the

diagonal elements for the cases where the number of VMs remain unchanged (card "0" in Figure 4.6) as non-zero values. This helps drive initial decisions, e.g., to penalize scale-in and promote scale-out if the average utilization of the current number of VMs is too high. We set these diagonal values based on the following rationale: if the average utilization is below x_{lim}, i.e., the safe limit learned by the AC agent, no blocking is expected (zero probability). Conversely, if the utilization exceeds x_{bnd}, blocking is almost sure to happen. Instead, intermediate utilization values $x_{\text{lim}} < x < x_{\text{bnd}}$ yield a blocking probability that increases with x. Formally, we set the diagonal elements to equate the blocking probability $P_0(x)$, defined as follows:

$$P_0(x) = \begin{cases} 0, & x < x_{\text{lim}} \\ 1, & x > x_{\text{bnd}} \\ \frac{1}{2}\left[1 + \text{erf}\left(\eta(x)\frac{\text{e}}{\sqrt{2}}\right)\right], & \text{otherwise} \end{cases} \tag{4.14}$$

where

$$\eta(x) = \frac{x - x_{\text{lim}}}{x_{\text{bnd}} - x_{\text{lim}}}, \tag{4.15}$$

and $x = x_1^{(t)}, x_2^{(t)}, \ldots, x_n^{(t)}, 1$. The above definition for the case $x_{\text{lim}} < x < x_{\text{bnd}}$ yields a smooth transition between blocking probabilities 0 and 1, as shown in Figure 4.8. We recall the AC agent adaptively learns x_{lim} during its own training phase, so we can assume that the scaler knows the safe value of x_{lim} for any admissible number of active VMs. The diagonal elements computed above serve as the reference action values, $Q(S^{(t+1)}, a)$, for the updates in Equation (4.4) after horizontal scaling.

The scaling procedure—With the actions, states, rewards and initialization described above, the scaling process is succintly depicted in Algorithm 1. Figure 4.9 details a scale-out action. To describe the latter, we consider starting after a previous action having taken place at instant $t-1$. The first cell index is the quantized level of the average utilization in the interval $[t-2, t-1)$. At instance t (bottom bubble), our scaler obtains the quantized level of the average utilization in the interval $[t-1, t)$. This serves as the second cell index to be considered in selecting the action. The current number of active VMs, K, is also evaluated.

With this triplet of values, the current state is established, and we are ready to choose a scaling action (i.e., scale-in, scale-out, or keep the current number of VMs) based on Equation (4.5) and Equation (4.6). To do so, recall that every card corresponds to a scaling action, e.g., add 1 VM, remove 2 VMs, etc. We check the convenience of every action by reading the Q-value of the cell indexed by the quantized average utilization values at the current epoch (t), and at the previous one ($t-1$).

Then, we choose a scaling action based on Equation (4.5) and Equation (4.6), by leveraging the above Q-value entries in every card of the action space within the bubble

Algorithm 1: Scaling Agent Algorithm

```
   Result: Scaling action, updated table of Q-values
1  RunCount ← 0
2  n ← 0
3  while True do
4      N_t ← getActiveVMs()
5      U_current ← 0
6      foreach vm ∈ VMs do
7          U_current ← U_current + getUtils(interval, vm)
8      end
9      x_t = getQuantizedUtil(U_current / N_t)
10     if RunCount >= 2 then
11         Q_{t+1} = ReadQTable(N_t, x_t, x_{t-1})
12         n = getStateVisits(N_t, x_t, x_{t-1})
13         Q_t = ReadQTable(N_{t-1}, x_{t-1}, x_{t-2})
14         R = ComputeR(N_t, getBlocking(interval))
15         R' = R + γ * Q_{t+1}
16         Δ = R' - Q_t
           // update Q-value
17         Q(S^(t), A^(t)) ← (n/(n-1)) * Q_t + (1/n) · Δ
           // update state visits
18         N(S^(t), A^(t)) ← n + 1
19     end
20     if n < N_RefVisits then
21         ϵ ← 1 - (n / N_RefVisits)
22     else
23         ϵ ← ϵ_min
24     end
25     ArrWFE = 0_{1000ϵ×1000ϵ}
26     ArrGRD = 1_{1000(1-ϵ)×1000(1-ϵ)}
27     ArrALL = concatenate(ArrWFE, ArrGRD)
28     if ArrALL[RandomInt()] == 0 then
29         Scale with weighted fair exploration
30     else
31         Scale according to max(ReadQTable(N_t, x_t, x_{t-1}))
32     end
33     N_{t-1} ← N_t
34     x_{t-2} ← x_{t-1}
35     x_{t-1} ← x_t
36     RunCount ← RunCount + 1
37 end
```

defined by K VMs.

For later reference we term the cell of the chosen action card as R-Cell (marked red in Figure 4.9). After waiting shortly for the VMs to start up or shut down, and for the

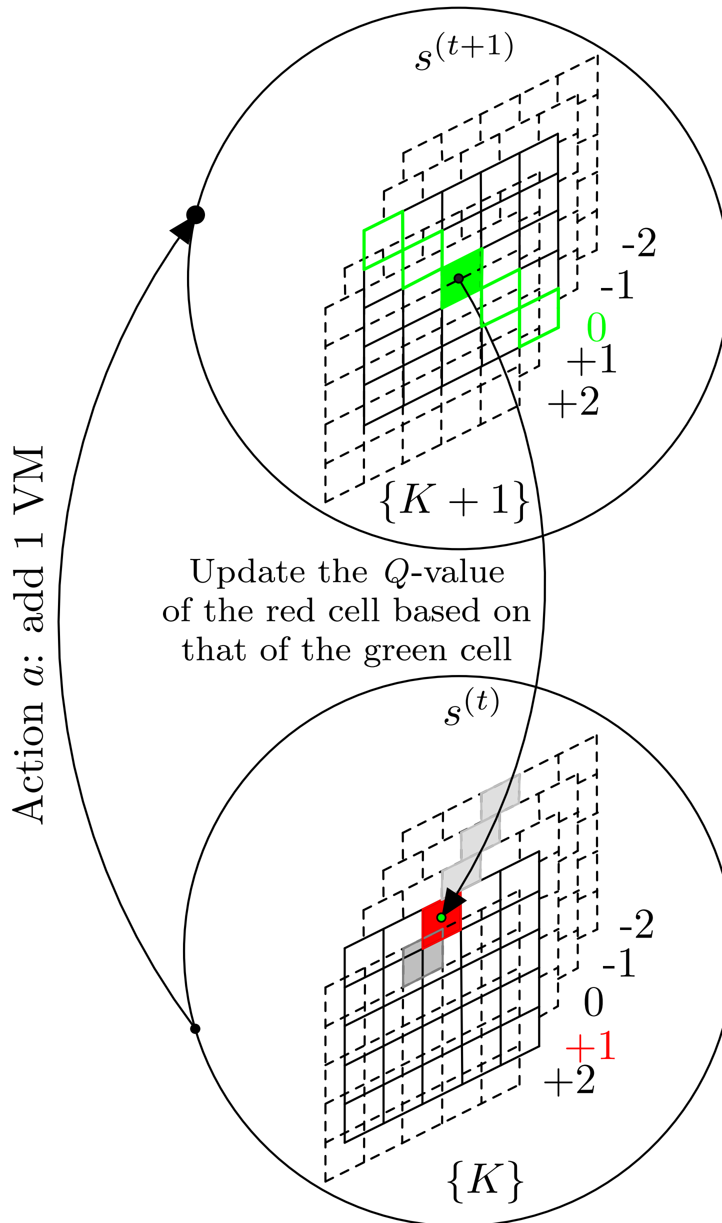


Figure 4.9: SQLR's horizontal scaling mechanism. We compare the Q-values of the grey-shaded cells in order to determine the best action according to Equation (4.6). Here, we choose a scale-out of +1 VM. After the scaling action a, the Q-value in the red cell receives the update as specified in Equation (4.4). One component of the update is the Q-value contained in the green cell of card "0" in bubble "$\{K+1\}$".

effect of the change to become manifest, we reach instant $t+1$. We can now compute the immediate reward as described in Equation (4.13): this accounts for the blocking probability observed between time instants t and $t+1$, and for the number of active VMs

at instant $t+1$. We also take into account the accumulated reward stored in card "0" at the diagonal cell indexed by the quantized average utilization value over the interval $[t, t+1)$ (this cell is colored green in Figure 4.9). The above two values are used to update the Q-value in R-Cell as prescribed in Equation (4.4).

How to train the scaling agent—After having trained the AC, we need to create a set of tables of the Q-values for all scaling actions. The number of tables depends on the highest number of VMs that can be provisioned, as well as on the number of VMs that can be added or removed within a single scaling decision. Then, with an instance of the (already trained) AC running at each active VM and a global scaling agent running, we expose the system to varying offered load profiles, and act according to Algorithm 1. As the scaling agent adds or removes VMs from the host, we monitor the blocking rates experienced and the number of running VMs, and generate rewards to update the table of Q-values related to the scaling agent's decision.

We check the relevant table every 120 seconds, which constitutes one epoch, and immediately call for a scaling decision whose action is selected according to Equation (4.5) and Equation (4.6). When the number of visits of a state reaches the prescribed count level M, then $\epsilon = \epsilon_{\text{min}}$ and the policy for that state has converged.

4.3. Experiments

4.3.1. Testbed

In order to evaluate the effectiveness of our scheme, we run experiments on a testbed that mirrors the operations of a CSP. We set up the testbed as shown in Figure 4.10. The architecture of our Dell T640 server consists of two processor sockets with non-uniform memory allocation (NUMA), 10 hyper-threaded CPU cores per socket for a total of 40 logical cores with a variable clock rate. The server memory is 128 GB.

The server runs Ubuntu 18.04 LTS as its operating system and acts as a host for VMs. The client PCs run on Ubuntu 16.04.3 LTS. We use the KVM hypervisor, and manage the VMs using libvirt [35]. Each instance of a VM is configured with 4 virtual CPUs and 4 GB of memory. The client PCs and the server are connected via a Cisco switch to form a Gigabit/s local area network. The PCs function as ASPs running bash scripts that generate requests to the server with varying rates as depicted in Figures 4.11 and 4.12.

As our cloud application, we choose the algorithm used for proof-of-work computation in bitcoin mining [53]. It is a suitable stand-in for resource-hungry, computationally challenging tasks that are commonly deferred to the cloud such as encryption [54] and transcoding [55]. Each iteration of this computation involves incrementing a counter variable (nonce), hashing it together with a given hash code and merkle root, and then hashing the outcome again. The hashing mechanism is the 256-bit Secure Hash Algorithm

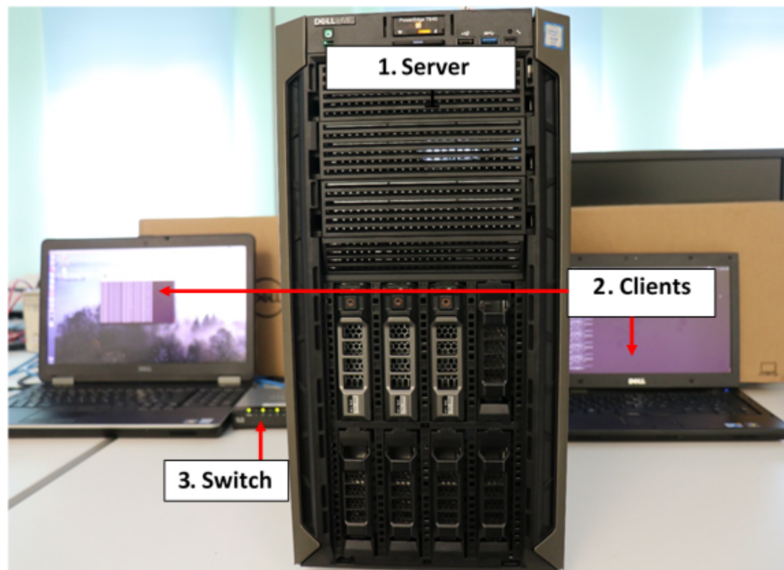


Figure 4.10: Testbed setup. (1) Dell T640 server: Hosts KVM hypervisor, VMs, Admission controllers and Scaling Agent. (2) Client PCs: Generate requests towards the server according to demand profile. (3) Gbps switch: Creates LAN between Clients and Server.

(SHA-256). We will use the word *job* to refer to one proof-of-work iteration from hereon. In order to mimic the varying degrees of complexity of typical cloud applications, we consider a different number of iterations for each request. Specifically, a request can generate any number of iterations in the discrete set {300k, 400k, ... , 1200k}.

The server launches VMs to handle incoming requests according to one of the following schemes: static provisioning, extended Kalman filtering based prediction [25], Reinforcement Learning-based Proactive Auto-Scaler (RLPAS) [22] and our proposed scaling scheme. All agents, including the admission control and load balancer, are implemented in Python and run within the host operating system. For reproducibility, we fully share SQLR's code.[3]

The scheme proposed in [25] leverages a queuing system model enhanced with an Extended Kalman Filter (EKF). It makes near time predictions of response times based on measurements of arrival rates and system utilization. Using a queue model refined by a tuned EKF with the maximum allowable response time (from an SLA) as input, the scheme then calculates the number of nodes needed and scales appropriately to approach this number.

We make some slight modifications to the EKF algorithm to make it more robust. We increase the interval between the predict and update phases from 10s to 90s. This provides sufficient time for starting up a VM and letting it handle tasks. Additionally, instead of the instantaneous measured system utilization and response times, we provide

[3]https://github.com/Constantine-Ayimba/SQLR

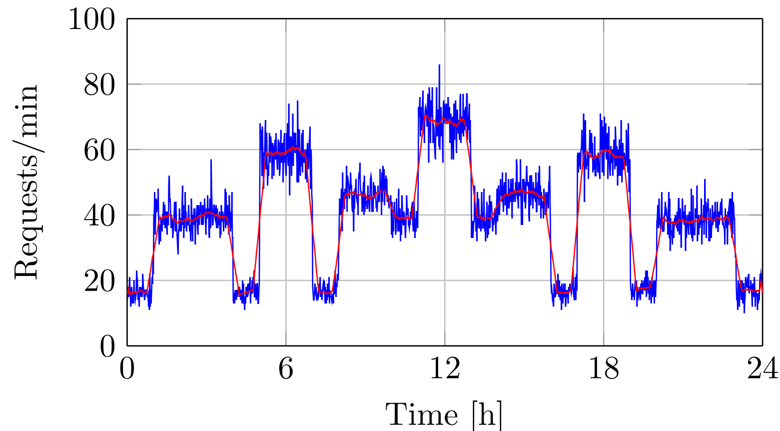


Figure 4.11: Pre-training workload profile. The red line is the moving average of the number of requests per minute, computed over windows of 30 samples.

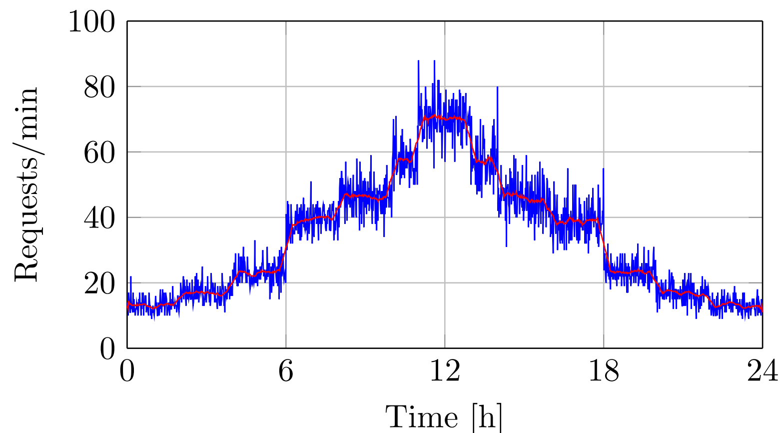


Figure 4.12: Test workload profile. The red line is the moving average of the number of requests per minute, computed over windows of 30 samples.

their average over the predict and update intervals of the filter as input to the EKF. This prevents the scaler from over/under estimating input parameters, and thus yields a fairer comparison to our scheme. Further, we dispense with the network delay in the system model as the response times are taken directly on the server. We consider a single-tiered application, and one class of requests. This also has the effect of simplifying the process and measurement noise covariance matrices to have size 2×2 (as only two parameters are taken into account in each case), thereby enhancing the tuning of the EKF.

We also compare our scaling system to the state-of-the-art RLPAS proposed in [22]. We only consider the response time parameter in our implementation and not throughput, since our stand-in cloud application is compute-intensive. Owing to the use of the load balancer, which distributes the offered load evenly, we set the ratio of utilized to provisioned VMs to 1.

For our scheme, we limit the number of VMs that can be added or removed within a

single scaling action to 2. This truncates the action space, reducing the number of visits required for a state to achieve a stable policy to $M = 50$ (cf. Table 4.1). Therefore, it also limits the number of learning steps needed to attain a stable policy.

As part of the training for our scheme, we combine several workload profiles with different averages, resulting in the composite shown in Figure 4.11.

For the test workload, we again use a combination of several profiles with different averages to obtain the composite shown in Figure 4.12. To achieve this, we configure requests to be sent with inter-arrival times ω drawn uniformly at random from the set of values $\{0, 1, \cdots, \omega_{\max}\}$ seconds, for each hour slot. For example for the busy-hour slot, $\omega_{\max} = 5\,\mathrm{s}$, and for low workload period, $\omega_{\max} = 9\,\mathrm{s}$. This results in high entropy (given the uniform distribution of inter-arrival times) but still allows us to procure similarities between workloads, and evaluate contextual knowledge re-use. Our choice of inter-arrival statistics leads to patterns encountered in real workloads, with rapid variations over short intervals, but with veritable trends over longer observation windows. It also includes sudden bursts and drops, such as those observed at the start of hours 8, 10, 14 and 18.

4.3.2. Implementation on large scale

Although our experiments took place on a small testbed, our scheme still lends itself well to large-scale deployments, e.g., in data centers. The latter can be achieved via the modular approach presented in Section 3.1.2. A conventional *layer-4 load balancer* such as [56] can be used to route tasks to physical servers as shown in Figure 4.13. Given the cost benefits of operating homogeneous hardware in large scale settings [57], most servers in a data center will have the same specifications. This means that, in most cases, the scaling policies learned for one server can be re-used with no need for retraining.

4.4. Results

In this section, we show the effectiveness of the AC and scaling policies. We then examine the results with respect to two SLOs: service availability (as measured via blocking rates) and response times.

4.4.1. Admission control policy convergence

First, we briefly discuss our AC agent. We recall that this component learns the appropriate utilization limit, x_{lim}, that ensures bounded response times.

Figure 4.14 shows how the learning algorithm for the AC trades off exploration and exploitation using our weighted fair exploration scheme, cf. Equation (4.6). The evolution of the accumulated reward for a subset of three state-action pairs is shown in Figure 4.14a. Red, blue and teal-colored lines denote the Q-value evolution for low, intermediate, and

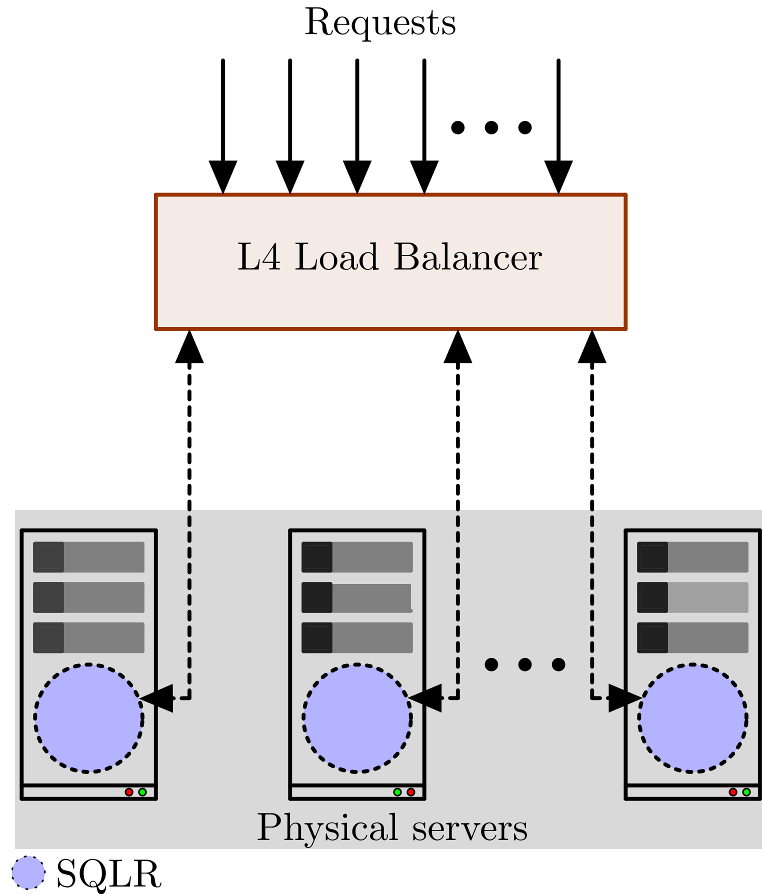
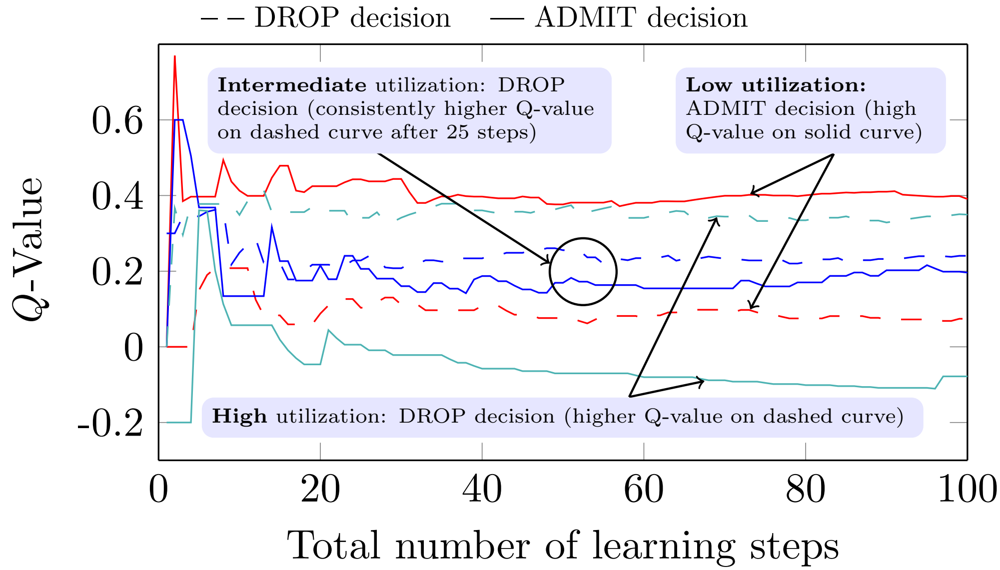


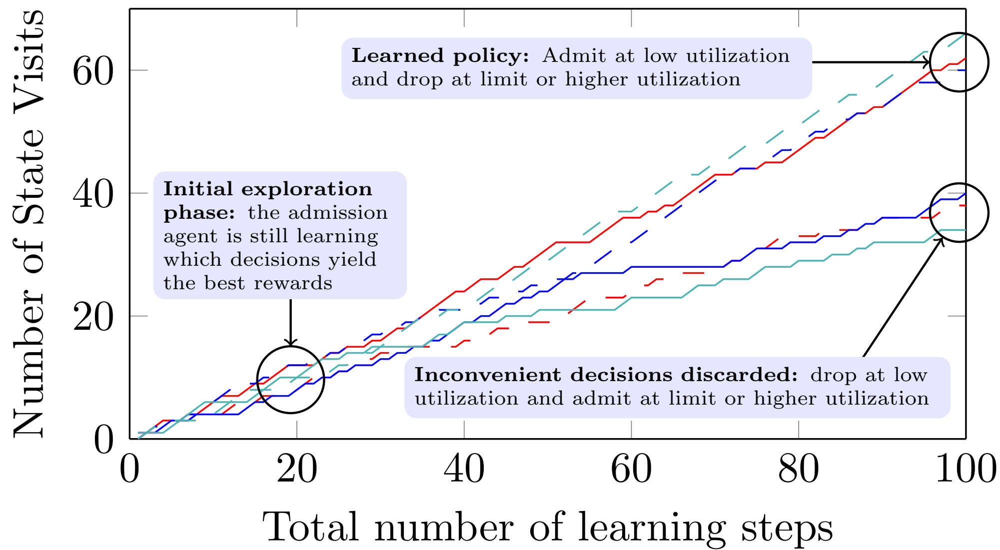
Figure 4.13: Schematic of a modular large-scale deployment. (cf. Section 3.1.2.)

high utilization levels, respectively. Dashed lines refer to Q-values for drop decisions, wheres solid lines refer to admit decisions. In the initial learning phases, the difference between the values is not as distinct, and the admission control agent makes a DROP or ADMIT decision with about the same probability. After 1000 learning steps, the agent has understood which decisions yield better rewards (or at least lower penalties). For example, in high utilization regimes (teal lines), DROP decisions (dashed line) have a much higher Q-value, and are thus much more likely than ADMIT decisions (solid line). Conversely, at low utilization (red lines), ADMIT decisions are much more likely. At intermediate utilization, the difference between the Q-values of ADMIT and DROP decisions is not as stark, but still associated with a DROP decision.

The above results suggest that utilization levels up to 0.45 result in ADMIT decisions, whereas levels exceeding 0.45 start making DROP decisions more convenient. In other words, the scaling agent learns the limiting value of utilization x_{lim} to be 45%. Therefore, once the LB has chosen a VM that should serve an incoming task, the AC agent drops the task if the VM's utilization is higher than this learned value of x_{lim}, and accepts the requests otherwise.



(a) Reward accumulated with experience.

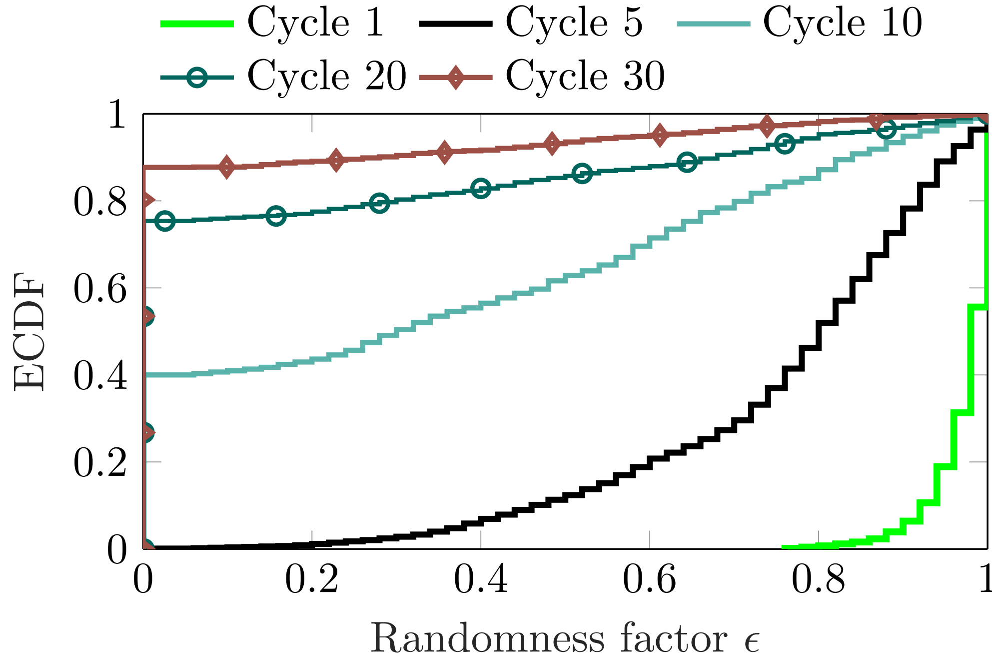


(b) Frequency of DROP and ADMIT decisions.

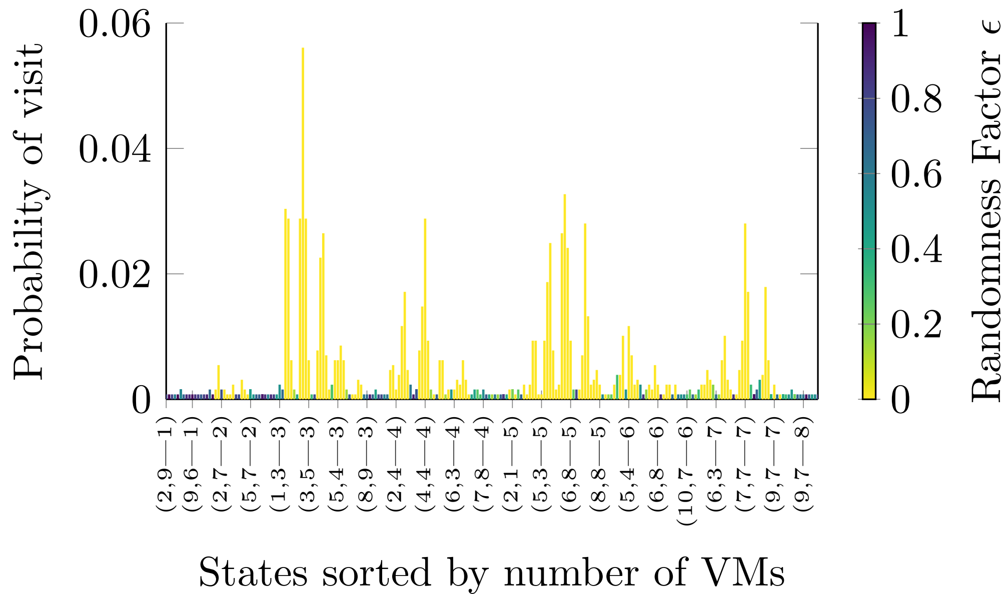
Figure 4.14: Admission Control training. Red curves: low utilization level between 30% and 45%. Blue curves: intermediate utilization levels between 45% and 53%. Cyan curves: high utilization levels of 60% and above. Dashed lines convey the Q-values of DROP decisions, solid lines of ADMIT decision.

4.4.2. Scaling agent's policy convergence and complexity

In order to characterize the overall state of convergence of the scaling agent, we consider the probability of randomness in action selection, ϵ. Recall that, for each state, we decrease ϵ from 1 down to 0 linearly with the number of visits to that state. Therefore, we use $1 - \bar{\epsilon}$ to express the convergence level, where $\bar{\epsilon}$ is the average value of ϵ computed



(a) Convergence inferred from the distribution of the randomness factor. The rate is faster in the initial cycles as the weighted fair exploration drives the agent to the most promising states more often.



(b) State visitation probability with convergence at approx 93.5% ($\bar{\epsilon} = 0.065$). The scaling agent acts more greedily, exploiting actions in the converged states.

Figure 4.15: Scaling agent convergence behavior.

across all states.

In Figure 4.15a, in the initial stages are shown, e.g., after one full cycle of the test

workload in Figure 4.12 (green curve), ϵ is high in every state, such that its distribution has a mean $\bar{\epsilon} = 0.97$, corresponding to approximately 3% convergence.

The scaling agent progressively gains greater experience about the workload profile and the corresponding system states. Thanks to this, the agent develops the appropriate scaling policy for each state, and acts less randomly. This yields diminishing values of ϵ in the related states and its distribution shifts leftwards and upwards, such that $\bar{\epsilon} = 0.065$ at the 30th cycle (brown curve) in Figure 4.15a, which corresponds to approximately 93.5% convergence.

Over subsequent runs, the weighted fair exploration mechanism drives the scaling agent to visit the most pertinent states more often, as they procure better rewards. These states correlate more strongly to the underlying workload profile. At advanced levels of convergence, with low values of $\bar{\epsilon}$, the scaling agent chooses actions promising higher rewards, resulting in more visits to familiar states with fully converged policies. This is shown in Figure 4.15b, where those states for which the policy has converged (yellow bars) are visited most likely as expected. However, weighted fair exploration still ensures a few visits to less familiar states (blue and green bars), and guarantees that the agent will be able to learn different policies, should it observe different workload patterns in the future.

For a reinforcement learning agent, the ultimate aim is take actions that *maximize the accumulated reward; or minimize the penalties (negative rewards).* Figure 4.16 shows the sum of all the Q-values corresponding to any action, computed at different snapshots. For an increasing number of learning steps, the scaling agent approaches convergence, and accrues progressively lower penalties. As depicted in Figure 4.16, policy convergence occurs rapidly within the first 5000 epochs. This is because the parts of the workload with similar patterns influence the scaling agent to visit some states more often. The rate then slows as the scaling agent should observe less frequent workload patterns repeatedly in order to decide on the best policy. After about 30k epochs, most pertinent states have fully converged. The negative values are expected because of the way we structured the reward function in Equation (4.13): this function issues penalties commensurate to the number of VMs provisioned in excess of the first one.

Complexity—We first consider complexity in terms of the number of learning steps required to attain convergence. Recall the representation of utilization in the agent's state space as depicted in the "cards" of Figures 4.7 and 4.9. Each cell of a grid requires M updates (the number of visits until $\epsilon_{\min}$) for convergence. Call S the utilization component of the agent's state space and define $\chi := |S|$. Define also the total number of actions from all states as $\zeta := \sum_{i \in n} |A(i)|$ where n is the maximum number of VMs available to an ASP, and $A(i)$ is the total number of actions available to the scaling agent when i VMs are active. In the worst case, the maximum number of steps required to reach convergence is at most $O(M\zeta\chi)$ steps. Given that $\chi < M\zeta < \chi^2$ by design, our complexity is between

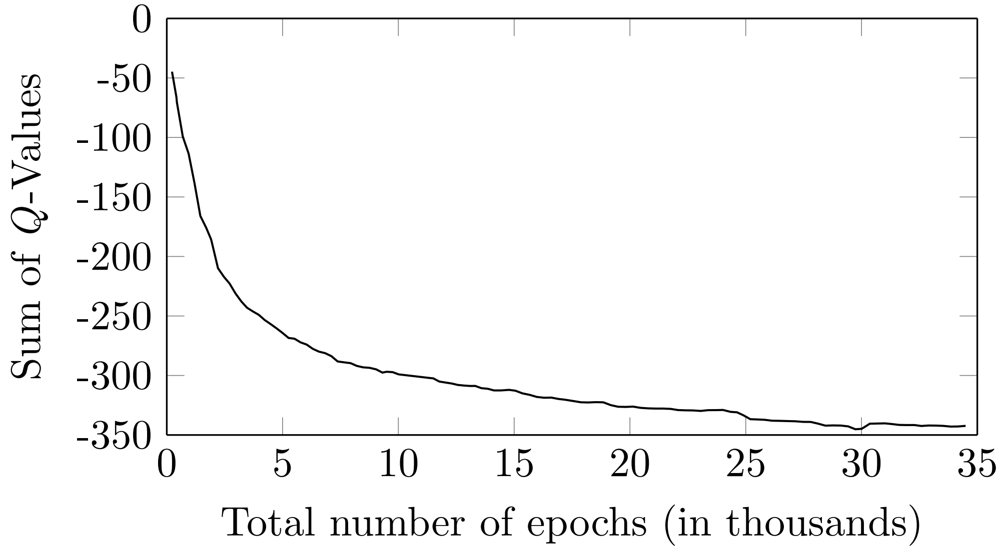


Figure 4.16: Cumulative Q-Values, i.e., the sum of Q-values for all states, taken at different snapshots in the course of the experiment.

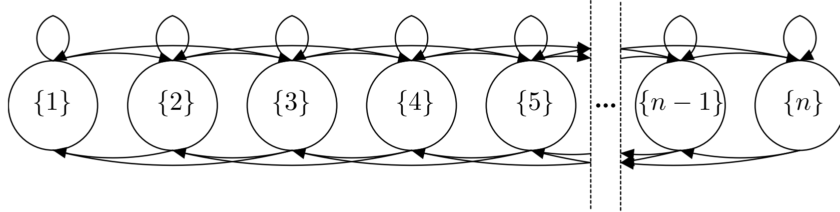


Figure 4.17: Markov chain of possible actions from selected states. The numbers in curly brackets within each bubble, $\{\cdot\}$, point to the number of VMs. Right-pointing arrows from a state indicate scale-out, whereas left-pointing arrows indicate scale-in. The re-entrant arrows above each bubble indicate no scaling.

$O(\chi^2)$ and $O(\chi^3)$ which aligns with the classical analysis presented in [58].

In our implementation, for the worst-case complexity we have that the total number of VMs is $n = 10$, and the number of permissible actions $A(i)$ depends both on the number of active VMs i and on n. We clarify this through the Markov chain representation in Figure 4.17, which shows all possible transitions between different numbers of VMs, denoted as the value $\{i\}$ inside each bubble, $1 \leqslant i \leqslant n$. For example, when 1 VMs is active, the agent can decide to keep 1 VM or rather scale out to 2 or 3 VMs. Instead, when 5 VMs are active, the agent can remove or add up to $N_\nabla = N_\Delta = 2$ VMs, or keep the current 5 VMs. Therefore, the admissible actions are 3 (for states having 1 and 10 VMs), 4 (for states having 2 and 9 VMs) and 5 for the rest. Therefore $\zeta = 2 \cdot 3 + 2 \cdot 4 + 6 \cdot 5 = 44$. The utilization component of the state space comprises 10 quantized levels (from 0% to 20%), 5 levels (from 20% to 45%) and 1 level (beyond 45%). Therefore, $\chi = 16 \cdot 16 = 256$. Because we set $M = 10|A(i)|$, $1 \leqslant i \leqslant n$, the complexity of our implementation is $O(10 \cdot 44 \cdot 256) = O(112640)$ steps. We ameliorate this worst-case

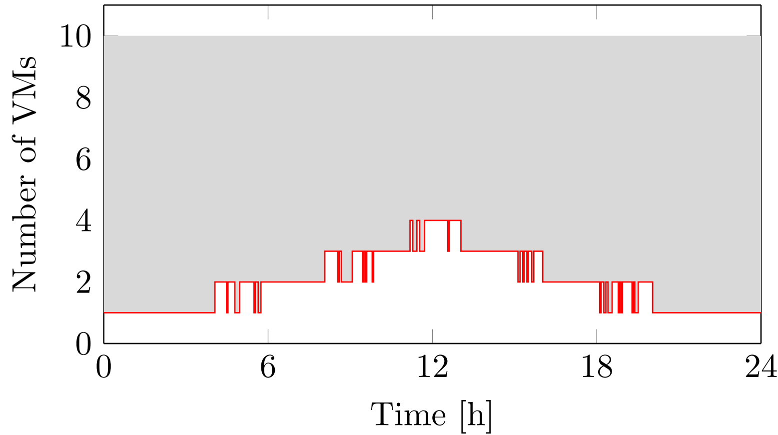


Figure 4.18: VM scaling for the EKF-based horizontal scaling scheme proposed in [25]. We represent resource savings with respect to static over-provisioning with 10 VMs via the gray-shaded area.

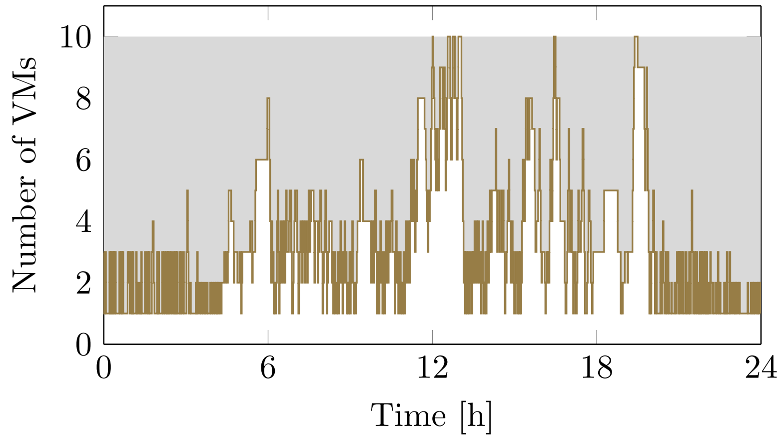
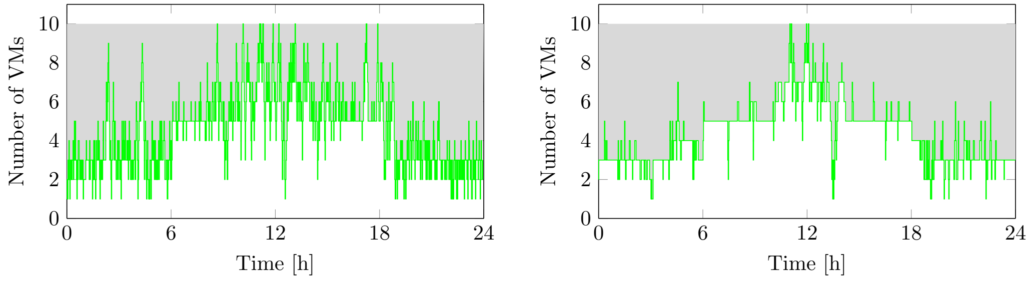


Figure 4.19: VM Scaling for RLPAS: the Q-Learning horizontal scaling scheme proposed in [22]. We represent resource savings with respect to static over-provisioning with 10 VMs via the gray-shaded area.

complexity by implementing weighted fair exploration (cf. Section 4.1.6) and initializing the Q-values of the diagonal elements (cf. Section 4.2.5). Furthermore, the operational complexity of the scaling agent presented in Algorithm 1 is $O(1)$ for all operations except for the load summation loop, which is $O(i_t)$, where i_t is the number of active VMs at epoch t.

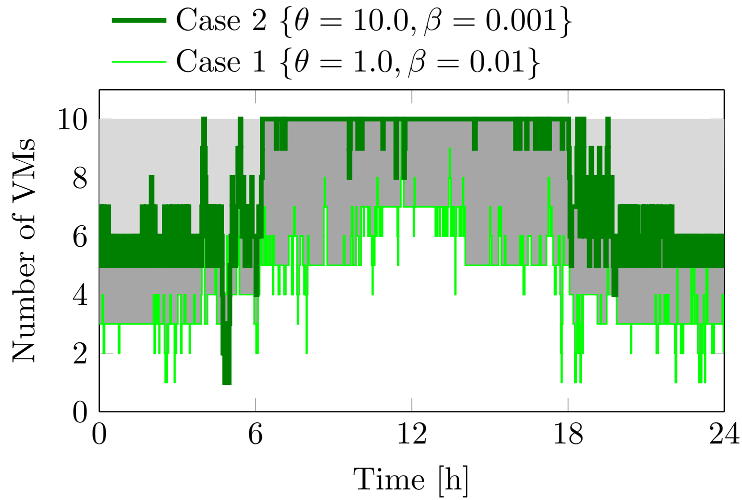
4.4.3. Scaling profiles

We now move to discussing scaling profiles in response to the test workload of Figure 4.12. We do so for all schemes considered in this work, namely SQLR, RLPAS [22], static provisioning, and the EKF-based scheme [25]. The latter produces the profile



(a) After 10 cycles, at 59% convergence ($\bar{\epsilon} = 0.41$). With $\theta = 1$, $\beta = 0.01$. We represent resource savings with respect to static over-provisioning with 10 VMs via the gray-shaded area.

(b) After 20 cycles, at 89% convergence ($\bar{\epsilon} = 0.11$). With $\theta = 1$, $\beta = 0.01$. profile. We represent resource savings with respect to static over-provisioning with 10 VMs via the gray-shaded area.



(c) After 30 cycles, at 93.5% convergence ($\bar{\epsilon} = 0.065$). The lighter shade of gray area represents resource savings made compared to static over-provisioning with 10 VMs for Case 2. The darker shade of gray represents extra resource savings made in Case 1 compared to Case 2.

Figure 4.20: SQLR scaling behavior evolving with experience. For this training phase, we set $P_{\text{blk}} = 0.001$.

depicted in Figure 4.18. The scaling behavior in this scheme is quite stiff, because the EKF tends to filter out bursty workload, that would require greater agility instead.

The RLPAS scaling profile is depicted in Figure 4.19. RLPAS is quite agile compared to the EKF-based scaler. However, it is susceptible to premature scaling decisions. In fact, it relies on the instantaneous workload arrival rate, which is highly stochastic in the test workload profile. Moreover, both the EKF-based scaler and RLPAS require some knowledge of the underlying application, such as its ideal response time. From empirical observations on our test application, we set such ideal response time at 5 μs per job for both schemes, as this amount of time is amply sufficient to serve the greatest majority of

the jobs.

The scaling profile obtained from our proposed scheme, in reference to the test workload, is shown in Figure 4.20. The behavior of the scaler steadily improves with increased exposure to the test workload. As more states converge, the scaling behavior becomes more predictable, as seen by moving from Figure 4.20a to Figure 4.20b and Figure 4.20c. The number of VMs provisioned settles around a suitable number that achieves the best tradeoff between resource cost and penalties as driven through the training parameters θ and β.

Moreover, in Figure 4.20b, we see that the first intervals to exhibit convergence (hence greater stability in the scaling behavior) are those with higher similarity to the training workload of Figure 4.11. For instance the intervals of hours 6 to 8 and 16 to 18, with an average of 40 requests per minute (cf. Figure 4.12), closely resemble those of hours 1 to 4 and 20 to 23 of the training workload (cf. Figure 4.11). This shows that SQLR can re-use contextual knowledge learned from one workload on any subsequent one with similar characteristics.

Assigning different values to the training parameters θ and β results in different scaling responses. As shown in Figure 4.20c, a low value of θ relative to β (Case 1) results in cost-focused scaling policies that emphasize resource cost more than service unavailability due to blocking. This is the same configuration as in Figures 4.20a and 4.20b. When $\theta \gg \beta$, as in Case 2, more service-focused policies are learned, giving greater importance to service availability than to resource cost. The exploration mechanism of the Q-Learning algorithm at the core of SQLR means that it may sometimes make sub-optimal decisions in less known states, resulting in under-provisioning (such as at hour 18 for Case 1, and at hour 5 for Case 2 in Figure 4.20c). This results in relatively high blocking rates, as shown in Figure 4.21. Our guided fair exploration mechanism ameliorates the effects of such under-provisioning, ensuring that their duration is short.

Since the EKF-based scaler relies on workload measurements to predict response times and scale accordingly, it is particularly susceptible to under-estimating resource requirements when demand is low. This is evident at off-peak intervals in Figure 4.21 where, between hours 0 and 7 and between hours 17 and 24, the EKF-based scaler allocates 1 VM on average, resulting in considerable blocking, much higher than the other schemes. The RLPAS scaler adjusts resources too abruptly, resulting in unpredictable blocking at both peak and off-peak hours. This is due to its dependence on direct workload measurements, which are highly stochastic.

Static provisioning results in significant under or over-provisioning, as exhibited by the black (2 VMs) and blue (10 VMs) curves, respectively. Both situations are clearly untenable: on the one hand, the CSP risks serious penalties for service unavailability; on the other hand, the CSP incurs significant yet unnecessary operational expenditure to maintain superfluous resources, even though over-provisioning results in zero blocking.

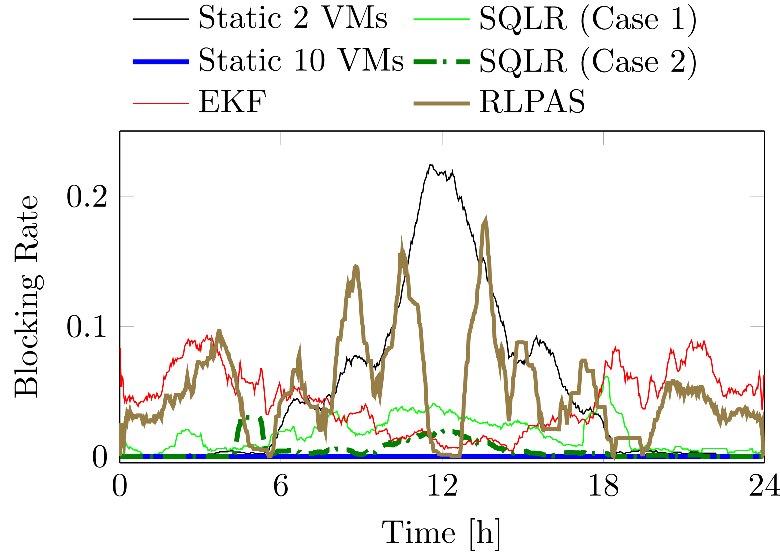


Figure 4.21: Blocking rates over two-minute intervals. Two SQLR configurations are shown: Case 1 ($\theta = 1, \beta = 0.01$) and Case 2 ($\theta = 10, \beta = 0.001$). For clarity, a moving average filter is applied with a window size of 30 samples.

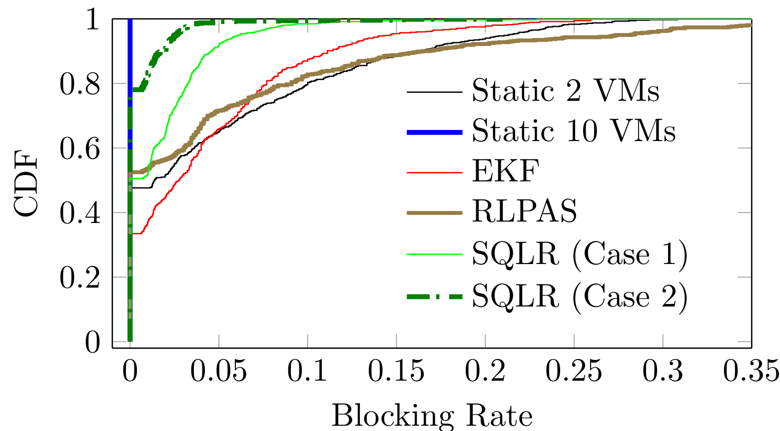


Figure 4.22: Blocking rate distribution. Two SQLR configurations are shown: Case 1 ($\theta = 1, \beta = 0.01$) and Case 2 ($\theta = 10, \beta = 0.001$).

The distribution of blocking rates is shown in Figure 4.22. The greater the number of VMs provisioned, the lower the blocking rate, as an incoming request will likely find a sufficiently under-utilized VM. Given the preceding insight, the EKF-based scaler that provisions the lowest number of VMs, exhibits poor blocking rate performance. RLPAS also performs poorly because of its premature scaling behavior, which occasionally leads to VM removals too soon when workload transients occur. Consider instead Case 2 of our scaling scheme. This configuration penalizes blocking more heavily than provisioning extra VMs by setting $\theta \gg \beta$ in Equation (4.13). Hence, it performs nearly as well as over-provisioning with 10 VMs. If lower service availability is acceptable, our scheme

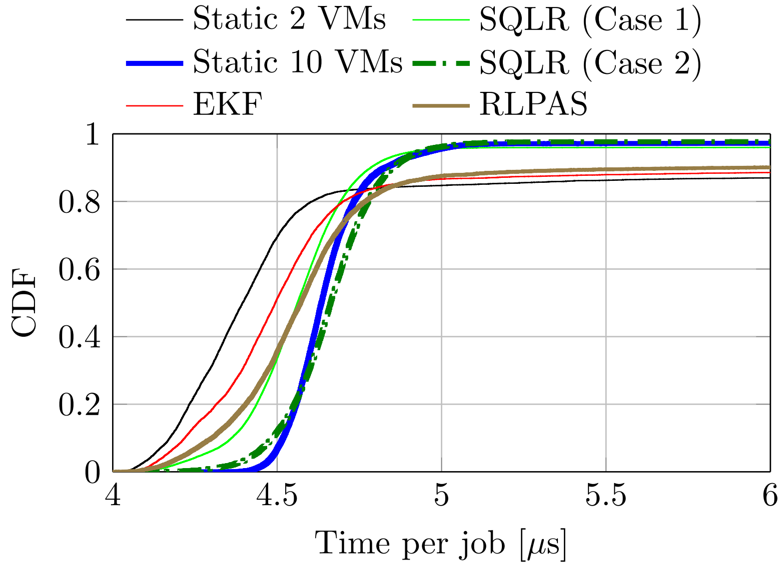


Figure 4.23: Service time distribution per job. Two SQLR configurations are shown: Case 1 ($\theta = 1, \beta = 0.01$) and Case 2 ($\theta = 10, \beta = 0.001$). The service time for each request is divided by the corresponding number of iterations it generates to obtain the time per job.

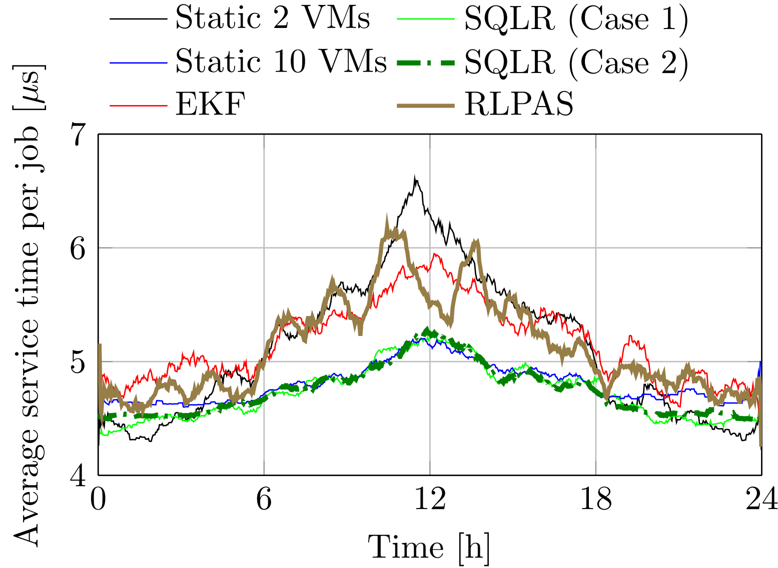


Figure 4.24: Moving averages of service times (taken over a window of 30 samples to smooth out switching overheads). Two SQLR configurations are shown: Case 1 ($\theta = 1, \beta = 0.01$) and Case 2 ($\theta = 10, \beta = 0.001$).

can prioritize resource cost over blocking, but still achieve reasonably low blocking rates (Case 1).

4.4.4. Service times

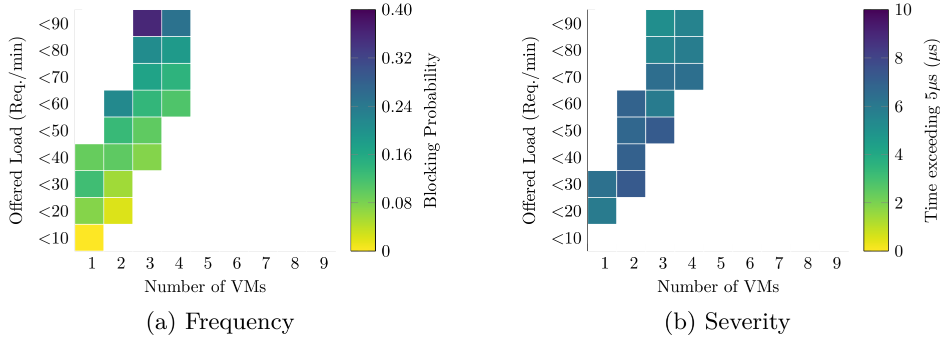
The distribution of service times is shown in Figure 4.23. We obtain the service time per job by dividing the service time of each request by the corresponding number of proof-of-work iterations it generates. These response times include the administrative overhead that the hypervisor incurs to switch between the host and guest while managing VMs. It also includes context switching between user mode and kernel mode of the corresponding operating systems.

This overhead increases with the number of VMs being administered as well as with how often they switch context. Dynamic scaling, which entails starting up and shutting down VMs, exacerbates the latter. The combination of these factors affects SQLR's Case 2 greatly, as it incurs higher penalties for blocking than resource usage. This is because the policies of SQLR's Case 2 implicitly employ more VMs. Therefore, SQLR's Case 2 closely follows the static over-provisioned case with 10 VMs, that incurs high administrative overhead throughout.

However, the over-provisioned scenario still provides the ideal case with the lowest-variance (highly predictable) service times. Both configurations of our scheme closely approach this ideal case with about 96% of the requests being served within 5 μs per job, compared to 95.5% for the over-provisioned case, 86.5% for the EKF case and 87.7% for RLPAS.

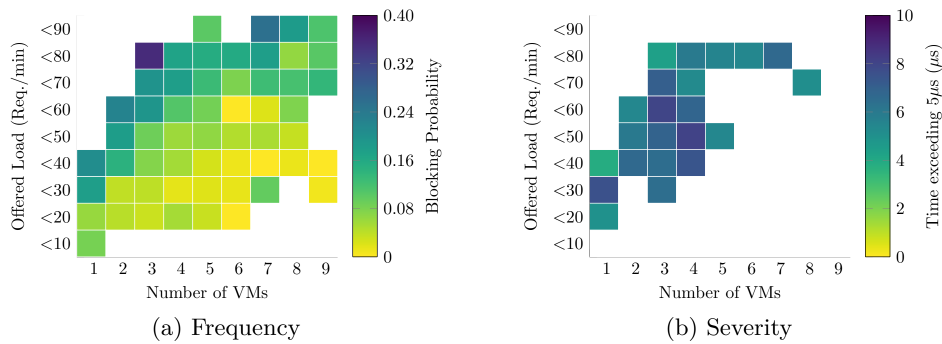
Moreover, for SQLR's Case 2 ($\theta = 10$, $\beta = 0.001$), the improvement in the proportion of responses within the cutoff service time of 5 μs is only marginal, compared to the more cost-focused Case 1 ($\theta = 1$, $\beta = 0.01$). This is despite the extra amount of resources deployed in Case 2, and is primarily due to the additional administrative overheads incurred.

In order to compare the scaling schemes without the biasing effect of the administrative overhead, we carry out a process akin to noise filtering in communication systems. We do this by first obtaining the average service times over two-minute intervals, and then applying a moving average filter having a window of 30 samples. Since context switching happens in the order of clock cycles, these two operations over intervals that are orders of magnitude longer than a clock cycle spread the cost of switching overhead over time and smooth the curves. When we apply the operations stated above to the service time per job, we obtain the results depicted in Figure 4.24. Both SQLR configurations closely follow the over-provisioned policy with the ideal response times. At low workload (hours 0-7 and 17-24), the administrative overhead to maintain a large number of VMs outweighs the gain of better service times resulting from the use of extra resources. Over these intervals, our scheme performs slightly better than the 10-VM case by provisioning fewer VMs. Conversely, the EKF-based scaler still under-performs: the single VM it provisions over these intervals is not sufficient to meet the demand within the cut-off service time. The RLPAS scaler, owing to its abrupt scaling behavior, exhibits response times that



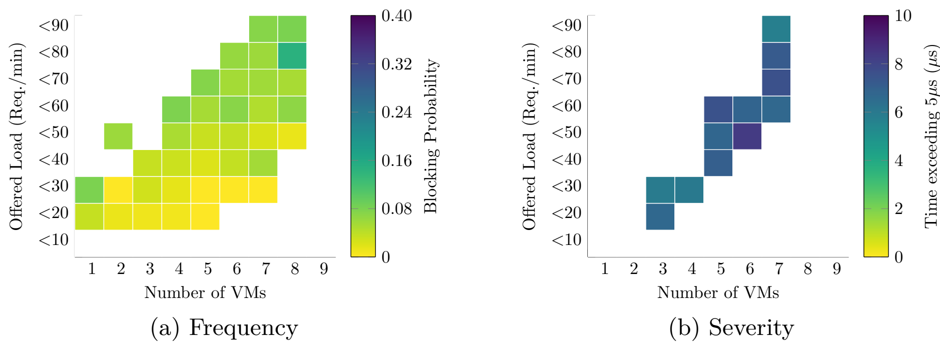
(a) Frequency (b) Severity

Figure 4.25: Soft Blocking Probability for the EKF scaler



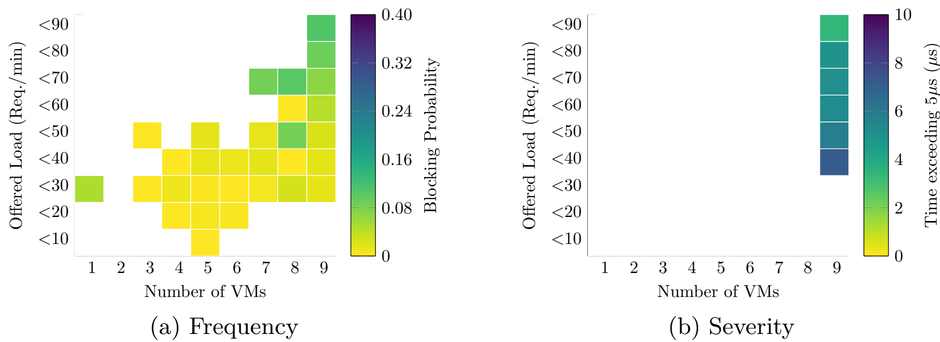
(a) Frequency (b) Severity

Figure 4.26: Soft Blocking Probability for the RLPAS Scaler



(a) Frequency (b) Severity

Figure 4.27: Soft Blocking Probability for SQLR's Case 1 ($\theta = 1$, $\beta = 0.01$)



(a) Frequency (b) Severity

Figure 4.28: Soft Blocking Probability for SQLR's Case 2 ($\theta = 10$, $\beta = 0.001$)

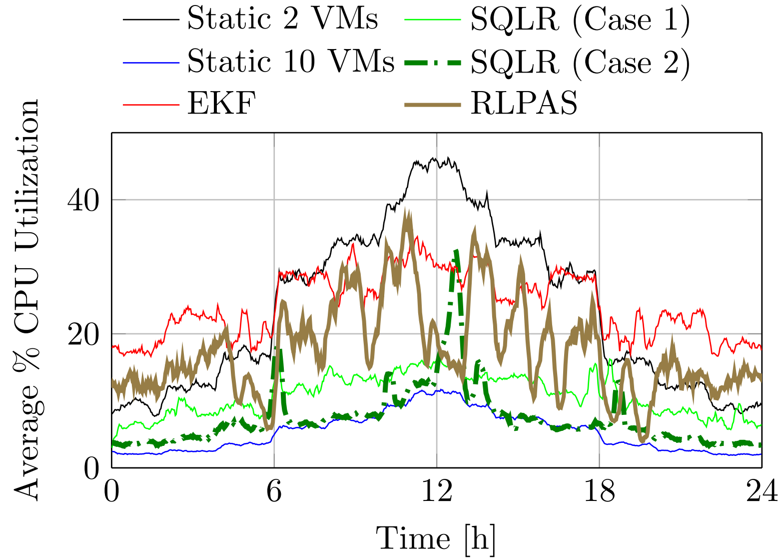


Figure 4.29: Moving averages of CPU utilization (taken over a window of 10 samples to smooth out switching overheads). Two SQLR configurations are shown: Case 1 ($\theta = 1, \beta = 0.01$) and Case 2 ($\theta = 10, \beta = 0.001$).

oscillate about those of the more stable EKF scaler.

This marked difference in response times, owing to differences in the scaling mechanisms, is clearly depicted in Figures 4.25 to 4.28, where we compare the *soft blocking* performance (the proportion of admitted requests whose service times extend beyond our cut-off of 5 μs per job). In these heatmaps, we consider only regions with statistical significance (30 or more responses). Moreover, the white region indicates resource allocation choices that remain unexplored for the considered input load. For the severity heatmaps of Figures 4.25b, 4.26b, 4.27b, and 4.28b, the white region indicates those allocations leading to service times within the limit of 5 μs. The offered load values on the y-axis indicate the upper bound with the value immediately below indicating the lower bound, e.g., “<20” indicates the interval [10,20) requests per minute. Therefore, in panels (a) of Figures 4.25–4.28, the best behavior is shown as yellow hues, as opposed to unwanted behavior (blue hues). Moreover, a larger number of yellow-colored cells denotes greater scaling agility through appropriate system states. Conversely, panels (b) of Figures 4.25–4.28 convey best scaling behavior both through yellow hues and through the presence of a greater number of white cells.

As depicted in Figure 4.25, the EKF scaler employed by [25] is prone to overruns even under light loads, given that it is very conservative in allocating extra VMs. As a result, this increases the strain on the few active ones. Figure 4.26 shows that RLPAS, instead, is prone to more widespread overruns across all loads. This is because it prematurely scales in, even at high loads, after transients in workload measurements. In our scheme, whose responses are shown in Figures 4.27 and 4.28, a significantly smaller proportion of

service times exceed the cut-off time (particularly at moderate to high offered loads). In fact, our scheme is more sensitive to abrupt workload changes, and assigns resources in a more agile fashion compared to the EKF-based scheme. The effect of the EKF is to quench scaling decisions, especially in the presence of short-lived workload bursts.

Further, comparing the two configurations of our scheme shown in Figures 4.27 and 4.28, the provisioning policies of Case 2 result in fewer instances of soft blocking than Case 1. This is because Case 2 provisions more VMs on average, which increases the likelihood of operating them at lower CPU loads. This allows more task admissions and lower service times. High severity (particularly in Case 2) comes from exploratory actions at high demand, whereby our scaler momentarily scales in. However, by evaluating the sub-optimality of these actions, our weighted fair guided exploration quickly scales out, as is evident around hours 10 and 12 in Figure 4.20c.

4.4.5. CPU utilization

The average CPU utilization over the duration of the experiments is shown in Figure 4.29. As before, we apply a moving average filter with a window of 10 samples to each curve. SQLR Case 1, SQLR Case 2, and the over-provisioned case with 10 VMs result in average utilization levels below 20%. The EKF scaler and the under-provisioned case result in utilization levels above 25% during the peak period. The RLPAS scaler results in a few excursions into utilization levels above 25%. Note that no curve passes 45% utilization in Figure 4.29, as the AC learned not to admit tasks beyond this limit regardless of the employed scaling scheme.

The utilization trends closely follow the service times shown in Figure 4.24, emphasizing the high correlation between these metrics. This confirms the suitability of CPU utilization as a metric to define the state of both the AC and the scaling RL.[4]

4.4.6. Summary of results

We summarize a comparison of the performance of the scaling schemes in Table 4.2. Here, we take the static over-provisioned case (with 10 VMs) as a reference benchmark with 0% blocking. We measure resources in terms of VM-hours. The tradeoff between resource use and service availability is apparent: schemes that procure lower blocking rates use up more resources to do so. Both configurations of our scheme achieve service times comparable to the over-provisioned benchmark with 10 VMs.

With particular regard to Case 2, our experiments show that we can reduce the amount of provisioned resources by about 20% with less than 1% overall service unavailability (due to blocking), while delivering similar response times close to those of an over-provisioned

[4]Conversely, system memory allocation is not an expressive metric: in all of our experiments, all scaling agents allocate about 360 MBytes of memory, with negligible oscillations around this value.

Table 4.2: Summary of Results

Scheme	Resources saved	Requests served with <5% blocking	Requests served with $<5\,\mu s$ per job
Static 10 VMs	0.00%	100%	95.47%
Static 2 VMs	80.00%	64.86%	84.89%
SQLR Case 1	55.13%	91.50%	95.60%
SQLR Case 2	20.54%	99.17%	96.17%
EKF [25]	80.15%	65.83%	86.47%
RLPAS [22]	67.30%	71.53%	87.69%

system. SQLR therefore achieves the delicate balance between saving resources and maintaning high performance.

4.5. Discussion

In this chapter, we have presented an agile horizontal scaling system, SQLR, that learns the most appropriate horizontal scaling decision to make under highly dynamic workloads, and without any fore-knowledge of the underlying system configuration. We show that our modified Q-learning scheme enables our system to learn multiple policies and re-use any applicable knowledge to new workload profiles exhibiting previously encountered characteristics.

SQLR progressively optimizes its policy by tuning the tradeoff between resource cost and service availability. These constraints come from the CSP after proper determination from their business processes. Such high-level objectives make SQLR easily configurable and adaptable to any cloud application, as no domain-specific knowledge is required. We compare our proposed scheme to different state-of-the-art scaling systems, and show that our scheme achieves better performance, similar to that of an over-provisioned system.

As with most machine learning-based schemes, our scheme is subject to a training overhead. However, because of its capacity for contextual knowledge re-use, it can be trained offline with representative workloads. Also, given our weighted fair exploration mechanism, any subsequent residual learning can be done in production workloads with a much reduced risk of poor decisions in the process. We show that even prior to full convergence, our scheme performs practically as well as the unconstrained resource benchmark (static over-provisioning).

In the following chapter, we extend SQLR container provisioning and discuss the adaptations that need to made to achieve acsla objectives.

5 Container based provisioning

The adoption of containers is growing at a rapid pace. This is mainly due to their comparatively simpler management with respect to VMs, and to the efficient resource utilization they enable. Moreover, the performance of containers is comparable to that of native computing platforms in terms of throughput and CPU utilization whereas VMs typically impose non-negligible overhead [59, 60]. Characteristics such as quick deployment, short boot time [60], easy network management and the use of layered, small-sized container images have encouraged container adoption in global systems like the Google Cloud Platform (GCP) and Amazon Web Services (AWS) [61].

The features mentioned above make it possible to achieve rapid elasticity (the ability to quickly scale the amount of allocated resources according to workload intensity) using containers. Managing container deployments in cloud environments is still an open issue that mainly involves provisioning and scaling strategies.

The performance of a container depends not only on how many CPU threads it uses but also on which core of the CPU these threads are executed. It is also contingent on the level of contention by other applications on the same CPU. In order to make the cost calculation transparent to the user, such inter-dependencies must be avoided and the relationship between the quality of service delivered and the amount of used resources needs to be clarified. Existing container provisioning platforms delegate the CPU scheduling to the operating system. Given that current operating systems are not hyper-threading aware, interference among running containers cannot be completely avoided, and load is typically distributed sub-optimally across CPU threads. This results in unpredictable service times.

Containers are currently managed using software platforms such as HPA by Kubernetes or Docker Swarm, which offer reliability by default [15]. However, some limitations exist regarding performance guarantees and adaptability to rapid changes in the operating environment. For instance, currently adopted solutions favor over-provisioning policies over rapid elasticity involving system adaptation. Capacity allocations are statically sized to serve peak loads, so resources remain underutilized most of the time. We however argue that over-provisioning is neither efficient nor strictly

needed. We show that automatic scaling is a better option provided that containers can be mapped onto hardware resources to avoid resource access conflicts.

To this end, we implement *core pinning* to ensure that a CPU core is reserved for a container, thereby forestalling contention side-effects due to hyper-threading. This approach also simplifies pricing models by making the cost of a container proportional to that of a CPU core. With regard to provisioning, we show that scaling the number of containers allotted to an application yields better performance than scaling the amount of resources allotted to a single container. Scaling the number of containers makes service time predictable, and thus provides the technical basis for, e.g., the stipulation of *SLAs* between service providers and customers.

Moreover, we implement an automatic scaling system that predicts the required amount of resources and proactively makes scaling decisions in order to maximize the application workload processing throughput and minimize the infrastructure allocation costs. Our automatic scaling subsystem is a Q-Learning agent. Since it is based on a model-free reinforcement learning technique, this agent can learn the operating environment autonomously and adapt its scaling policies without manual intervention.

The rest of this chapter is structured as follows. In Section 5.1, we discuss the challenges involved in container provisioning and propose our self-scaling provisioning solution in Section 5.2. We describe our experimental testbed in Section 5.3. We present and analyze our experimental results in Section 5.4. In Section 5.5 we discuss the main contributions of this chapter.

5.1. Container provisioning

In this section we explain how to avoid interference (i.e., contention in accessing shared computing resources) among running containers and explain how we operate to make service time predictable.

A Linux container is a group of isolated processes running on the host machine without any resource virtualization. A container can be granted an arbitrary amount of resources on the host machine: the amount of actually available resources depends both on the host capacity and on the resources allotted to other containers. For this reason, containers running on the same host will interfere with each other.

The use of hyper-threaded CPUs results in additional interference. Each CPU comprises multiple cores, each of which can run two threads. These threads share the hardware for the execution phase. Hyper-threading leads to a performance improvement for each core since it minimizes the impact of cache-miss interruptions. Unfortunately, such an architecture may also yield unpredictable performance, depending on which thread is used to run a process [62, 63]. In fact, as different threads in the same core share part of the architecture [64], execution performance is affected by other processes

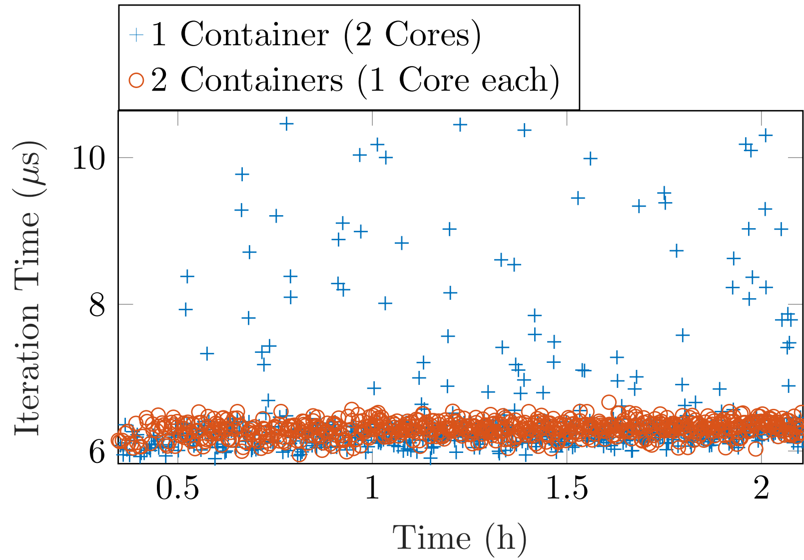


Figure 5.1: Vertical vs. horizontal scaling: two containers running on distinct cores provide more predictable performance than one container running on two cores.

in the same core.

The solution to these problems is the use of resource limitation in conjunction with core pinning, as demonstrated in [65]. Concretely, this means that we dedicate one or more (entire) cores of a CPU to a given container for its exclusive use. This configuration curtails any interference with other processes and eliminates the interference related to L1 caching mechanisms since each process stably runs on the same core. CPU core pinning also leads to efficient resource isolation [66], improved throughput and improved power utilization [67].

In Linux, both resource limitation and core pinning can be achieved by leveraging the `cgroup` feature. Using this feature requires the specification of the threads to be used for each container. To avoid disparity in hyper-threaded architectures, we select threads belonging to the same core. For the Linux distribution we use in our test-bed, this involves consulting the `cpuinfo` file.

Core pinning also clearly delineates the number of resources used as containers are mapped onto a known number of allocated cores. Therefore, following [68], the cost of running a container over k time intervals can be computed as:

$$\mathrm{Cost}(k) = \alpha \sum_{n=1}^{k} C_{n,n-1} \cdot (t_n - t_{n-1}), \tag{5.1}$$

where $C_{n,n-1}$ is the number of cores dedicated to the container during the interval $[t_{n-1}, t_n]$, and α is a configurable cost scaling parameter chosen by the Cloud provider. As regards scaling, two strategies are possible: vertical scaling and horizontal scaling.

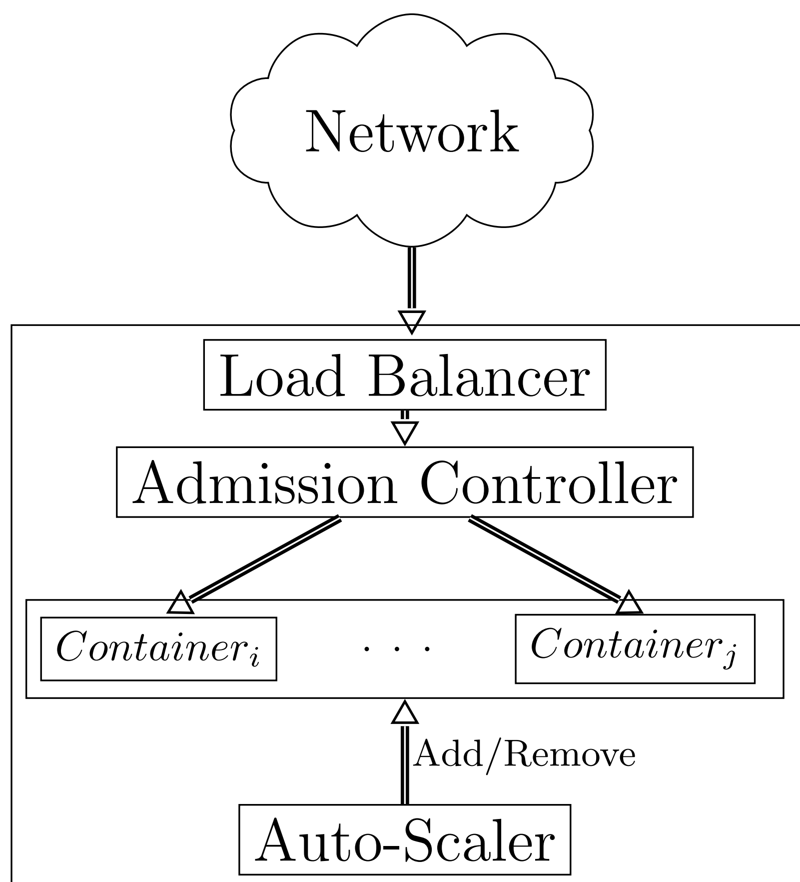


Figure 5.2: Proposed container auto-scaling architecture.

The former entails the addition or removal of cores from a running container while the latter involves instantiating new containers or decommissioning active ones without adjusting their resource allocations. If vertical scaling is used such that multiple cores are assigned to a container, even with CPU core pinning on hyper-threaded cores, Linux's Completely Fair Scheduler (CFS) might assign all processes to some cores leaving the rest in an idle state. This is because these schedulers are hyper-threading unaware and may introduce interference among competing processes on the same container. Horizontal scaling, instead, allows fine-grained resource management and prevents intra-container interference. This effect is exemplified in Figure 5.1 where the time per iteration of the double 256-bit bitcoin hash computation is shown for multiple such requests over a time period. The response times exhibited by two containers running on independent cores is markedly predictable compared to one container with two cores. In the next section we employ core pinning to spawn containers using different cores. In particular we allocate one core for each container.

5.2. Automatic provisioning system

Here we exploit the container-provisioning approach to build a system capable of optimizing resource utilization. Our system scales the number of allocated containers to align with the varying demand in order to minimize costs while maintaining a high level of service. As a result of employing core pinning as explained in Section 5.1, scaling decisions do not affect response time. A schematic diagram of our system is shown in Figure 5.2. It comprises three components: LB, AC and Auto-Scaler (AS). As suggested in [52], the LB component directs an admitted request to the active container reporting the most recent and lowest utilization rate. The AC component leverages CPU utilization statistics from the `cgroup` file-system to decide whether an incoming request should be handled. The AS spawns new containers or removes active ones as appropriate, depending on demand.

We now delve into our design of the AC and AS components. Our LB implementation is inherited from [52].

5.2.1. Admission controller

Considering the iterative double 256-bit bitcoin hashing algorithm as an exemplary cloud application, the behaviour of a container (with a dedicated CPU core) is shown in Figure 5.3. Each request triggers a different number of iterations. The response time, normalized by the number of iterations, is shown to be independent of the particular request's characteristics.

The plot in Figure 5.3 also shows that the relationship between the amount of allotted resources and SLA terms of service (such as the minimum response time) is not necessarily linear. In particular, there exists a discrepancy between CPU utilization (as reported by the operating system) and the actual occupied capacity of the core due to the use of hyper-threading [62]. This disparity is because the operating system considers two threads of the same core as two independent cores. Bearing this in mind and using the operating system metrics, a container exhibits a tri-stable CPU behavior: 0% CPU resource utilization when idle, 50% while continuously busy on a single core, 100% when continuously hyper-threading on the same physical core. The latter case points to saturation and unpredictable service times. The service time remains predictable only when a single thread (50% of a hyper-threaded core) is busy, as shown in Figure 5.3. The above suggests that new requests should be admitted only when the container is not busy (0% utilization).

We finally remark that transients have a non-negligible impact on the correctness of admission decisions. Specifically, an admission error may occur in two cases: (*i*) a request was just assigned to the container, but the reported CPU usage value is still close to 0%, triggering the admission of an (otherwise undesirable) additional request; and (*ii*) the container just finished serving a request, but the reported CPU usage value is still

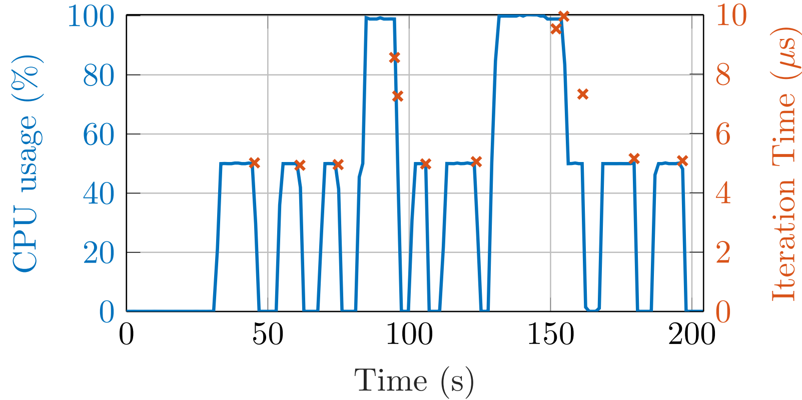


Figure 5.3: Relationship between CPU utilization and service time. When the reported utilization is $\leqslant 50\%$, the iteration time is predictable.

close to 50%, triggering the rejection of a request that should have been admitted. These transients in reported CPU utilization are typically short-lived such that the inter-arrival time of requests, even at peak time, is much longer in comparison. However, in order to further reduce the likelihood of the first event, a new request is admitted if the reported value is $\leqslant 25\%$. In practice, this solution also reduces the likelihood of the second case.

5.2.2. Auto-scaler

The purpose of the AS is to allocate the minimum number of containers that is commensurate to the demand while still minimizing the number of dropped requests as reported by the empirical blocking probability. The scaling mechanism we employ is similar to the Q-Learning paradigm presented in Chapter 4 as shown in Figure 5.4.

We designate the permissible scaling actions as: $-n$ (remove n containers), $+n$ (add n containers) and 0 (maintain the existing number of containers). For $n = 1$, we can therefore describe the action space as follows:

$$A = \begin{cases} [-1; 0; +1] & \text{if } 1 < N_t < M; \\ [0; +1] & \text{if } N_t = 1; \\ [-1; 0] & \text{if } N_t = M; \end{cases} \tag{5.2}$$

where N_t is the current number of active containers and M is the maximum number of containers that can be allocated.

The state space is described by the triplet set of the number of containers, and the utilization in the previous and current epoch. We quantize the latter two components of the state space into nine levels, from 0% to 45% in steps of 5%. The last level encompasses the range from 45% to 100% utilization. The latter detail is required because the admission control function ensures that utilization never exceeds 50%. Fine-grained

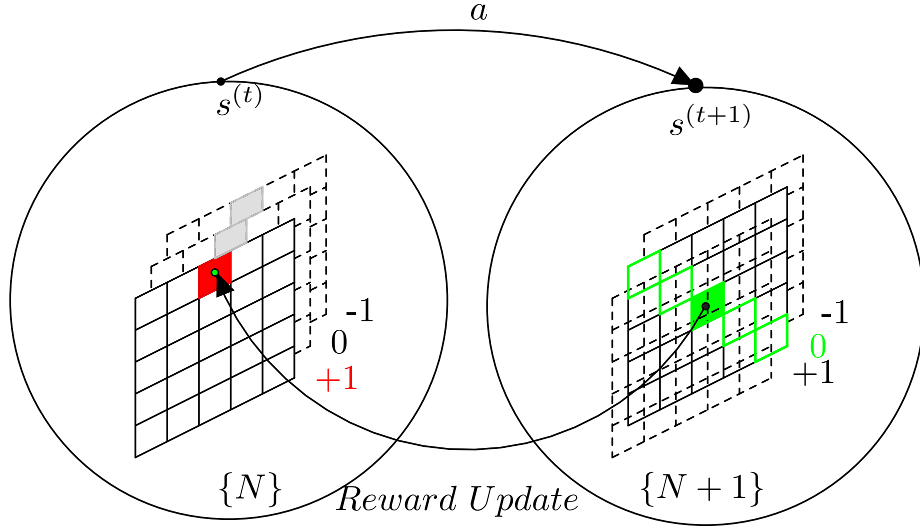


Figure 5.4: Short-Term Memory Q-Learning mechanism used for the auto-scaler, cf. Chapter 4. This schematic shows a scale out operation where in the epoch given by the interval $[t, t+1)$, action a increases the number of containers from N to $N+1$ and the state transitions from $s^{(t)}$ to $s^{(t+1)}$.

quantization of the state space is unnecessary as the action space is restricted to 3 discrete actions as presented in Equation (5.2). Higher levels of quantization would increase the training time (owing to the curse of dimensionality) with little benefit to the quality of scaling policies learned. The reward function (R_{sqlr}) consists of a penalty for blocking (R_{blk}) and another based on the number of containers provisioned (R_{res}):

$$\begin{aligned}
R_{\text{sqlr}} &= R_{\text{blk}} + R_{\text{res}} \\
R_{\text{blk}} &= \begin{cases} R_{\min}, & \text{if } P \leqslant P_{\text{apt}} \\ \theta \left(P_{\text{apt}} - P\right), & \text{if } P > P_{\text{apt}}, \end{cases} \\
R_{\text{res}} &= \beta(1 - N_t)
\end{aligned} \tag{5.3}$$

where P is the actual blocking rate, P_{apt} the acceptable level of outage as per the SLA, $R_{\min}$ is a small positive reward assigned to the agent when it keeps within the acceptable limits of service outage due to blocking, θ is the weight given to outage exceeding the acceptable level and β is the weighted cost incurred by the provider in deploying containers to handle client requests.

Compared to this approach given in Chapter 4, we simplify the mechanism in order to expedite learning. In particular, we set $\theta = 1$ so that only β is adjusted. The ratio of the two influences the policies learned.

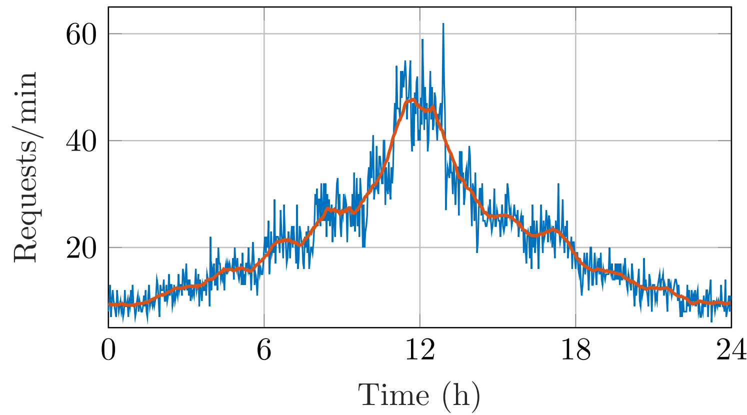


Figure 5.5: Traffic profile observed during the Docker experiments. Traffic rates are taken over two-minute windows. The red line is the moving average over 30 samples.

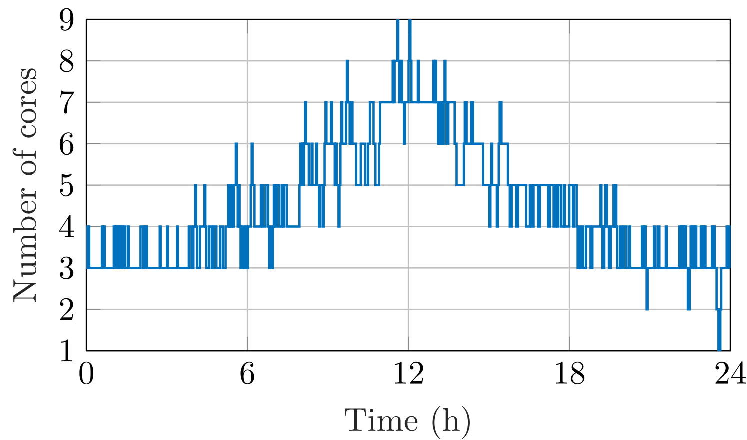


Figure 5.6: Scaling decisions taken by Auto-Scaler algorithm during the Docker experiment (using $\beta = 0.02$).

5.3. Experiment setup

We implement our provisioning scheme using Docker containers on a Dell T640 server with 20 hyper-threaded cores running Ubuntu 18.04. In order to expedite learning, we choose $n = 1$ (see Section 5.2.2) which effectively reduces the size of the action space. We implement CPU pinning and limit the maximum number of containers (M in Section 5.2.2) that can be provisioned to 9. This ensures that server capacity is never exceeded and that the host processes run on an independent core. The AC and LB functions are implemented as python applications and run on the host core.

The server is connected to client PCs in an isolated LAN via a high-speed switch. Bash scripts on the client PCs spawn requests to the server with varying frequency at different hours of the day to mimic peak and off-peak periods of typical real-world traffic profiles.

A 24-hour cycle is split into hourly periods as shown in Figure 5.5. Each period has inter-arrival times following a discrete distribution $\lambda \backsim U(0, \lambda_{\max})$. By varying $\lambda_{\max}$ we create a suitable peak/off-peak profile. The use of a uniform distribution ensures high entropy in order to evaluate the robustness of the schemes in challenging conditions. As mentioned in Section 5.2.1, we deploy the double 256-bit bitcoin hashing algorithm as our cloud application. Each admitted request triggers a different number of iterations, which makes it possible to mimic the diverse complexity of cloud applications.

5.4. Results

We compare our provisioning scheme with the Google Horizontal Pod Autoscaler (HPA) for Kubernetes, a widely used container management tool. We also benchmark our scheme with static over-provisioning and under-provisioning. As comparative measures, we consider the following metrics:

1. Saved cost: cf. Equation (5.1), the difference in cost between employing the maximum amount of resources throughout and using an auto-scaling algorithm to provision variable amounts of resources,

2. Service time: the time taken to process a request normalized by the number of iterations triggered by it, measured at the server side in order to exclude network effects,

3. Blocking rate: a measure of service availability defined as the percentage of dropped requests with respect to those received by the server.

5.4.1. Docker experiments

We initially test our scheme in a Docker environment. With reference to the offered load presented in Figure 5.5, our scheme generates the scaling profile shown in Figure 5.6. With the parameters we have adopted ($\theta = 1$ and $\beta = 0.02$), the system learns to act quite aggressively with respect to blocking and encourages the provisioning of additional containers even with modest rises in the traffic profile. A higher setting of β would result in a stiffer reaction of the scaler, and would likely result in higher blocking rates. With this setting, our solution achieves 51% saved cost.

The performance in terms of blocking rate is shown in Figure 5.7. Our scaler achieves blocking rates that are comparable to the over-provisioned case with 9 containers throughout the 24 hours. It considerably outperforms the under-provisioned case in which only 4 containers are statically deployed and no adaptation is enforced over time. The saved cost for the under-provisioned case stands at 55% which is only marginally higher than that of our auto-scaler but with much poorer service availability, especially at peak

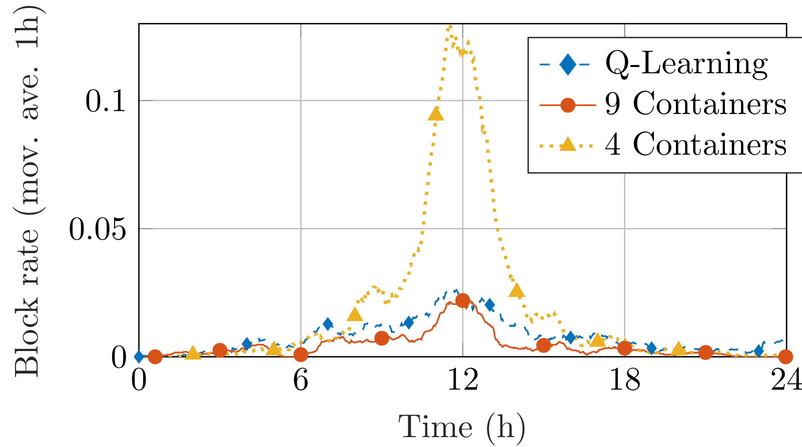


Figure 5.7: The blocking rate observed over the time during Docker experiments in terms of rejected requests per second.

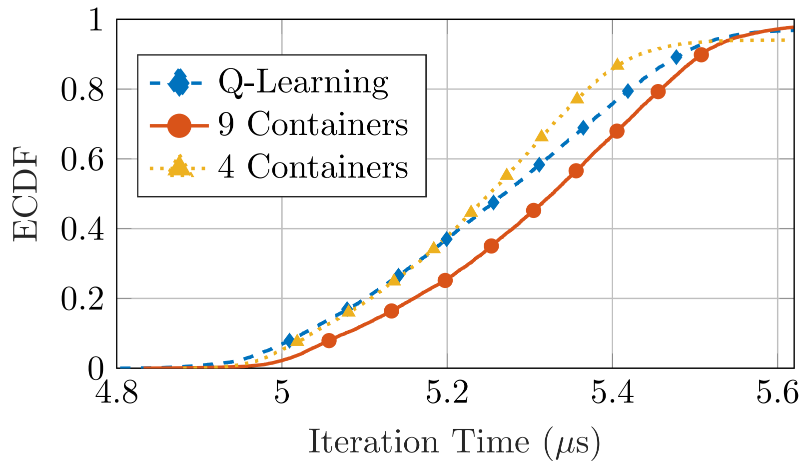


Figure 5.8: Empirical CDF of the service time for Docker experiments.

traffic. From Figure 5.8, it is clear that the number of containers deployed has an effect on the service time. This is due to the fact that container processes share the L2 and L3 cache memory. The greater the number of active containers the more pronounced the impact. For this reason, the over-provisioned case with 9 containers exhibits slightly higher service times, whereas the under-provisioned case with 4 containers yields the lowest service time. Our scaling solution suffers a small deviation from the low service times for about half of the cases owing to the instances when it provisions more than the benchmark 4 containers at peak traffic. However for all cases considered, the maximum difference in service times is small with respect to the minimum values observed, i.e., the difference is less than 0.1 μs, for more than 93% of the cases. This is because admission control ensures that the system only rarely reaches saturation.

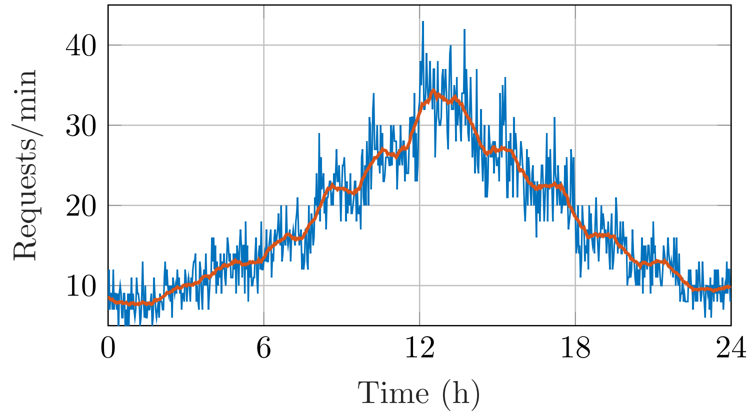


Figure 5.9: Traffic profile observed during the Kubernetes experiments. Traffic rates are taken over two-minute windows. The red line is the moving average over 30 samples.

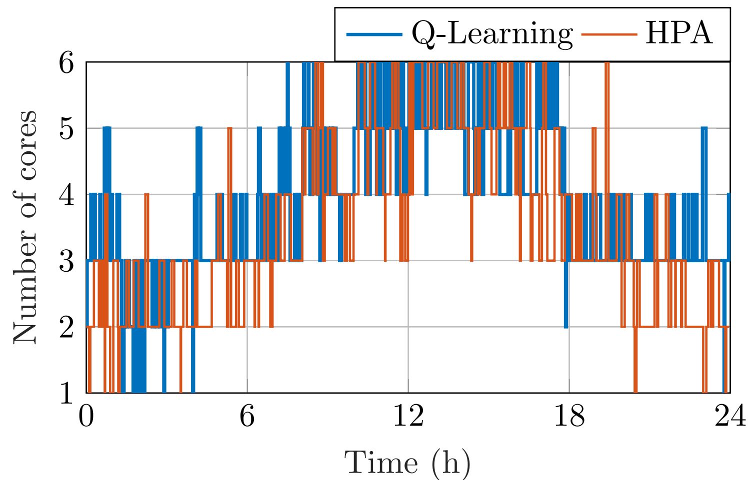
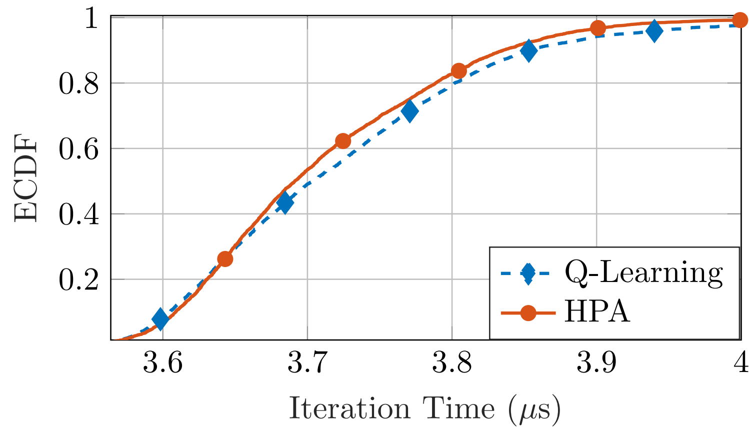


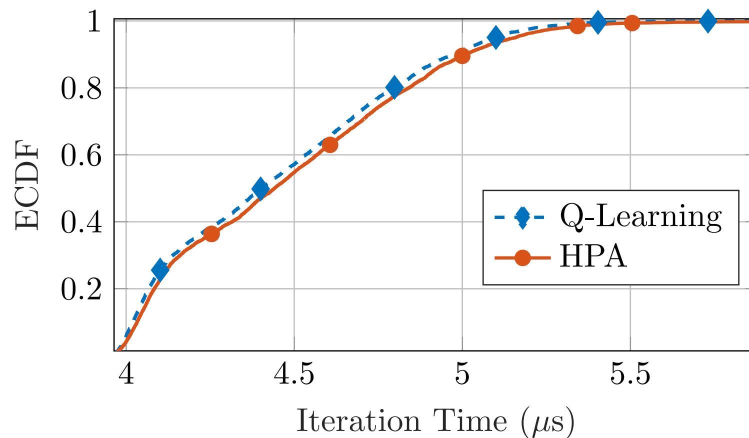
Figure 5.10: Scaling decisions taken by the AS algorithm during the Kubernetes experiment (with $\theta = 1.0$ and $\beta = 0.02$).

5.4.2. Kubernetes experiments

We now compare the performance of our scaler against the commercial HPA for Kubernetes. We re-run the experiments with the traffic profile shown in Figure 5.9 using two computers with different specifications. Our scaler autonomously learns the appropriate operating conditions for each computer to trigger the addition or removal of containers. HPA however requires that the threshold be set as an external input. Such a setting is often a trial and error process and is both application and configuration dependent. To obtain comparable results to our scaler, we set this threshold as 28%. The comparison between the decisions of our scaler and the ones made by HPA are shown in Figure 5.10. Two different servers are used in these experiments to guarantee different service times. Even in this scenario, our provisioning scheme achieves predictable service times, albeit different given the difference in compute power. In this case the less powerful



(a) Master node



(b) Slave node

Figure 5.11: CDF of the service time for the Kubernetes experiments.

compute engine (slave node) determines the service time benchmark.

While the two schemes save about the same costs and achieve similar results in terms of service time, as depicted in Figure 5.11, their blocking performance differs. Our approach based on CPU core pinning and Q-Learning largely outperforms HPA in terms of blocking rate, as shown in Figure 5.12.

5.5. Discussion

In this chapter, we have demonstrated the consistent performance achieved by implementing CPU core pinning and horizontal scaling when compared to vertical scaling with hyper-threaded cores. We show that CPU core pinning simplifies the pricing models for cloud providers by facilitating an easy mapping between actual resources used and

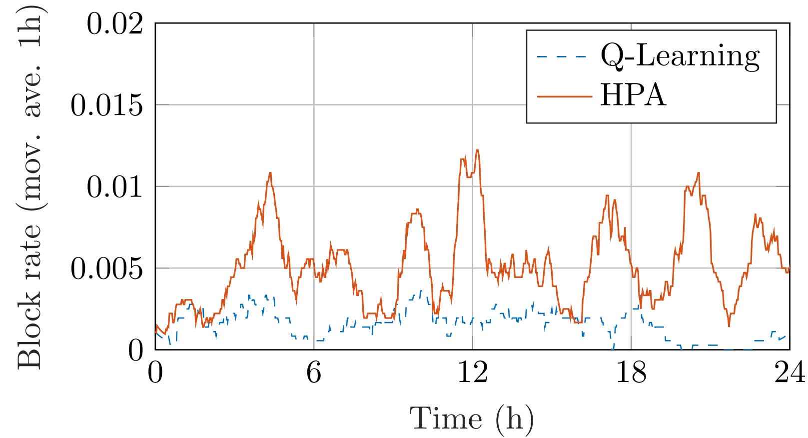


Figure 5.12: Blocking rate observed over time during Kubernetes experiments in terms of rejected requests per second.

container resources assigned to tenants. Although our scheme curtails the application of hyper-threading and its advantages, the benefits of predictable and consistent high performance outweighs this disadvantage by far.

We have also demonstrated the superior performance of the SQLR scaling scheme when compared to the HPA for Kubernetes which is widely adopted in container provisioning platforms. Our Q-Learning scheme attains predictable response times in the face of highly dynamic traffic without the need for manual threshold setting. It is able to learn the appropriate scaling triggers without prior knowledge of the system configuration or the cloud application.

In the second part of the thesis, we consider a special case of provisioning with regard to a high mobility low latency application. In this instance, where the resources are located on the network also becomes a significant consideration. We choose platooning as the quintessential application given that it exhibits both high mobility and has stringent latency requirements.

Part II

Multi-access Edge Computing (MEC) Provisioning for Low Latency Applications

6 Background and Related Work

Intelligent Transportation Systems (ITS) of the future aim to increase road throughput, improve safety and reduce emissions from vehicles [69, 70]. A key proposal of ITS is platooning, the coordinated driving of vehicles with short spacing between them. To realize platooning, legacy schemes have relied so far on Vehicular Ad hoc NETworks (VANETs). These networks are however susceptible to high packet losses, as they rely on contention based media access control [71]. Given the latency sensitive nature of platooning [72], VANETs cannot therefore safely support large platoons.

Moreover, to maintain speed-independent spacing between vehicles in such platoons, not only is it necessary that the vehicles receive status beacons from their predecessors, but all vehicles also need to receive communications from the platoon leader [73]. For long platoons, keeping all vehicles connected to the platoon leader may be challenging, owing to turns or obstacles along the road, which may prevent line-of-sight communications [71]. This exacerbates losses and hinders the information transfer required to maintain the platoon.

With the advent of MEC, each vehicle in the platoon can potentially have a low-latency connection to a controller hosted at the network edge. The use of scheduling and robust forward error correction in 5G means that, unlike contention-based protocols used in VANETs, packet losses are not as pronounced. However, delivery delays remain a key challenge [74]. There have been some recent efforts [75–77] to explore how the MEC architecture can be leveraged for Vehicle-To-Infrastructure (V2I) platooning. However, some key challenges have not been adequately addressed, including: (*i*) the development of robust controllers that can tolerate high delays; (*ii*) how to deal with out-of-order reports from platoon members; and (*iii*) how to intelligently migrate the controller across edge nodes to minimize latency as the platoon moves [78].

In this work, we address these challenges by modifying the platoon controller to handle V2I communication issues and develop a context-aware Q-learning migration scheme that deploys the controller in the most suitable location. Q-learning is a model free reinforcement learning technique in which an agent learns the action-value of a state by

trial and error initially before settling on the most rewarding action in the long run [50]. We thereby substitute a controller in each platoon member with a centralised one, which can be migrated from one MEC host to another according to the policies learned as the platoon moves.

Concretely, the contributions included in this chapter are: (*i*) we design a migration agent for the centralized control of a platoon from the MEC, based on a Q-learning algorithm that, unlike previous approaches, incorporates context-awareness; (*ii*) we modify CACC to estimate and compensate for network latency and messages delivered out of order, which is generally not required for direct vehicle-to-vehicle (V2V) communications; (*iii*) we define and study how multiple Q-learning agents can cooperate for rapid policy convergence with zero synchronization overhead; (*iv*) we enhance and combine existing vehicular and network simulation frameworks, through which (*v*) we evaluate the performance of intelligent controller migrations for MEC-assisted platooning, in the presence of single or multiple Q-learning agents. Moreover, (*vi*) we introduce Slow down And spLiT (SALT), a safety overlay to the working of the MEC controller as a key additional contribution to our work presented in [3]. This novel technique progressively increases the gap between vehicles and reduces the overall speed as network and/or computing conditions deteriorate persistently. When the spacing is sufficiently safe, Automated Cruise Control is triggered to keep the vehicles moving in a convoy. When the vehicles move into an area where network and compute delays are sufficiently low, the platoon is reformed and progressively moves at the desired speed and spacing.

In the following section, Section 6.1 we consider the state of the art in platooning and application migration.

6.1. Related work

Platooning *per se* and the possibility to control vehicles from the MEC have already been addressed in the literature [72, 76]. Our interest focuses on making the controller aware of communication issues that can occur, and how to take advantage of the fluid architecture of cellular edge networks, which offer multiple options where to run and possibly migrate the platoon controller.

6.1.1. Controllers

Work on controllers to achieve cooperative driving dates back to the PATH project [79]. Though robust, this controller fits a peer-to-peer ad hoc network with short delays and good discipline in the order of communications among vehicles. The controller proposed in [80] only requires communications between proximate members of the platoon. While relaxing the stringent requirement of the platoon leader communicating with all members, it imposes a speed-dependent spacing between vehicles.

Other works such as [81] have made controllers more delay tolerant and robust to packet errors, by taking into account the topology of the platoon and by employing speed-dependent spacing as well as communication between the platoon leader and all members. This approach is markedly string-stable, but remains limited to small platoons, and may not scale well in a MEC-driven scenario.

We enhance the well-known controller presented in [79] to account for varying communication delays and disorderly reporting from platoon members. These adaptations make the controller better suited for use in V2I MEC-enabled platooning.

6.1.2. Service migration

Many state-of-the-art service migration schemes exist for MEC deployments [82]. However, only a subset of these fit latency-critical applications. The authors of [83] propose a random start placement on the available nodes which is then refined by prediction based on collected performance metrics. In [84], the authors propose a cognitive edge computing architecture supporting a service migration scheme. The scheme relies on repeated evaluations of the quality of experience of the users as they move in a network. Owing to its focus on human users, this work does not address the strict latency requirements of platooning.

The authors of [74] design a service placement algorithm leveraging Lyapunov optimization to decompose the problem. Each subproblem is then solved by Markov approximation. The resulting scheme tracks user mobility and locates the service at the MEC host that minimizes the delay and cost. Although this approach considerably improves latency, it does not tackle the typically time-varying computing capability of the MEC hosts. This is particularly important given the myriad services that a MEC node may host, which thereby affect the experienced latency.

This limitation is however considered by the authors of [85] who create an Adaptive User-managed Service Placement (AUSP) algorithm taking into consideration the compute delay, the communication delay and the switching cost to a given candidate MEC host. They then use a multi-armed bandit approach which estimates the total delay distribution of each MEC host. At each time step, the MEC host with the lowest estimated total cost is chosen. This approach is however susceptible to unnecessary migrations.

Our proposed scheme takes all these delay elements into account and also minimizes the number of migrations. In so doing, it also cuts down the migration costs. In the following chapter we discuss in detail the working principles of our proposed scheme and compare its performance to state of the art algorithms.

7 V2I platooning

In legacy ad hoc platooning systems, the platoon leader communicates its speed and acceleration to each platoon member [86]. Each member also receives the speed and inter-vehicle spacing of its predecessor, typically from a radar system. With these data, the controller in each member calculates and applies the appropriate acceleration. In the V2I-assisted platooning case, the vehicles only need to communicate their own speed, acceleration and distance from the preceding vehicle to the controller resident in the MEC host as depicted in Figure 7.1. From the figure, it is also clear that not all vehicles of the platoon may be served by the same cell tower, and hence packets will experience varying delays. Indeed, network latency and the variation thereof over time strongly impact this process.

As discussed in [87], the architecture of the network greatly influences the latency of a service. Given the proliferation of small cells and network densification expected in 5G, handover delays will be of particular importance in determining the performance of 5G services. Handover latency values in the order of a few milliseconds can be tolerated.

Further, owing to the fact that uplink transmission opportunities are subject to channel-dependent scheduling, there exists a non-trivial delay between the moment a packet is available for transmission at the on-board unit of a platoon member and the time it reaches the MEC controller. Therefore, the data in this packet will be slightly stale, particularly in regard to speed and inter-vehicle spacing. Instead, we can safely assume that acceleration values will remain coherent within a transmission window as suggested by the performance of automatic control in real world platooning experiments [88]. Current efforts to achieve sub-millisecond delays in next generation networks are still in their infancy [89] and as such we do not make this assumption in our work.

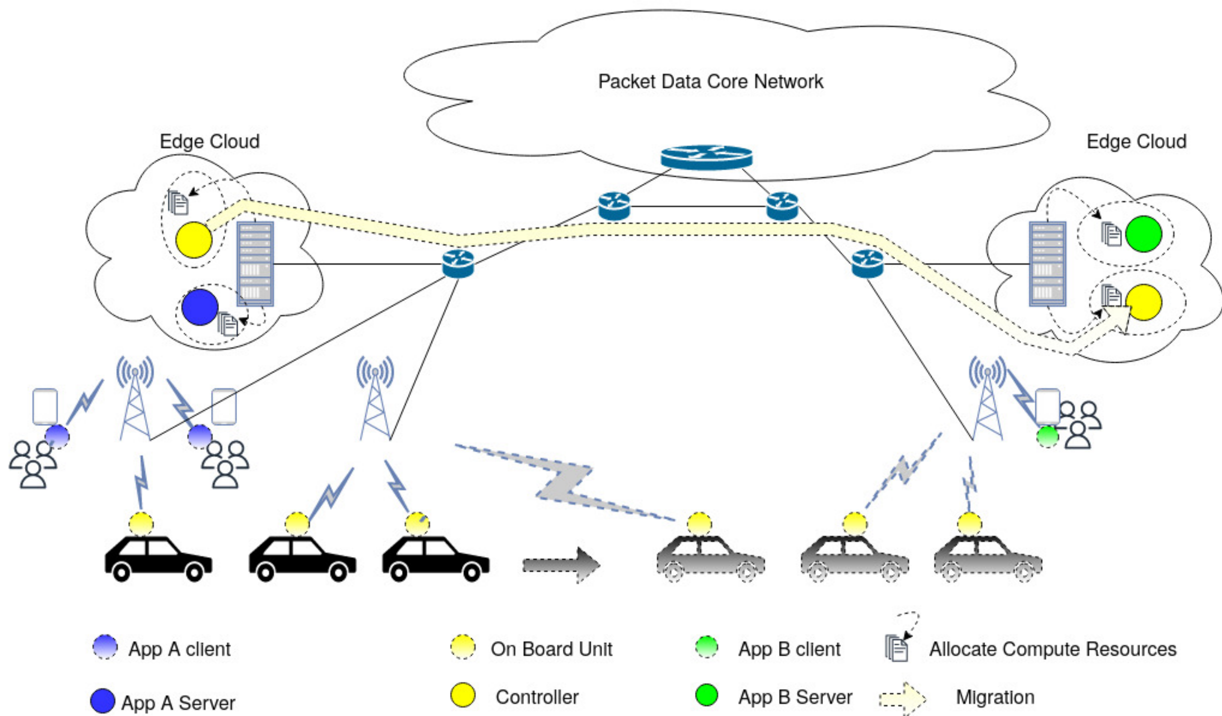


Figure 7.1: V2I Platooning on the network edge

7.1. Controller adaptations for V2I platooning

7.1.1. Controller operation adaptations

When porting CACC to the V2I context, we modify it to improve controller operation and cope with two key issues. Depending on when the cellular network grants a transmit opportunity to a platoon member, either or both of the following may occur:

1. consecutive packets from the same vehicle reach the controller in quick succession, owing to a delay that spans the interval of more than one periodic update;

2. a packet generated at an earlier time by a leading vehicle reaches the controller later than that of a following vehicle.

In the case 1, we implement a filter that keeps only the latest packet from a given vehicle, as it represents the most updated information. In case 2, as depicted in Figure 7.2, we implement a data collection window to receive the packets from all members of the platoon. The controller then uses these data to compute the acceleration directive for each vehicle.

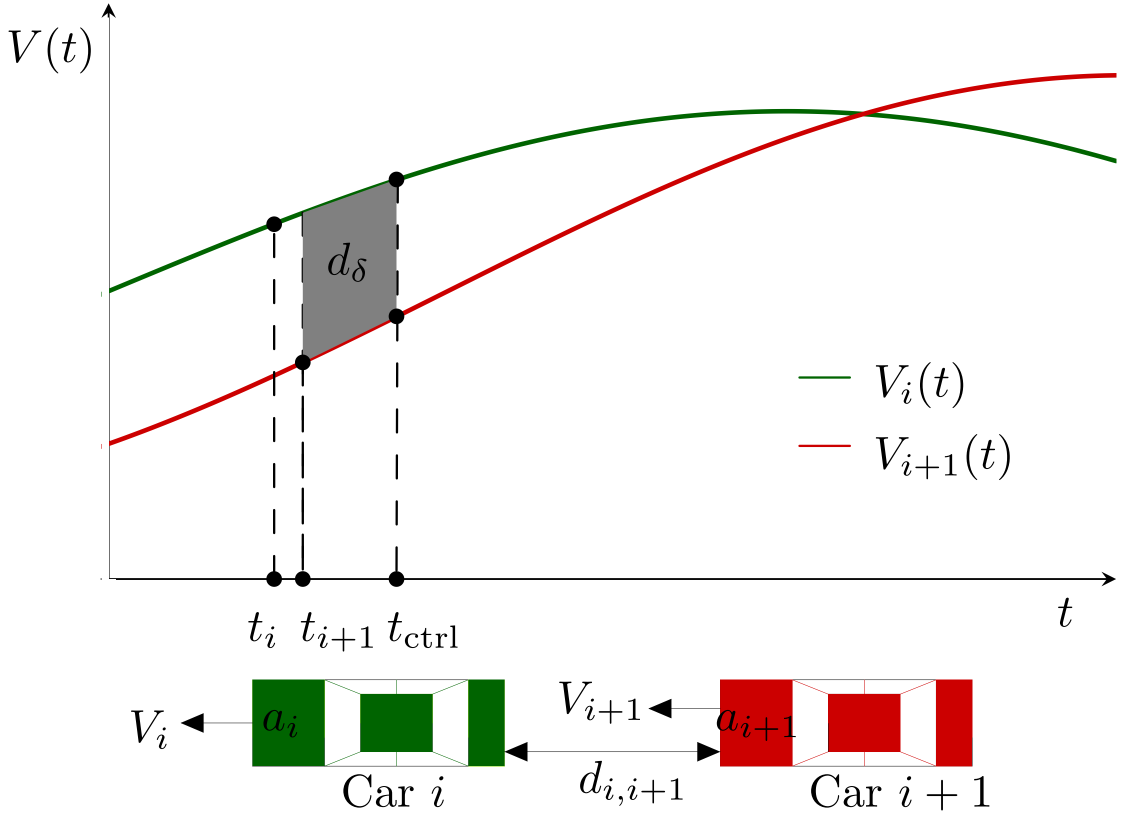


Figure 7.2: Controller delay compensation. At time t_i, car i generates the report packet with its own speed, acceleration and distance from vehicle $i-1$, and sends it to the MEC controller. Car $i+1$ does likewise at time t_{i+1}. At t_{ctrl}, the controller collates the data.

7.1.2. Latency compensation in the control law

The MEC controller has to account for the discussed latency values before computing the acceleration directives. Thus, we design the controller so as to update the speed assuming the previously assigned acceleration for the vehicle and update the spacing by estimating the distance travelled by the vehicle and by its corresponding predecessor. Given that such lags are in the order of up to several tens of milliseconds, these updates can be approximated as piece-wise linear functions as depicted in Figure 7.2. In particular, the speeds of vehicles i and its follower $i+1$ at control epoch t_{ctrl} can be computed based on speed and acceleration values sent with their freshest updates, at times t_i and t_{i+1}, respectively:

$$V_i(t_{\text{ctrl}}) \approx a_i(t_i) \cdot (t_{\text{ctrl}} - t_i) + V_i(t_i) \tag{7.1}$$

$$V_{i+1}(t_{\text{ctrl}}) \approx a_{i+1}(t_{i+1}) \cdot (t_{\text{ctrl}} - t_{i+1}) + V_{i+1}(t_{i+1})\,. \tag{7.2}$$

With the update generated at time t_{i+1}, the distance $\widehat{d}_{i,i+1}$ between the two vehicles is then estimated as:

$$\widehat{d}_{i,i+1} = d_{i,i+1} + \left(\int_{t_{i+1}}^{t_{\text{ctrl}}} V_i(t)\ dt - \int_{t_{i+1}}^{t_{\text{ctrl}}} V_{i+1}(t)\ dt \right). \tag{7.3}$$

7.1.3. Slow down And spLiT (SALT)

In some situations, the delay experienced by vehicle update messages may be too high to be satisfactorily compensated for as described in Section 7.1.2. In these cases, the MEC controller becomes an unreliable arbiter of the platoon, and it becomes safer to switch control back to the vehicles. In order to gracefully carry out this transition, the speed and spacing of the platoon need to be progressively adjusted as long as the untenable delay conditions persist. We do this by implementing a SALT overlay module inside the MEC controller. SALT leverages the total delay experienced as the trigger for its operation which consists in (*i*) adjusting the target distance used in Cooperative Adaptive Cruise Control (CACC) and (*ii*) switching between CACC and Adaptive Cruise Control (ACC) as needed. When under ACC, the spacing between the vehicles is typically larger given that it requires a fail-safe distance determined by a time headway greater than 1 s. Under such conditions, vehicles cannot leverage the slip-stream of the preceding vehicles to improve fuel efficiency. It is thus preferable to have the vehicles operating under CACC for as long as is feasible to do so.

As network and computing conditions vary, the interval between when a vehicle transmits a packet to the controller to when it receives a packet from the controller also varies. Simply capturing the upstream delay at the controller may not account for the computing and downstream delay. In order to get a complete picture of the delay, we include the creation timestamp of the vehicle report to which the controller is responding as part of the control packet. When the vehicle receives the control directive, it computes the difference between the current time and the creation timestamp, and sends it to the controller as part of the next report.

Given that these delay values are quite noisy, we first pass them through a low pass filter before using them as an input to our variable spacing function. If m is the number of average delays collected over an observation window[1] and $x[m]$ is the mth sample, at the nth observation window, the filtered delay $y[n]$ is given by

$$y[n] = \left(\frac{1}{2}\right)^{m+1} y[n-1] + \sum_{i=1}^{m} \left(\frac{1}{2}\right)^{i} x[m-i+1]. \tag{7.4}$$

We use this filtered delay to determine how to adjust the vehicle spacing. We choose

[1] We choose this to be 200 ms (twice the vehicular reporting interval), averages are collected over short intervals of 30 ms each corresponding to the data collection window. For a relatively large platoon, these values ensure that m is statistically significant.

$T_{\max}$ as reference upper bound on delay and $0 < \Psi < 1$ as the trigger point for when to adjust the spacing. The modified spacing $d[n]$ is obtained as follows

$$\Delta = \frac{1}{10}\left\lceil 10 \times \tanh\left(\frac{y[n] - T_{\max}}{T_{\max}}\right)\right\rceil, \tag{7.5}$$

$$d[n] = \begin{cases} d_\lambda, & \text{if } \Delta \leqslant \Psi \\ d_\lambda(1+\Delta), & \text{otherwise,} \end{cases} \tag{7.6}$$

where d_λ is the platooning target for the inter-vehicular spacing. Should delays exceed $T_{\max}$, the chances of successive packets with stale data arriving at the controller increases considerably. We choose the hyperbolic tangent since it has a smoothing effect on noisy inputs and its output is bounded in the interval [-1.0,1.0]. We use a simple geometric function with a ratio, $r < 1.0$, to adjust the speed at each observation window.

We choose $r \approx 1.0$ to avoid shocks that can cause string instability. The filtered delay is similarly used as the trigger. The speed at the at the nth observation window $V[n]$ is given by

$$V[n] = \begin{cases} V_{\text{ptn}}, & \text{if } d[n] = d_\lambda \\ \dfrac{V[n-1]}{r}, & \text{if } V[n-1] < V_{\text{ptn}} | \Delta \leqslant \Psi \\ V_{\text{acc}}, & \text{if } d[n] \geqslant 1.2\,\text{s} \times V_{\text{acc}} | \Delta > \Psi \\ V[n-1] \times r, & \text{otherwise.} \end{cases} \tag{7.7}$$

When $V[n] = V_{\text{acc}}$, vehicles in the platoon switch to a standard ACC that relies only on on-board sensors such as the radar. In Equation (7.7), 1.2 s is the time-headway we consider for ACC.

7.2. Q-learning agents for controller migration

As a platoon moves, the distance between the vehicles and the controller will change. Even if we resort to network slicing and assign dedicated resources to the platoon controller, at some point such controller may have to be migrated closer to the platoon in order to guarantee responsiveness. For this, we design a migration agent and assume that its actions are either to move the controller to a candidate MEC node or to leave it in its current location. With high probability, the chosen action will influence changes in communication delays impacting the delivery of acceleration directives to the platoon members. Consequently, the speed and the spacing between platoon members will be impacted. By defining the state as a combination of the processing capacity of a MEC host and of the platoon topology (relative spacing between platoon members), we can approximate the system as a state machine. In the following, we refer to the system as the "environment," in order to align with common reinforcement learning jargon [8]. The

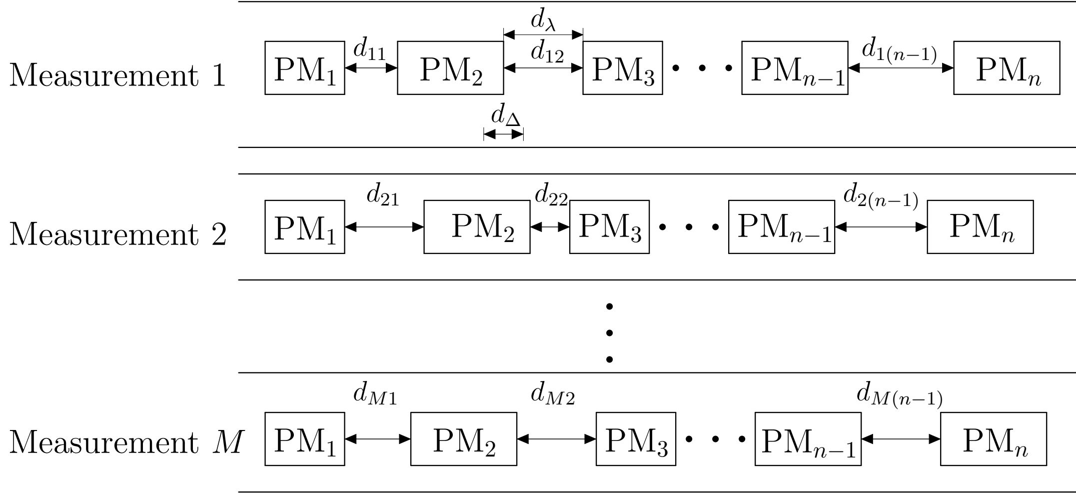


Figure 7.3: Platoon metrics as reported to the MEC Host. "PM" denotes a platoon member; d_λ is the required vehicle spacing; d_Δ is the spacing tolerance.

action of the migration agent will cause a state transition with some probability. We can therefore formulate the migration problem as a Markov Decision Process (MDP).[2]

7.2.1. Data for state context definition

A schematic showing the measurements received at the MEC is illustrated in Figure 7.3. The controller communicates at regular intervals to the vehicles to issue directives for braking and acceleration. The directive to migrate the controller from one MEC host to another takes place less frequently, hence several measurements of the platoon are received between two such directives. These measurements are used by the mobile edge application orchestrator (MEAO) to obtain platoon-specific state variables (or "context") for the migration agent. We derive the *platoon topology context* $\{\Gamma^{(-)}, \Gamma^{(+)}\}$ from the measurements depicted in Figure 7.3, with M measurements and n cars, as:

$$\gamma^{(-)} = \frac{1}{M(n-1)} \sum_{i=1}^{M} \sum_{j=1}^{n-1} \left\lfloor \frac{d_\lambda - d_{ij}}{d_\Delta} \right\rfloor \text{ for } d_{ij} \leqslant d_\lambda \tag{7.8}$$

$$\Gamma^{(-)} = \frac{\gamma^{(-)}}{1 + \gamma^{(-)}} \tag{7.9}$$

[2]We remark that other causes of delay may exist which are independent of migration. In this sense, our MDP is partially observable.

$$\gamma^{(+)} = \frac{1}{M(n-1)} \sum_{i=1}^{M} \sum_{j=1}^{n-1} \left\lfloor \frac{d_{ij} - d_\lambda}{d_\Delta} \right\rfloor \text{ for } d_{ij} \geqslant d_\lambda \tag{7.10}$$

$$\Gamma^{(+)} = \frac{\gamma^{(+)}}{1 + \gamma^{(+)}} \,. \tag{7.11}$$

Here, d_{ij} is the inter-vehicular spacing between vehicles i and j, and d_Δ is the spacing tolerance. The quantities $\gamma^{(-)}$ and $\gamma^{(+)}$ measure average deviations from the target, relative to tolerance, and group negative and positive deviations, respectively. Finally $\Gamma^{(-)}$ and $\Gamma^{(+)}$ are normalized versions of average relative deviations, which guarantee values in the range between 0 and 1, 0 being the ideal target. In particular, $\Gamma^{(-)}$ tells how well (or how bad) the controller is maintaining a safe distance, whereas $\Gamma^{(+)}$ signals whether the controller is accruing or losing the benefits of platoon compactness.

Besides, in order to make informed migration decisions, we need to obtain the network specific aspect of the context, so we estimate the delay in V2I communications. Specifically, there are two delay components due to data delivery and migration overhead. The estimated data delivery time, T_{ddt} is calculated as:

$$T_{\text{ddt}} = T_{\text{net}} + T_{\text{proc}} \,, \tag{7.12}$$

where T_{net} is the estimated time for data to traverse the network and T_{proc} is the estimated time taken by the platoon controller to calculate the driving directives for the platoon members. T_{proc} depends on the capabilities of the MEC host and can be more reliably measured than T_{net}. In fact multiple factors influence T_{net}, including the capacity of the wireless channel, the number of hops through routers in the wired network, and congestion on the switched links therein.

The overhead, T_{ovh}, includes the time it takes to migrate the VNF from one host to another, $T_{\text{migration}}$, and the time $T_{\text{signaling}}$ it takes to signal the platoon members about the new location to communicate with:

$$T_{\text{ovh}} = T_{\text{signaling}} + T_{\text{migration}}. \tag{7.13}$$

In case no migration takes place, $T_{\text{ovh}} = 0$.

Given a maximum delay budget, T_{budget}, the resulting reference time ratio, T_{R} is calculated as:

$$T_{\text{R}} = \frac{T_{\text{ddt}} + T_{\text{ovh}}}{T_{\text{budget}}} \,. \tag{7.14}$$

Any candidate MEC positions yielding estimates such that $T_{\text{R}} \geqslant 1$ are not considered for migration.

Further, given that the response times of a given MEC host will vary depending on

how busy it is over a given period, the change in processing time in successive epochs, T_Δ, is a crucial decision variable.

$$T_\Delta = 2\frac{T_{\text{proc}}^{(t)} - T_{\text{proc}}^{(t-1)}}{T_{\text{proc}}^{(t)} + T_{\text{proc}}^{(t-1)}}, \tag{7.15}$$

where (t) represents the current epoch defined as the interval $[t-1, t)$ and $(t-1)$ represents the previous epoch defined as the interval $[t-2, t-1)$. We set the duration of each epoch as 20 s. This design value is a trade-off between timely migration and adequate time to collect decision data.

Each candidate MEC with $T_R < 1$ presents a migration option characterised by a given relative migration delay

$$\theta = \frac{T_{\text{migration}}^{\text{candidate}}}{T_{\text{budget}}}, \tag{7.16}$$

and relative processing capability

$$\beta = \frac{T_{\text{proc}}^{\text{candidate}}}{T_{\text{proc}}^{\text{current}}}. \tag{7.17}$$

The tuple of θ and β serves to contextualize the migration option along with the change in processing time T_Δ. We quantize θ, β and T_Δ into discrete steps.

The state of the environment is therefore fully specified by the values of $\{\Gamma^{(-)}, \Gamma^{(+)}\}$ computed from the last measurement set, and the values of θ, β and T_Δ for all MEC hosts in the environment. Possible actions for the migration agent are restricted to keep the controller in the current MEC host or migrate it to any MEC host with $T_R < 1$.

7.2.2. Problem Formulation

We consider that there exist N possible locations on which the controller can be hosted over the epoch given by $(t+1)$. The selection options over epoch $(t+1)$ can be expressed in vector form as $\eta^{(t+1)} = [\eta_1^{(t+1)}, \eta_2^{(t+1)}, ..., \eta_N^{(t+1)}]$.

The compute node, k, chosen to host the controller has unknown background traffic (ρ_k) and its location impacts the network latency (τ_k) that is experienced. Consequently, these two elements affect the spacing errors of the platoon. The placement problem can be thus be formalised as:

$$\min \sum_{i=1}^{M} \sum_{j=1}^{n-1} \left| d_\lambda - \sum_{k=1}^{N} d_{ij}(\rho_k, \tau_k)\eta_k^{(t+1)} \right| \tag{7.18a}$$

$$\text{s.t.:} \sum_{k=1}^{N} \eta_k^{(t+1)} = 1\,, \tag{7.18b}$$

$$\eta_k^{(t+1)} \in \{0, 1\}\,. \tag{7.18c}$$

Given the stochastic nature of both the background traffic and network latency from each platoon member to each of the N locations over the entire epoch, a reliable model to solve this problem is unfeasible. We therefore use model free reinforcement learning.

7.2.3. Q-learning

We briefly discuss key aspects of Q-learning which are crucial to understanding the remainder of the text. In Chapter 4, we illustrated how we adapt this technique to elicit provisioning policies driven by the time series context of cloud application workloads. For clarity, we restate some of these aspects here and proceed to describe how we adapt our technique to evoke controller migration policies driven by the cyber-physical context of platooning on the network edge.

When the environment is in state S, the agent takes action a, obtains the reward R and the environment transitions to state S' [8]. Therefore, we have

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \, \text{max}_a Q(S', a) - Q(S, A) \right), \tag{7.19}$$

where $\gamma \in [0, 1]$ is a discounting factor that weighs the contribution future rewards will have when starting in state S', and $\alpha \in (0, 1]$ is the learning rate, and makes it possible to tune the pace at which the policy converges towards the expected action-value $Q(S, A)$. If all states are visited with equal probability, α can be chosen as a fixed parameter. However, given that the agent will realistically observe only a subset of the states, we keep track of the number of times k that the agent was in state S and took action A, and define $\alpha = \frac{1}{k}, k > 0$. As will become clear later, this choice also facilitates the weighing of the Q-values when multiple agents share their experiences in the asynchronous shared learning we use to expedite policy convergence. Actions are decided after a time window termed an epoch. The set of epochs from the beginning to when the environment encounters a terminal state, where no further actions can be taken, constitute an episode.

We also employ ϵ-greedy action selection, whereby an action is chosen randomly with probability ϵ. We initially set $\epsilon = 1$ for each state, and reduce this value progressively (down to $\epsilon_{min} = 0.01$) as more visits are made to the state. This encourages exploration in the initial phases to discover the most rewarding actions and exploitation in the latter stages to make use of the policies learnt. At convergence, action selection is largely on policy and the algorithm chooses the action procuring the highest reward.

Leveraging some aspects of our work presented in Chapter 4, we design a migration agent that exploits context-awareness. A procedural depiction of this process is presented in Algorithm 2. The agent retrieves the platoon spacing data, **line 10**. The agent also obtains the processing statistics of the MEC hosts in order to estimate their capacity with reference to the current host, **line 11**. These are retrieved to derive the context, **line 12**, as specified in Equation (7.9) and Equation (7.11). It then retrieves a listing

Algorithm 2: Contextual Q-learning migration

Result: Migrate Controller, update Q-Tables for n^{th} agent

1 $Q^{(n)} \leftarrow$ ReadSharedPolicyFile();
2 $\gamma \leftarrow 0.8$;
3 $i \leftarrow 0$;
4 $RunCounter \leftarrow 0$;
5 $T_{\text{mec}} \leftarrow$ DefaultMEC;
6 $InitCtx \leftarrow$ getInitialContext();
7 **while** *True* **do**
8 $\quad Q(S, A) \leftarrow$ ReadQ($Q^{(n)}, InitCtx$);
9 $\quad$ wait_migration_interval;
10 $\quad x_{\text{log}} \leftarrow$ getPlatoonStats();
11 $\quad C_{\text{log}} \leftarrow$ getGetMecOpStats(T_{mec})
12 $\quad Ctx \leftarrow$ getContext($x_{\text{log}}, C_{\text{log}}$);
13 $\quad$ **if** $RunCounter > 0$ **then**
14 $\quad\quad ArrMEC \leftarrow$ getAvailableMEC();
15 $\quad\quad T_{\text{mec}} \leftarrow$ MoveCtrl($Ctx, Q^{(n)}, ArrMEC$);
16 $\quad$ **end**
17 $\quad$ //Update Reward
18 $\quad R \leftarrow$ getImmediateReward(x_{log});
19 $\quad Q(S', a) \leftarrow$ ReadQ($Q^{(n)}, Ctx$);
20 $\quad R' \leftarrow R + \gamma Q(S', a)$;
21 $\quad k \leftarrow$ ReadStateVisits($Q^{(n)}, InitCtx$);
22 $\quad k \leftarrow k + 1$;
23 $\quad R_{\Sigma} \leftarrow \frac{1}{k}(R' - Q(S, A)) + \frac{k-1}{k}(Q(S, A))$;
24 $\quad Q^{(n)} \leftarrow$ UpdateQ($Q^{(n)}, InitCtx, R'$);
25 $\quad$ //Store visited state for experience sharing
26 $\quad ArrShared[i] = \{R', InitCtx\}$;
27 $\quad i \leftarrow i + 1$;
28 $\quad$ **if** $i == 5$ **then**
29 $\quad\quad$ SharedXP_Thread($ArrShared$);
30 $\quad\quad i \leftarrow 0$;
31 $\quad$ **end**
32 $\quad InitCtx \leftarrow Ctx$;
33 $\quad RunCounter \leftarrow RunCounter + 1$;
34 **end**

of the available MEC hosts and estimates the migration time to the new host, **line 14**, either based on any previous logs of a similar migration, or based on a default value if no recent migration targeted that host. With these data it calculates the quantized levels of θ, β and T_{Δ} for each MEC host. The latter parameters identify the Q-values of the migration options in the decision structure shown in Figure 7.4. Comparing these options, according to Equation (7.19), the agent makes an ϵ-greedy decision as to which MEC host

.

Figure 7.4: Migrator State and Action Spaces. The relative migration delay and relative candidate MEC power tuple $\{\theta, \beta\}$, the change in processing time between successive epochs, T_Δ and the platoon topology given by the tuple $\{\Gamma^{(-)}, \Gamma^{(+)}\}$ constitute the state space. The actions "Remain in the current MEC host" and "Migrate to any MEC host characterized by $T_R < 1$" form the action space. The table $(\cdot)$ indicates migration options to MEC hosts with the same power as the current host; $(\uparrow)$ to more powerful MEC hosts; and $(\downarrow)$ to less powerful hosts than the current one.

to migrate to , **line 15**, as shown in Figure 7.5. Once the migration has been carried out, the agent updates the reward, **lines 18-24**. We also note that this migration experience is stored to be shared with other parallel agents, **lines 26-31**. The details of this shared learning will be presented in Section 7.2.5. We now discuss what comprises the reward.

7.2.4. Reward functions

A key element in an MDP is the reward function which serves as a feedback to the agent to evaluate how suitable or unsuitable a decision was. The reward function we design and implement, given the cyber-physical nature of platooning, consists of two components: the network-related reward and the vehicular positioning reward. These rewards are calculated at every epoch. Furthermore the network component takes into account the delivery intervals of control packets and a small penalty, ζ, for controller migration. The latter is to dissuade unnecessary migrations, which may perturb the platoon owing to the resulting delays.

We now consider the network component of the reward as regards packet delivery intervals. Our controller deems a packet urgent and therefore require expedited delivery if *all* of the following conditions are fulfilled: (*i*) its current estimate of the spacing between the recipient vehicle and its immediate predecessor is smaller than the inter-vehicle gap d_λ; (*ii*) the current speed of the recipient is higher than the speed of its immediate predecessor; (*iii*) the previous estimate of the spacing between the recipient vehicle and its immediate predecessor was smaller than the platoon gap; and (*iv*) the previous speed of the recipient vehicle was higher than that of its immediate predecessor. If the latter conditions are not fulfilled then the packet is deemed normal. If X_{urgent} and X_{normal} are the number of packets deemed urgent and normal over an epoch, respectively, the resulting network reward component is

$$R_{\text{net}} = \frac{X_{\text{normal}}}{X_{\text{urgent}} + X_{\text{normal}}} - \zeta \, . \tag{7.20}$$

To evaluate the second component, which accounts for platoon spacing rewards, we define the following sets and quantities in analogy to what we did for the topology context

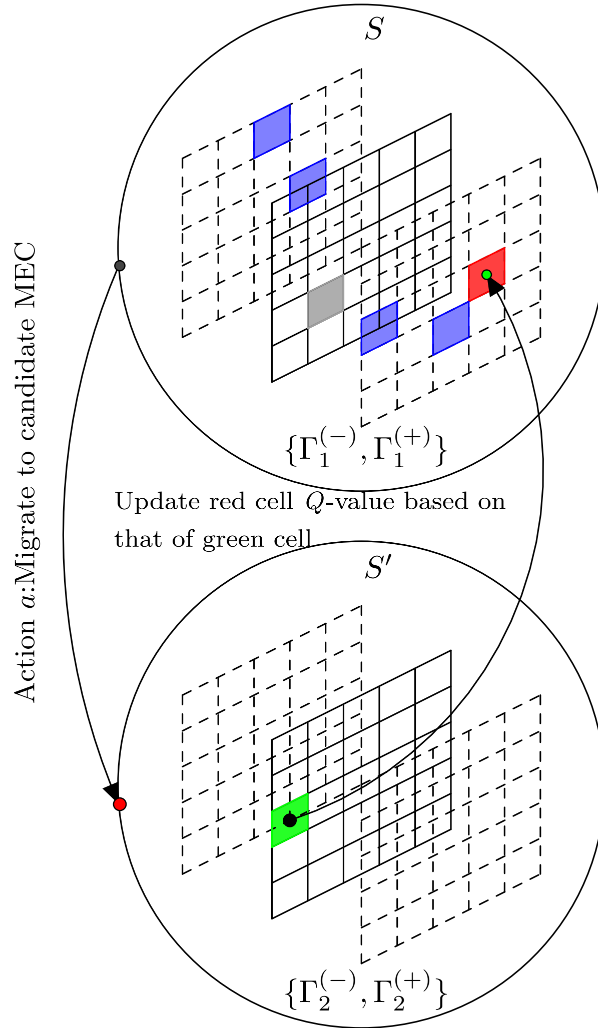


Figure 7.5: The Q-learning update process of the migration agent. The gray cell represents the Q-value of the MEC currently hosting the controller. The red cell represents $Q(S, A)$, the Q-value of the MEC host chosen to host the controller from the context of the current host and platoon topology specified by $\{\Gamma_1^{(-)}, \Gamma_1^{(+)}\}$. The green cell represents $Q(S', a)$: the Q-value of the chosen MEC in the eventual context specified by the platoon topology $\{\Gamma_1^{(-)}, \Gamma_1^{(+)}\}$. The blue cells represent Q-values of the other alternative MEC hosts not chosen for the migration.

with $\{\gamma^{(-)}, \gamma^{(+)}\}$ and $\{\Gamma^{(-)}, \Gamma^{(+)}\}$:

$$\mathcal{D} \coloneqq \{d_{ij}\}_{i=1,j=1}^{i=M,j=n-1}$$
$$\mathcal{D}_\epsilon \coloneqq \{d \in \mathcal{D} \ : \ d < d_\lambda - d_\Delta\}$$
$$\mathcal{D}_\Omega \coloneqq \{d \in \mathcal{D} \ : \ d > d_\lambda + d_\Delta\}$$
$$\mathcal{D}_0 \coloneqq \{d \in \mathcal{D} \ : \ |d - d_\lambda| \leqslant d_\Delta\}$$

$$r_\epsilon = \frac{1}{|\mathcal{D}_\epsilon|} \sum_{d_{ij} \in \mathcal{D}_\epsilon} \left| \frac{d_\lambda - d_{ij}}{d_\Delta} \right| \tag{7.21}$$

$$R_\epsilon = \frac{r_\epsilon}{1 + r_\epsilon} \tag{7.22}$$

$$r_\Omega = \frac{1}{|\mathcal{D}_\Omega|} \sum_{d_{ij} \in \mathcal{D}_\Omega} \left| \frac{d_\lambda - d_{ij}}{d_\Delta} \right| \tag{7.23}$$

$$R_\Omega = \frac{r_\Omega}{1 + r_\Omega} \tag{7.24}$$

$$r_0 = \frac{1}{|\mathcal{D}_0|} \sum_{d_{ij} \in \mathcal{D}_0} \left| \frac{d_\lambda - d_{ij}}{d_\Delta} \right| \tag{7.25}$$

$$R_0 = 0.5 - \frac{r_0}{1 + r_0} \tag{7.26}$$

Here, we have tri-partitioned inter-vehicular distances into the sets of distances within a given target interval $d_\lambda \pm d_\Delta$, or below that interval, or above. In addition, we define a safety indicator that tells whether the spacing between two cars is dangerously below the target, i.e., below $d_{\text{safe}} \ll d_\lambda$:

$$R_{\text{safe}} = \begin{cases} 0 & \text{if } d_{ij} \geqslant d_{\text{safe}} \; \forall d_{ij} \in \mathcal{D} \\ 1 & \text{otherwise} \end{cases} \tag{7.27}$$

With the above definitions, we can then compute the vehicular positioning component of the reward as:

$$R_\lambda = |\mathcal{D}_\epsilon| w_\epsilon R_\epsilon + |\mathcal{D}_\Omega| w_\Omega R_\Omega \tag{7.28}$$

$$+ |\mathcal{D}_0| w_0 R_0 - |\mathcal{D}| R_{\text{safe}}, \tag{7.29}$$

where $w_\epsilon, w_\Omega, w_0$ are weights assigned so as to prioritise either safety or spacing of the platoon and d_{safe} is the minimum spacing below which a crash is likely to occur. We finally compute the total reward R, accrued in one epoch as:

$$R = |\mathcal{D}| R_{\text{net}} + R_\lambda. \tag{7.30}$$

A training episode is characterised by several consecutive epochs and terminates with the platoon successfully completing a given road segment (e.g., a lap in a circuit).

7.2.5. Asynchronous shared learning

We now propose an extension to the above framework that enables shared learning across different platoons. This extension makes it possible to leverage the state exploration in different platoons and thus explore the state space more extensively in less time. We assume that multiple platoons exist and visit the same road segments. Each platoon

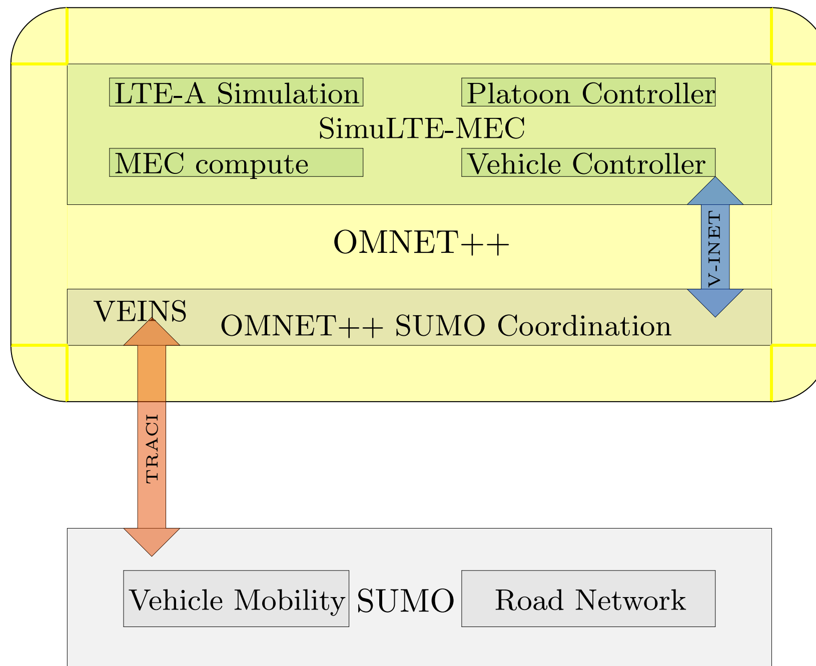


Figure 7.6: Simulation frameworks

controller runs on an independent slice in the MEC host. As such, each platoon could be transparent to the others. However, we further assume that the migration agent of each platoon can read and write a common migration policy file, from where it fetches its own policy. The agent can also update the policy by annotating what it learns. Each migration agent can only access this shared file every so often, so that we can neglect the overhead associated with sharing the experience learned by other platoons.

In this scheme, agents do not need to be synchronized, and rather could access the migration policy at any time. Between two consecutive accesses to the shared policy file, each agent keeps its own version of the file, with updates to Q-values of the visited states that will only later be reflected in the shared policy.

The scheme described above can model the behaviour of a distributed learning process, in which a central entity asynchronously collates updates from all agents, e.g., when they connect to a given eNodeB, or when they migrate to a specific MEC host. The scheme can also be extended to become virtually overhead-free and distributed, if we assume that each MEC host maintains a migration policy file, and that platoon controllers migrating to a MEC host bring in all their past learned policy values. This way, the policy built at a MEC host can be spread to the other MEC hosts by simply being transferred jointly with controllers during migrations. We argue that this asynchronous scheme is potentially effective to speed up the convergence of learned migration policies.

Table 7.1: Simulation Parameters

Parameter	**Value**
Path loss model	ITU Rural macro cell
eNodeB antenna gain	18 dB
eNodeB height	25 m
eNodeB transmit range	500 m
eNodeB transmit power	43 dBm
Size of uplink and downlink packet	39 Bytes
Block error rate	1%
Handover latency (per vehicle)	$X \sim \mathcal{U}(0\text{ ms}, 10\text{ ms})$
Core routing delay (per packet)	$X \sim \text{Exp}(1\text{ ms})$
X2 hop delay (per packet)	50 μs
Migration delay (per vehicle)	$X \sim \mathcal{U}(1\text{ ms}, 3\text{ ms})$
Migration epoch interval	20 s
Vehicular reporting interval	100 ms
Platoon speed V_{ptn} (Average)	25 m/s
Platoon speed (Oscillation)	20% V_{ptn}
Platoon speed (Frequency)	0.1 Hz
Number of cars (n)	30
Platoon spacing (d_λ)	10 m
Migration penalty (ζ)	0.05
d_{safe}	1 m
d_Δ	0.5 m
w_ϵ	-0.5
w_Ω	-0.1
w_0	0.25

7.3. Performance evaluation

7.3.1. Method

Our simulation testbed comprises three main components as shown in Figure 7.6. The first is the robust vehicular simulator Simulation of Urban MObility (SUMO) from the German Aerospace Agency DLR [90]. It is widely used in research given its highly realistic depiction of vehicular mobility. The second component is the OMNeT++ Framework "Vehicles in Network Simulation (VeINS)" [91]. This framework provides an API that facilitates the coordination of the vehicle simulation in SUMO with the network simulation of corresponding UE in OMNeT++. The third component of the simulator is SimuLTE-MEC [92], an OMNeT++ framework that realistically simulates the LTE-Advanced network with mobile edge computing extensions.

In order to test the operation of SALT, we simulate the road and MEC network as depicted in Figure 7.7. In this scenario, only one MEC host is available and migration is not an option. Whereas eNodeB1 and eNodeB2 have a fast connection to the MEC host,

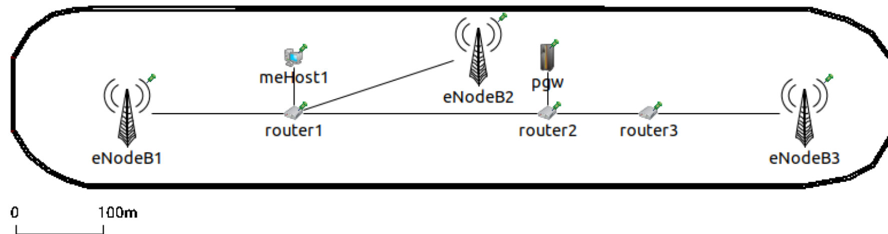


Figure 7.7: Road and MEC Network for evaluating SALT. The scale corresponds to the road network, eNodeBs and the MEC host are exaggerated to make them discernible.

eNodeB3 does not have a LAN connection and has to rely on slower routing through the core network when it handles packets between the platoon members and the controller. We vary the average exponential routing delay through the core network between 1 ms and 10 ms. With respect to Equation (7.5), we choose $T_{\text{max}} = 50\,\text{ms}$ which is half the reporting interval for the vehicles. In addition, with reference to Equation (7.6) and Equation (7.7), we set $\Psi = 0.3$, $V_{\text{acc}} = 10\,\text{m}\,\text{s}^{-1}$ and $r = 0.9$. The rest of the simulation parameters are as specified in Table 7.1.

As regards the other scenario we consider with multiple MEC hosts, we implement the migration scheme as a python script that monitors the logs generated by OMNeT++ as the platoon traverses its course. The script uses the logs as input to the Q-learning migration scheme. It also monitors the simulation time and, at regular intervals, updates a reference file with the selected MEC host. In the process of generating a vehicular report to send to the MEC host, SimuLTE reads the reference file. If there is a change, it triggers a migration from the current location to the selected MEC host. Table 7.1 lists the key parameter choices we make in SimuLTE to better capture the envisaged 5G network capabilities. In particular, we set the handover delay to random values below 10 ms. A handover occurs when a vehicle receives 3 successive signals from a target eNodeB that have a higher RSSI compared to the one that is currently serving it.

Table 7.1 also lists the parameters used to compute the reward function of platooning. We choose the weights such that the negative reward (i.e., the penalty) of being 20% below the target spacing or 50% above has comparable impact to a migration. The rationale is that we have observed up to about 20% of variation in speed profiles with migrations, which is comparable with driving 20% below the target for what concerns safety, and with the platoon occupying about 20% more road space when driving at 15 m instead of 10 m, although the exact value depends on the platoon size.

We curtail fading aspects of the channel given that network densification with small cells will lead to higher probability of line-of-sight communications. We enhance the computing infrastructure of the MEC hosts with FIFO queues, in order to simulate the variability of processing delays due to competing third-party background processes. In addition we introduce queuing delays on the switching elements to account for routing

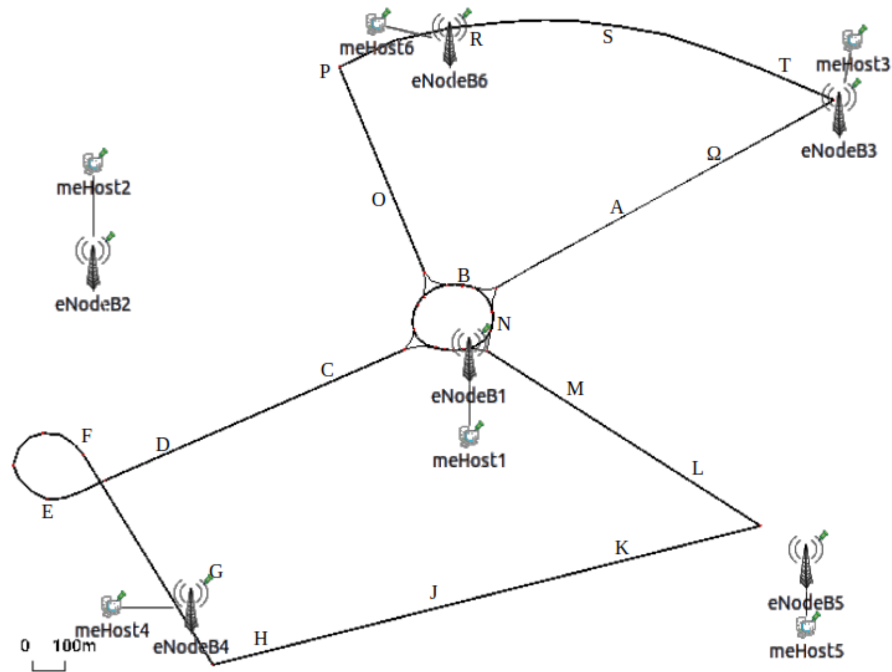


Figure 7.8: Road and MEC Network. The scale corresponds to the road network, eNodeBs and MEC hosts are exaggerated to make them discernible.

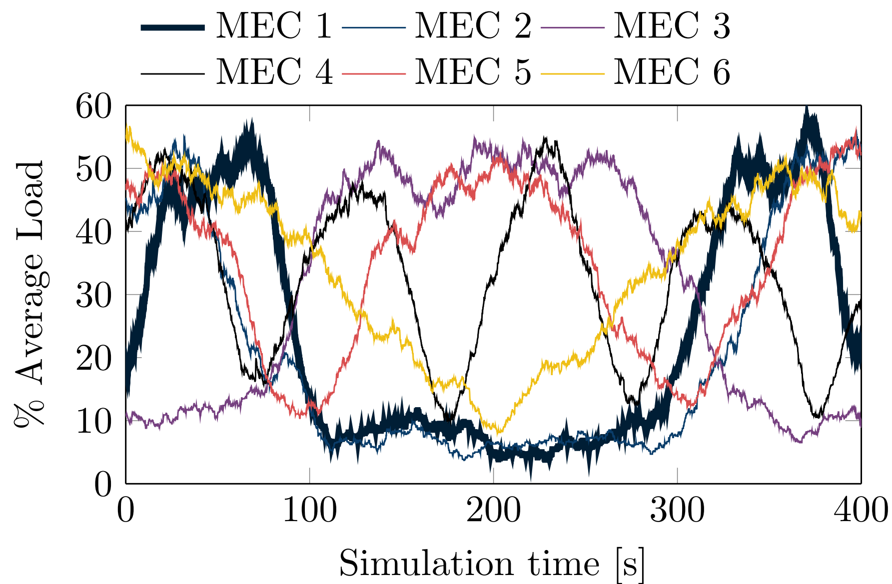


Figure 7.9: MEC host load due to background traffic.

delays that occur when the MEC node hosting the controller is not directly connected to the eNodeB serving a vehicle. Figure 7.8 depicts the road circuit that we use in SUMO to delimit a training episode, along with the communication network in SimuLTE-MEC. We remark that our scheme is independent of the placement of the MEC hosts within the network (whether closer to the eNodeB or to the core), given that it distinguishes them

Algorithm 3: Asynchronous Shared Learning

Result: Update Shared Policy File
1 **Function** SharedXP_Thread*(xArr)*
2 //Ensure this thread has sole access to shared file LockSharedPolicyFile();
3 //Capture experiences of other parallel agents $Q^{(n)} \leftarrow$ ReadSharedPolicyFile();
4 //Update the working policy with own experiences
5 **foreach** $XP \in xArr$ **do**
6 $R_n \leftarrow XP[0]$;
7 $TmpCtx \leftarrow XP[1]$;
8 $k_n \leftarrow$ ReadStateVisits($Q^{(n)}, TmpCtx$);
9 $k_n \leftarrow k_x + 1$;
10 $Q(S, A) \leftarrow$ ReadQ($Q^{(n)}, TmpCtx$);
11 $R_x \leftarrow \frac{1}{k_x}(R_n - Q(S, A)) + \frac{k_x - 1}{k_x}(Q(S, A))$;
12 $Q^{(n)} \leftarrow$ UpdateQ($Q^{(n)}, TmpCtx, R_x$);
13 **end**
14 //Publish updated policy for all agents WriteSharedPolicyFile($Q^{(n)}$);
15 //Release file for use by other agents UnLockSharedPolicyFile();
16 *return*;
17 **end**

based on processing capacity and migration delay.

The simulation time at location A is 31 s, the subsequent intervals after that are 20 s each. These are the epochs at which the migration agent makes its decisions. In order to make the migration scenario challenging, each eNodeB has a MEC node attached to it such that the controller may be hosted at those locations. The background traffic of the MEC nodes, independent of the platooning load, is depicted in Figure 7.9. We chose these patterns to represent realistic workload traces with different periodicity. We remark that none of the patterns is in sync with the time the platoon needs in order to complete a loop along the circuit of Figure 7.8.

7.3.2. Asynchronous shared learning extension

To test the potential of the asynchronous shared learning scheme described in Section 7.2.5, we use parallel simulations of platoons of cars lapping through the same circuit. We test the centralized version of our scheme, with each platoon accessing the shared migration policy every 5 epochs.

At initialization, the parent platooning simulation directory is cloned into a number of directories commensurate to the number of parallel agents chosen. However, the overall policy file is placed in a directory accessible to all participating agents. After cloning, a

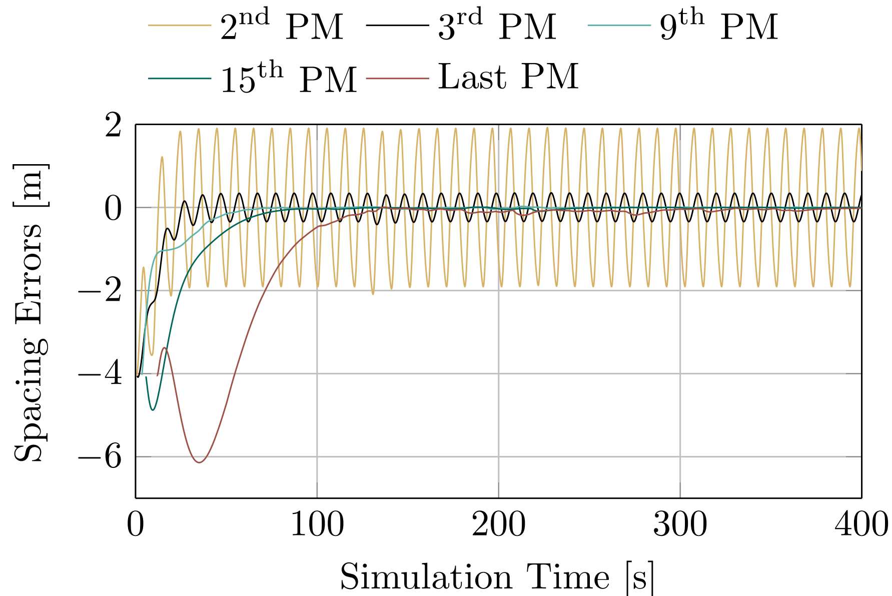


Figure 7.10: Spacing errors of platoon members (negative means farther).

shell script triggers the start of the SUMO/OMNeT++ simulation as well as that of the migration script in each directory.

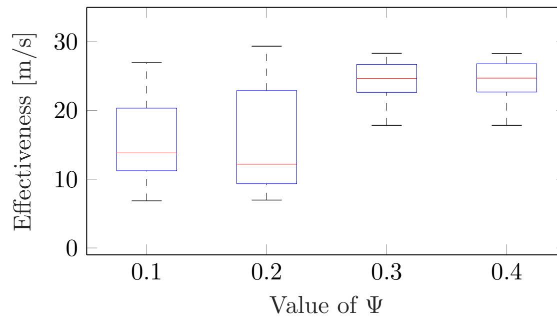
Our shared learning procedure is as shown in Algorithm 3. When the migration agent starts, it reads the overall policy file into a data structure and proceeds to update it as it makes its decisions. The start time of the simulation is different in each directory, in order to mirror the separation of the platoons in the real world. After about 100 s (which corresponds to about five migration decisions), the migration agent locks the policy file and reads the overall policy into its data structure, **lines 2-3**, so as to capture any updates by the other participating agents. The agent then proceeds to update the data structure with Q-values and immediate rewards of the states it visited, **lines 4-12**, as prescribed in Eq. Equation (7.19). It then overwrites the policy file with the updated data structure, **line 13**. Finally, it unlocks the policy file, **line 14**. Should an agent find the file locked by another, it waits for a random time and then retries until the file is unlocked.

7.3.3. Evaluation results

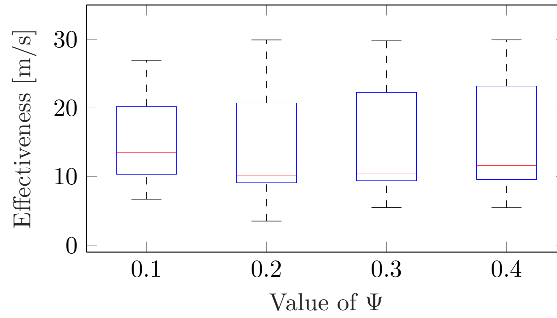
In this section we examine the results from the modifications on the controller. We then consider platoon performance resulting from the use of our migration scheme, compared to that of the state of the art scheme proposed in [74].

7.3.3.1. Controller modifications

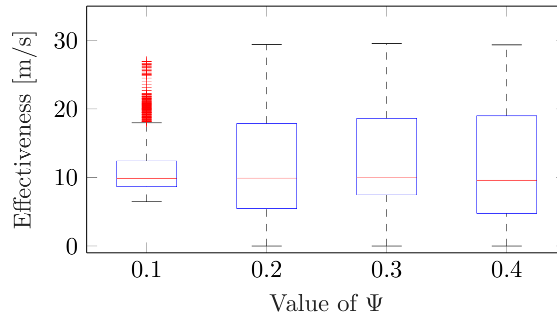
We first examine the platoon formation when the controller is hosted on the MEC. In order to set the most challenging condition for the controller [81], we add a sinusoidal



(a) Core routing delay $X \sim \text{Exp}(1\,\text{ms})$



(b) Core routing delay $X \sim \text{Exp}(5\,\text{ms})$



(c) Core routing delay $X \sim \text{Exp}(10\,\text{ms})$

Figure 7.11: Platoon effectiveness (spacing fairness × average speed) for different SALT triggers.

perturbation to the movement speed of the platoon leader. This sinusoidal driving pattern also accounts for speed variations as the platoon may need to slow down or speed up depending on driving conditions. The platoon speed oscillates between $21.5\,\text{m}\,\text{s}^{-1}$ ($77\,\text{km}\,\text{h}^{-1}$) and $\approx 29\,\text{m}\,\text{s}^{-1}$ ($104\,\text{km}\,\text{h}^{-1}$). The 30 vehicles of the platoon fully stabilize in about 100 s, as shown in Figure 7.10. The spacing between the 2nd PM (i.e., the first follower) and the platoon leader exhibits the highest variation, as expected. The effect on the rest of the followers is progressively damped down towards the tail of the platoon. From this result, we conclude that our modifications on the MEC-hosted controller preserve string stability [93] and thereby ensure platoon safety.

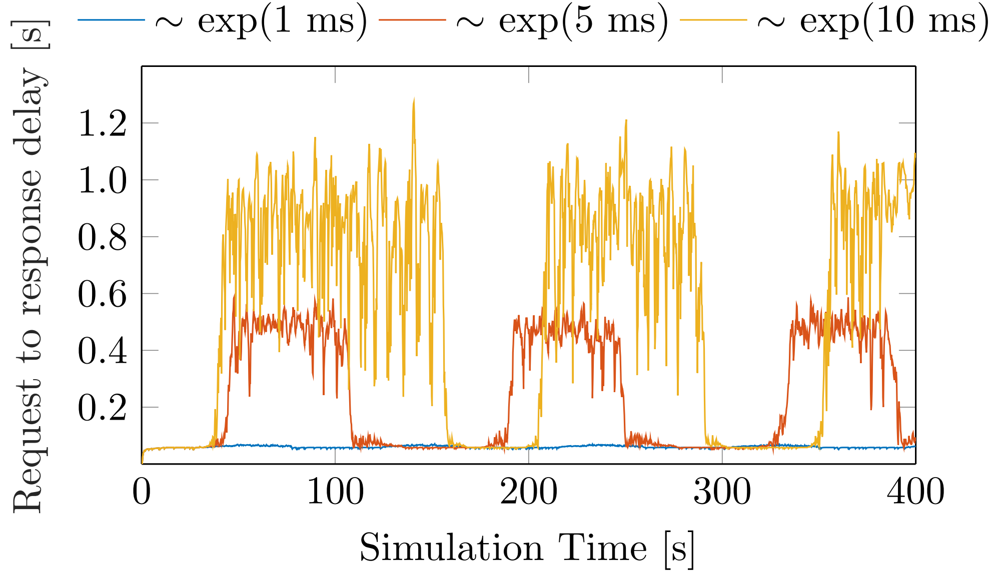


Figure 7.12: Filtered overall delay (interval between a vehicle sending packet to receiving a control directive from the controller) for the SALT scenario.

7.3.3.2. SALT

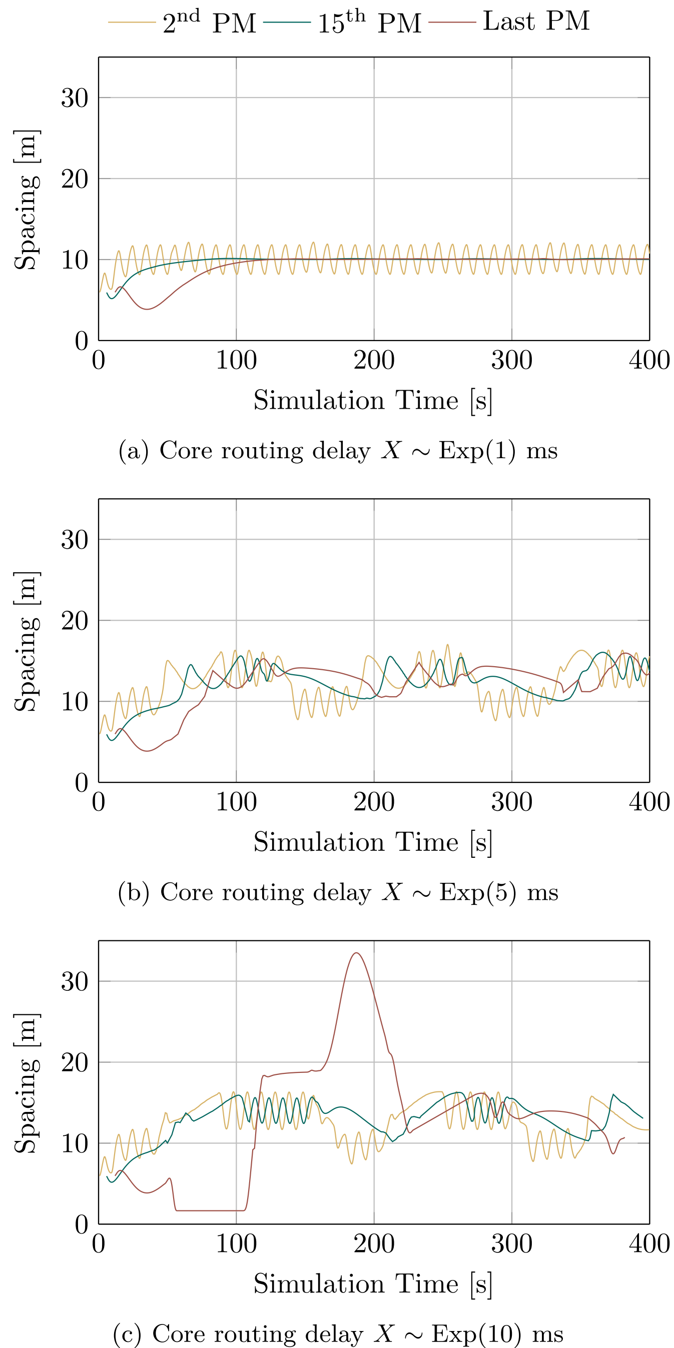
In order to compare platoon performance with regard to the safety trigger value Ψ, we derive an Effectiveness metric by multiplying Jain's fairness index [94] in spacing, x, and the average speed, $\dot{x}$ for all n platoon members over each observation window.

$$\text{Effectiveness} = \frac{\left(\sum_{j=1}^{n-1} x_j\right)^2}{(n-1)\sum_{j=1}^{n-1} x_j^2} \times \frac{1}{n}\sum_{j=1}^{n} \dot{x}_j \tag{7.31}$$

The distribution of this metric is depicted in Figure 7.11. A low value, $\Psi = 0.1$ or $\Psi = 0.2$, makes the controller too sensitive such that the platoon devolves into Automated Cruise Control even when the core routing delay is small as in Figure 7.11a. On the other hand a higher value, $\Psi = 0.4$, does not trigger the safety measure quickly enough when the delay values are substantially higher possibly resulting in low effectiveness overall as shown in Figure 7.11c. A setting of $\Psi = 0.3$, however, provides a balance between these extremes.

The total delay, filtered using Equation (7.4), between when a vehicle sends a packet to the controller to when it receives a packet from the controller is shown in Figure 7.12. When the core routing delay is sufficiently low, with an average exponential distribution of 1 ms, all responses from the controller arrive within the 100 ms interval between consecutive vehicular reports. As the core routing delay increases, the periods in which platooning can be safely carried out reduces.

We also evaluate the performance of SALT with increasing delay as shown in Figure 7.13. When the delay is low, cf. Figure 7.13a, ACC is not triggered and CACC platooning is maintained throughout. With a moderate increase in delay, as shown in



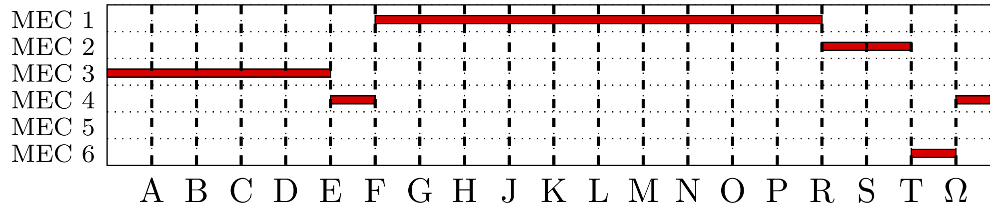
(a) Core routing delay $X \sim \text{Exp}(1)$ ms

(b) Core routing delay $X \sim \text{Exp}(5)$ ms

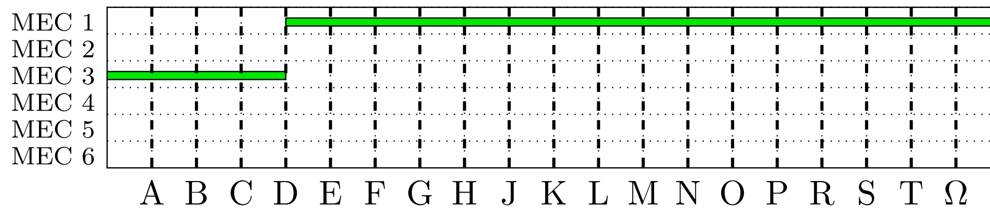
(c) Core routing delay $X \sim \text{Exp}(10)$ ms

Figure 7.13: SALT performance for varying degrees of delay with $\Psi = 0.3$

Figure 7.13b, there is a smooth transition between ACC and CACC. When the delay is high (e.g., in the experiment time interval between 50 s and 150 s, see Figure 7.12), the vehicles are controlled by ACC with larger spacing between them. When the delay becomes tolerable (e.g., from 150 s to 190 s of simulated time, cf. Figure 7.12), the platoon is reformed and the vehicle spacing reverts to the design value. These transitions



(a) Q-migration Policy A



(b) Q-migration Policy B

Figure 7.14: Sample Q-migration policies

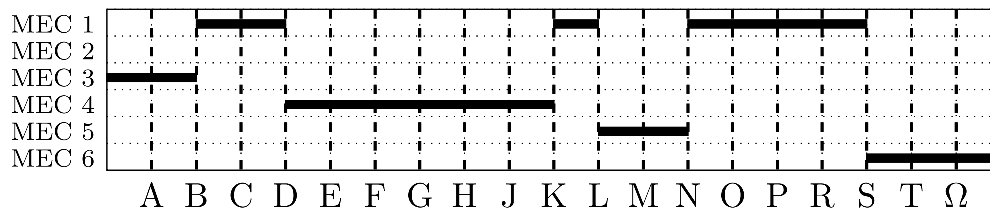


Figure 7.15: Follow ME migration policy [74]

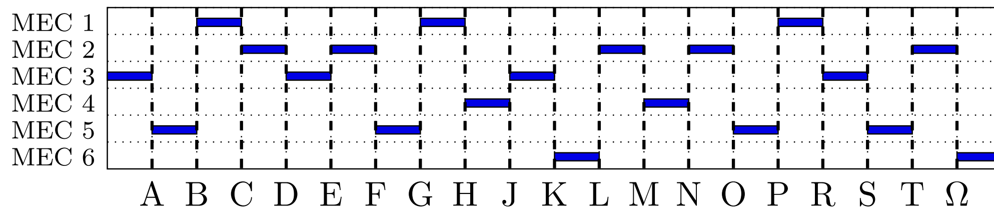


Figure 7.16: AUSP migration policy [85]

in spacing are more gradual. However, when the routing delay is substantially higher, Figure 7.13c, the switch between ACC and CACC is more drastic especially towards the tail of the platoon. There are also brief periods in which the vehicles at the tail end of the platoon almost come to a stop as ACC control re-establishes safe spacing. Overall, increasing delays lead to increasing ACC dominance over CACC and vice-versa.

In the following sections, we update the results of our previous experiments in [3] incorporating SALT into the workings of the controller.

7.3.3.3. Migration strategies

The proximity to an eNodeB and the load on a given MEC host may be contrasting objectives for the migration agent to pursue. For example, the agent may run on the MEC host connected to the eNodeB closest to the platoon (hence it perceives a low air interface communication latency). However, if the load on this MEC host increases, the agent may need to migrate to a different MEC host, and the resulting routing and switching delays may nullify the advantage of being connected to a near eNodeB. Another important consideration is that the migrations should be kept at a minimum given the extra delays involved in switching from one MEC host to another.

A subset of the migration strategies obtained by our algorithm are depicted in Figure 7.14. These sequences were the most recurrent in which the migration agent attained full convergence for every visited state (i.e., $\epsilon = \epsilon_{\min}$). We omitted less frequently observed sequences for brevity. By design, we initially position the controller at MEC 3 (towards the top-right section of the track). This avoids any biases that might give undue advantage to one migration policy over the other.

Our algorithm learns policies that exploit both the proximity and the computing capability of the MEC host. This reflects in the agent migration patterns, which initially involve different MEC servers, but then prefer stabilizing the controller on the same server despite the server load and the additional routing delays. Notably, different policies attain the same conclusions, and elect a MEC server on which they remain until the end of our simulations.

In contrast, the state-of-the-art Follow ME [74], that only considers proximity when selecting the preferred MEC host, forces numerous migrations as shown in Figure 7.15. Moreover, the other state-of-the-art scheme we consider in this chapter (AUSP [85]), which also takes into account the processing delay, migration cost and communication delay), results in even more migrations as it tries to minimize the total delay as depicted in Figure 7.16. Yet, these migrations may prove inconvenient, as we discuss next.

7.3.3.4. Platoon stability

Considering the same sinusoidal driving pattern employed so far, Figure 7.17 shows that using our Q-Migration scheme, the speed of the first follower adapts very well to that of the platoon leader. This takes place despite the sinusoidal perturbations on the platoon leader speed, and confirms that the learned migration policies show very good robustness. Policy A which involves more migrations compared to Policy B exhibits instances in which SALT is triggered but these are short-lived. In comparison, Follow ME [74] and AUSP [85] exhibit much greater variability given that SALT is triggered over longer intervals. AUSP which carries out a considerable number of migrations exhibits a higher number of speed deviations as a consequence of the extra delays incurred. This points to the efficacy of

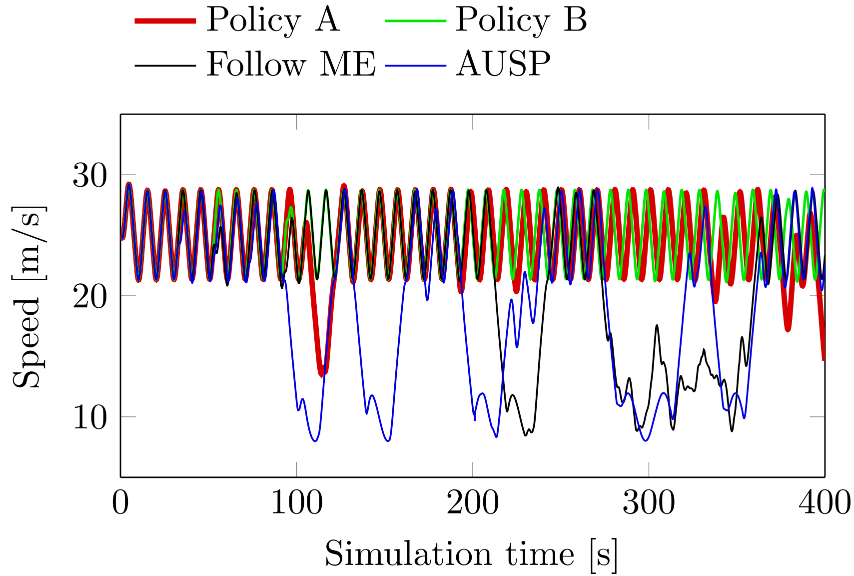


Figure 7.17: Speed profiles of the first follower.

the minimal migrations that our algorithm decides to perform.

When compared to Follow ME and AUSP, our scheme exhibits better platoon spacing discipline. This is shown in Figure 7.18, which employs box-plots to convey the distribution of vehicle spacing across the platoon at different simulation times. Each blue box extends from the 1st to the 3rd quartile of the distribution, the red bar denotes the median, and whiskers cover the 10th-90th percentile range. Red "+" markers denote the data along the tails of the distribution (primarily due to the spacing between the first follower and the platoon leader). From Figure 7.18, we observe that our Q-migration algorithm achieves smaller inter-quartile ranges (Figure 7.18a and Figure 7.18b) compared to those of Follow ME (Figure 7.18c) and AUSP (Figure 7.18d). Follow ME exhibits small dispersion when it maintains the controller on MEC 4 between simulation time 100 s to 200 s (cf. 7.15) since it does not incur migration costs. As it resumes more migrations after 200 s, dispersion becomes significant. AUSP attempts to even out the spacing resulting in fewer outlying values from the first follower. It however also results in noticeable dispersion throughout. The gains that AUSP achieves in migrating to the lowest utilized, closest MEC host are curtailed by the large migration costs incurred in the process.

The narrow inter-quartile range achieved by our scheme implies higher string stability and a generally better driving experience. In particular, the occurrences of long whiskers are due to the the spacing between the first follower and the platoon leader, which varies the most under the sinusoidal motion of the leader.

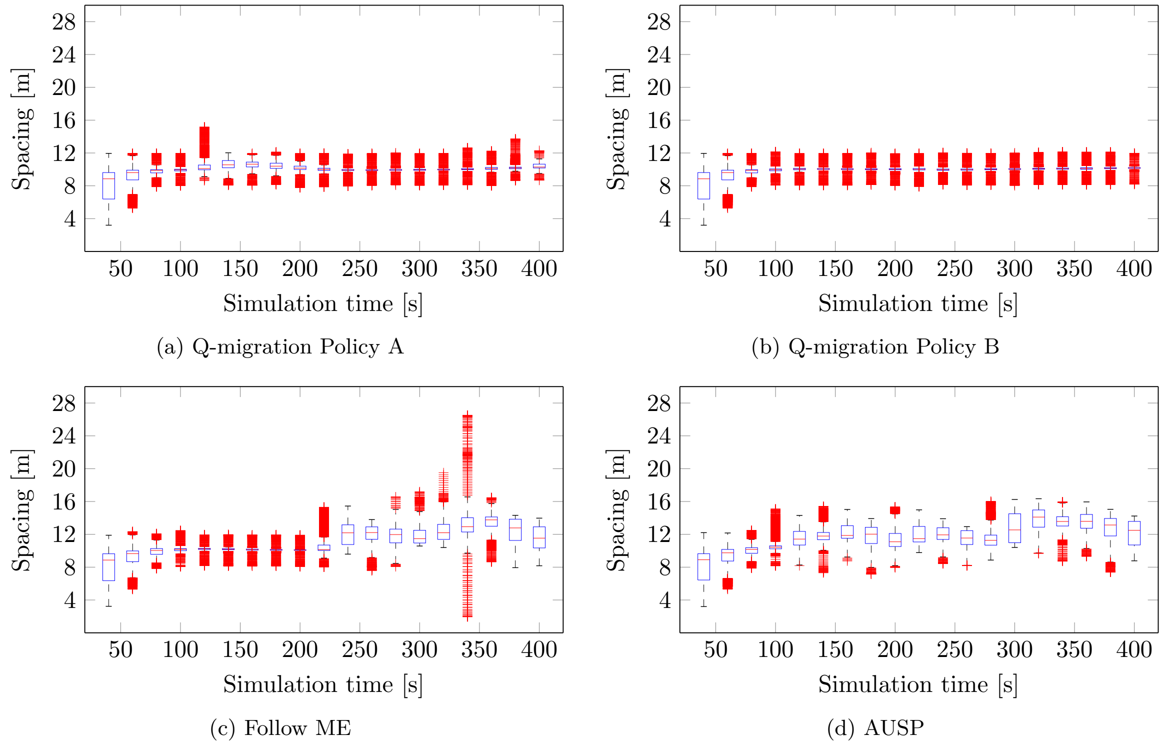


Figure 7.18: Distribution of spacing between platoon members. Each box plot represents data taken over 20 s windows. The last box plot represents data in the last 10 s.

7.3.3.5. Asynchronous shared learning

The acceleration of convergence through the use of asynchronous shared learning is apparent from Table 7.2. In this table, we consider each set of parallel episodes, and examine the log of the states that each parallel agent visits. If $\epsilon = \epsilon_{\text{min}}$ at every epoch for any one or more of the parallel agents, we consider that set of episodes as exhibiting convergence given that the policy may be accessed by all agents. The nth time that this is observed, for a set of parallel episodes, is termed the nth convergence. The results prove that having more agents sharing their own experience with other agents greatly increases the speed at which migration policies fully converge. However, after the initial significant gain, the advantage of having more parallel agents decreases. This points to a diminishing return in the value of parallel agents after the first convergence.

7.4. Summary and discussion

In this chapter, we have presented a context-aware Q-learning-based migration algorithm for vehicle platoon controllers running on the network edge. The algorithm learns the appropriate strategies to migrate the controller from one MEC host to another in a cellular network context. Our approach reduces the number of migrations required for

Table 7.2: Parallel episodes until convergence

# Agents	1st Conv.	2nd Conv.	3rd Conv.	4th Conv.	5th Conv.
10	73	75	76	77	78
5	304	311	315	321	348
1	867	1293	1306	1322	1340

the controller to keep pace with the platoon and maintain it in formation. In addition, we have also presented our asynchronous shared learning algorithm that leverages the presence of multiple platoons in order to learn migration policies faster. In particular, compared to the state-of-the-art schemes, FollowME and AUSP, our approach achieves better adherence to platoon speed and spacing presets. It therefore greatly reduces statistical dispersion of spacing between platoon members.

We have also presented the customization needed to enable CACC operation from the network edge. Furthermore, we have presented an overlay safety module that enables the graceful switching of control between the MEC controller (CACC) and the vehicle (ACC) in case of sustained high delays.

8 Conclusions

In this thesis, we have presented zero-touch provisioning systems that can dynamically adapt the number of cloud network compute resources to fit demand and migrate MEC enabled applications to the most suitable location on the network edge.

In part I, we have shown how the interaction between incident demand and compute resource response can be leveraged to model the cloud environment as a MDP. We have shown that if the cloud application is known, a robust dynamic provisioning system can be obtained by leveraging the stationary transition probabilities of the underlying Markov chains. In cases where it is either impractical or infeasible to model the cloud application or where the resource configuration may not be known *a-priori*, we have presented a contextual Q-Learning algorithm. Our algorithm can solve the underlying MDP and thereby know when and by how much to adjust the commited compute resources in the face of changing demand. We have shown that the algorithm is versatile enough to be adapted to different Service Level Agreement (SLA) constraints and can also be applied to provisioning in server-less cloud computing environments using containers.

In second part of the thesis, we have shown that by changing the time context of the algorithm to compute capacity and location dependent contexts, we can create a MEC provisioning system that can learn when and where to migrate a latency sensitive application for users with high mobility. We have also presented a joint learning scheme whereby multiple agents can cooperate to share their experiences of an environment and in so doing greately speed up the convergence of a Q-Learning system by more than an order of magnitude.

References

[1] C. Ayimba, P. Casari, and V. Mancuso, "Adaptive resource provisioning based on application state," in *IEEE ICNC 2019 - International Conference on Computing, Networking and Communications (ICNC)*, 2019, pp. 663–668.

[2] ——, "SQLR: Short-term memory Q-learning for elastic provisioning," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 2, pp. 1850–1869, 2021.

[3] G. Somma, C. Ayimba, P. Casari, S. P. Romano, and V. Mancuso, "When less is more: Core-restricted container provisioning for serverless computing," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2020, pp. 1153–1159.

[4] C. Ayimba, M. Segata, P. Casari, and V. Mancuso, "Closer than close: Mec-assisted platooning with intelligent controller migration," in *Proceedings of the 24th International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, ser. MSWiM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 23â32. [Online]. Available: https://doi.org/10.1145/3479239.3485681

[5] ——, "Driving Under Influence: MEC-Enabled Platooning," *Under revision in Elsevier Computer Communications*, 2022.

[6] J. Anselmi, D. Ardagna, J. C. S. Lui, A. Wierman, Y. Xu, and Z. Yang, "The economics of the cloud," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 2, no. 4, pp. 18:1–18:23, Aug. 2017.

[7] C. Wang, B. Urgaonkar, G. Kesidis, A. Gupta, L. Y. Chen, and R. Birke, "Effective capacity modulation as an explicit control knob for public cloud profitability," *ACM Trans. Auton. Adapt. Syst.*, vol. 13, no. 1, pp. 2:1–2:25, May 2018.

[8] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 2nd ed. Cambridge, MA, USA: MIT Press, 2020. [Online]. Available: http://incompleteideas.net/book/RLbook2020.pdf

[9] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, "Automated control in cloud computing: Challenges and opportunities," in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ser. ACDC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 13â18. [Online]. Available: https://doi.org/10.1145/1555271.1555275

[10] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck, "From data center resource allocation to control theory and back," in *Proc. IEEE ICCC*, Miami, FL, July 2010.

[11] S. Farokhi, E. B. Lakew, C. Klein, I. Brandic, and E. Elmroth, "Coordinating CPU and memory elasticity controllers to meet service response time constraints," in *Proc. ICCAC*, Boston, MA, Sep. 2015, pp. 69–80.

[12] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi, "Integrated and autonomic cloud resource scaling," in *Proc. IEEE/IFIP NOMS*, Maui, HI, Apr. 2012, pp. 1327–1334.

[13] P. Jamshidi, A. Ahmad, and C. Pahl, "Autonomic resource provisioning for cloud-based software," in *Proc. SEAMS*, Hyderabad, India, Jun. 2014, pp. 95–104.

[14] "Rightscale," https://www.rightscale.com, accessed: 2020-12-23.

[15] K. Hightower, B. Burns, and J. Beda, *Kubernetes: up and running: dive into the future of infrastructure.* O'Reilly Media, Inc., 2017.

[16] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Elsevier FGCS*, vol. 27, no. 6, June 2011.

[17] J. Bi, H. Yuan, W. Tan, M. Zhou, Y. Fan, J. Zhang, and J. Li, "Application-aware dynamic fine-grained resource provisioning in a virtualized cloud data center," *IEEE TASE*, vol. 14, no. 2, April 2017.

[18] S. K. Garg, A. N. Toosi, S. K. Gopalaiyengar, and R. Buyya, "SLA-based Virtual Machine Management for Heterogeneous Workloads in a Cloud Datacenter," *Elsevier JNCA*, vol. 45, Oct 2014.

[19] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Elsevier FGCS*, vol. 28, no. 1, Jan 2012.

[20] D. A. Menasce, "TPC-W: a benchmark for e-commerce," *IEEE Internet Computing*, vol. 6, no. 3, May 2002.

[21] A. Alsarhan, A. Itradat, A. Y. Al-Dubai, A. Y. Zomaya, and G. Min, "Adaptive resource allocation and provisioning in multi-service cloud environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 1, pp. 31–42, Jan. 2018.

[22] J. V. Bibal Benifa and D. Dejey, "RLPAS: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment," *Springer Mobile Netw. Appl.*, vol. 24, no. 4, pp. 1348–1363, Aug. 2019.

[23] J. Rao, X. Bu, C. Xu, and K. Wang, "A distributed self-learning approach for elastic provisioning of virtualized cloud resources," in *Proc. IEEE MASCOTS*, Singapore, Jul. 2011, pp. 45–54.

[24] V. Ravi and H. S. Hamead, "Reinforcement learning based service provisioning for a greener cloud," in *Proc. ICECCS*, Mangalore, India, Dec. 2014, pp. 85–90.

[25] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Model-driven optimal resource scaling in cloud," *Softw. Syst. Model.*, vol. 17, no. 2, pp. 509–526, May 2018.

[26] O. Ibidunmoye, M. H. Moghadam, E. B. Lakew, and E. Elmroth, "Adaptive service performance control using cooperative fuzzy reinforcement learning in virtualized environments," in *Proc. ACM UCC*, Austin, Texas, USA, Dec. 2017, pp. 19–28.

[27] J. Liu, Y. Zhang, Y. Zhou, D. Zhang, and H. Liu, "Aggressive resource provisioning for ensuring QoS in virtualized environments," *IEEE Trans. on Cloud Comput.*, vol. 3, no. 2, pp. 119–131, Apr. 2015.

[28] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling Web Applications in Heterogeneous Cloud Infrastructures," in *Proc. IEEE IC2E*, Boston, MA, Mar. 2014, pp. 195–204.

[29] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar, "Cost-efficient and application SLA-aware client side request scheduling in an Infrastructure-as-a-Service cloud," in *Proc. IEEE CLOUD*, Honolulu, HI, Jun. 2012, pp. 213–220.

[30] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "DejaVu: Accelerating resource allocation in virtualized environments," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 423–436, Mar. 2012.

[31] F. Xu, H. Zheng, H. Jiang, W. Shao, H. Liu, and Z. Zhou, "Cost-effective cloud server provisioning for predictable performance of big data analytics," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 1036–1051, May 2019.

[32] M. L. Littman, "Algorithms for sequential decision making," Ph.D. dissertation, Brown University, Providence, RI, Mar 1996.

[33] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, “Planning and acting in partially observable stochastic domains,” *Elsevier Artificial Intelligence*, vol. 101, no. 1, pp. 99–134, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S000437029800023X

[34] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “Kvm: the linux virtual machine monitor,” in *Proceedings of the Linux symposium*, vol. 1, no. 8. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[35] M. Bolte, M. Sievers, G. Birkenheuer, O. Niehörster, and A. Brinkmann, “Non-intrusive Virtualization Management Using Libvirt,” in *Proc. DATE*, Leuven, Belgium, Mar 2010, pp. 574–579.

[36] “Gstreamer open source multimedia framework,” https://gstreamer.freedesktop.org, accessed: 2018-07-11.

[37] “Big Buck Bunny,” http://bbb3d.renderfarming.net/download.html, accessed: 2018-07-11.

[38] N.-M. Nguyen, D.-H. Bui, N.-K. Dang, E. Beigne, S. Lesecq, P. Vivet, and X.-T. Tran, “An Overview of H.264 Hardware Encoder Architectures Including Low-Power Features,” *JEC*, vol. 4, June 2014.

[39] “FFmpeg,” https://www.ffmpeg.org, accessed: 2018-07-11.

[40] D. K. Krishnappa, M. Zink, and R. K. Sitaraman, “Optimizing the video transcoding workflow in content delivery networks,” in *Proc. ACM MMSys*, Portland, Oregon, Mar 2015.

[41] Y. O. Sharrab and N. J. Sarhan, “Detailed comparative analysis of vp8 and h.264,” in *Proc. 2012 IEEE ISM*, Irvine, CA, Dec 2012.

[42] S. T. King and et al., “Operating system support for virtual machines,” in *Proc. USENIX ATC*, San Antonio, TX, June 2003.

[43] C. Marquez, M. Gramaglia, M. Fiore, A. Banchs, C. Ziemlicki, and Z. Smoreda, “Not all apps are created equal: Analysis of spatiotemporal heterogeneity in nationwide mobile service usage,” in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 180â186. [Online]. Available: https://doi.org/10.1145/3143361.3143369

[44] P. Cong, L. Li, J. Zhou, K. Cao, T. Wei, M. Chen, and S. Hu, “Developing User Perceived Value Based Pricing Models for Cloud Markets,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 12, pp. 2742–2756, Dec. 2018.

[45] T. Chen, R. Bahsoon, and X. Yao, "A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 61:1–61:40, Jun. 2018.

[46] X.-L. Huang, X. Ma, and F. Hu, "Editorial: Machine Learning and Intelligent Communications," *Mobile Networks and Applications*, vol. 23, pp. 68–70, Feb. 2018.

[47] T. Young, "Deployment challenges in multi-access edge computing (MEC)," 2020, accessed: 2020-12-21. [Online]. Available: https://www.a10networks.com/blog/deployment-challenges-in-multi-access-edge-computing-mec/

[48] Fortinet^{ő}, "Multi-access edge computing (MEC) and the edge cloud," 2020, accessed: 2020-12-21. [Online]. Available: https://www.fortinet.com/fr/solutions/mobile-carrier/securing-5g-innovation/mobile-edge-computing

[49] L. Kleinrock, "Time-shared systems: A theoretical treatment," *Journal of the ACM*, vol. 14, no. 2, pp. 242–261, Apr. 1967.

[50] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, UK, May 1989.

[51] G. Xu, J. Pang, and X. Fu, "A load balancing model based on cloud partitioning for the public cloud," *Tsinghua Science and Technology*, vol. 18, no. 1, pp. 34–39, 2013.

[52] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic load balancing on web-server systems," *IEEE Internet Computing*, vol. 3, no. 3, pp. 28–39, May 1999.

[53] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Dec. 2008, accessed: 2020-12-23. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[54] J. Li, J. Li, X. Chen, C. Jia, and W. Lou, "Identity-based encryption with outsourced revocation in cloud computing," *IEEE Trans. Comput.*, vol. 64, no. 2, pp. 425–437, Feb. 2015.

[55] X. Li, M. A. Salehi, Y. Joshi, M. K. Darwich, B. Landreneau, and M. Bayoumi, "Performance analysis and modeling of video transcoding using heterogeneous cloud services," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 4, pp. 910–922, Apr. 2019.

[56] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, and et al., "Ananta: Cloud scale load balancing," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 207â218, Aug. 2013.

[57] H. Qian, F. Li, R. Ravindran, and D. Medhi, "Optimal resource provisioning and the impact of energy-aware load aggregation for dynamic temporal workloads in data centers," *IEEE Trans. Netw. Service Manag.*, vol. 11, no. 4, pp. 486–503, Dec. 2014.

[58] S. Koenig and R. Simmons, "Complexity analysis of real-time reinforcement learning." in *AAAI*, Menlo-Park, CA, 1993, pp. 99–105.

[59] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *ISPASS*. IEEE, 2015, pp. 171–172.

[60] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Adv. Sci. Tech. Lett.*, vol. 66, no. 105-111, p. 2, 2014.

[61] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[62] M. Tegtmeier, "CPU utilization of multi-threaded architectures explained," 2015. [Online]. Available: https://blogs.oracle.com/solaris/cpu-utilization-of-multi-threaded-architectures-explained-v2

[63] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini, "An empirical study of hyper-threading in high performance computing clusters," *Linux HPC Revolution*, vol. 45, 2002.

[64] Y. Ding, E. D. Bolker, and A. Kumar, "Performance implications of hyper-threading," in *Int. CMG Conference*, 2003, pp. 21–29.

[65] C. D. Balaji Subramaniam. (2018) Future Highlight CPU Manager. [Online]. Available: https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager/

[66] C. Prakash, P. Prashanth, U. Bellur, and P. Kulkarni, "Deterministic container resource management in derivative clouds," in *IC2E*. IEEE, 2018, pp. 79–89.

[67] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma, "Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency," in *CCGrid*. IEEE, 2015, pp. 1–10.

[68] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *CLOUD*. IEEE, 2017, pp. 472–479.

[69] M. Boban, A. Kousaridas, K. Manolakis, J. Eichinger, and W. Xu, "Connected roads of the future: Use cases, requirements, and design considerations for vehicle-to-everything communications," *IEEE Veh. Technol. Mag.*, vol. 13, no. 3, pp. 110–123, 2018.

[70] S. Barmpounakis, G. Tsiatsios, M. Papadakis, E. Mitsianis, N. Koursioumpas, and N. Alonistioti, "Collision avoidance in 5G using MEC and NFV: The vulnerable road user safety use case," *Computer Networks*, vol. 172, p. 107150, 2020.

[71] F. A. Teixeira, V. F. e Silva, J. L. Leoni, D. F. Macedo, and J. M. Nogueira, "Vehicular networks using the IEEE 802.11p standard: An experimental analysis," *Vehicular Commun.*, vol. 1, no. 2, pp. 91–96, 2014.

[72] S. ÃncÃ¼, J. Ploeg, N. van de Wouw, and H. Nijmeijer, "Cooperative adaptive cruise control: Network-aware analysis of string stability," *IEEE Trans. Intell. Transp. Syst.*, vol. 15, no. 4, pp. 1527–1537, 2014.

[73] F. Dressler, F. Klingler, M. Segata, and R. Lo Cigno, "Cooperative driving and the tactile Internet," *Proc. IEEE*, vol. 107, no. 2, pp. 436–446, 2019.

[74] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2333–2345, 2018.

[75] A. Virdis, G. Nardini, and G. Stea, "A framework for MEC-enabled platooning," in *Proc. IEEE WCNCW*, 2019, pp. 1–6.

[76] C. Quadri, V. Mancuso, M. Ajmone Marsan, and G. P. Rossi, "Platooning on the edge," in *Proc. ACM MSWiM*, 2020. [Online]. Available: https://doi.org/10.1145/3416010.3423220

[77] K. Serizawa, M. Mikami, K. Moto, and H. Yoshino, "Field trial activities on 5g nr v2v direct communication towards application to truck platooning," in *2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall)*, 2019, pp. 1–5.

[78] L. Lv, Y. Shi, and W. Shen, "Mobility-as-a-service research trends of 5g-based vehicle platooning," pp. 1–3, 2021.

[79] R. Rajamani, Han-Shue Tan, Boon Kait Law, and Wei-Bin Zhang, "Demonstration of integrated longitudinal and lateral control for the operation of automated vehicles in platoons," *IEEE Trans. Control Syst. Technol.*, vol. 8, no. 4, pp. 695–708, 2000.

[80] J. Ploeg, B. Scheepers, E. van Nunen, N. van de Wouw, and H. Nijmeijer, "Design and experimental evaluation of cooperative adaptive cruise control," in *Proc. IEEE ITSC*, 2011, pp. 260–265.

[81] S. Santini, A. Salvi, A. S. Valente, A. PescapÃ©, M. Segata, and R. Lo Cigno, "A consensus-based approach for platooning with intervehicular communications and its validation in realistic scenarios," *IEEE Trans. Veh. Technol.*, vol. 66, no. 3, pp. 1985–1999, 2017.

[82] F. A. Salaht, F. Desprez, and A. Lebre, "An overview of service placement problem in fog and edge computing," *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3391196

[83] K. Velasquez, D. Abreu, M. Curado, and E. Monteiro, “Service placement for latency reduction in the internet of things,” *Proc. IEEE*, vol. 72, no. 2, pp. 105–115, 2017.

[84] M. Chen, W. Li, G. Fortino, Y. Hao, L. Hu, and I. Humar, “A dynamic service migration mechanism in edge cognitive computing,” *ACM Trans. Internet Technol.*, vol. 19, no. 2, Apr. 2019. [Online]. Available: https://doi.org/10.1145/3239565

[85] O. Tao, X. Chen, Z. Zhou, L. Li, and X. Tan, “Adaptive user-managed service placement for mobile edge computing via contextual multi-armed bandit learning,” *IEEE Transactions on Mobile Computing*, 2021.

[86] M. Segata, B. Bloessl, S. Joerer, C. Sommer, M. Gerla, R. Lo Cigno, and F. Dressler, “Toward communication strategies for platooning: Simulative and experimental evaluation,” *IEEE Trans. Veh. Technol.*, vol. 64, no. 12, pp. 5411–5423, 2015.

[87] M. Lauridsen, L. C. Gimenez, I. Rodriguez, T. B. Sorensen, and P. Mogensen, “From LTE to 5G for connected mobility,” *IEEE Commun. Mag.*, vol. 55, no. 3, pp. 156–162, 2017.

[88] H.-S. Tan, R. Rajamani, and W.-B. Zhang, “Demonstration of an automated highway platoon system,” in *Proceedings of the 1998 American control conference. ACC (IEEE Cat. No. 98CH36207)*, vol. 3. IEEE, 1998, pp. 1823–1827.

[89] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. ElBakoury, “Ultra-low latency (ull) networks: The ieee tsn and ietf detnet standards and related 5g ull research,” *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 88–145, 2019.

[90] D. Krajzewicz, G. Hertkorn, C. Rössel, and P. Wagner, “SUMO (simulation of urban mobility) - an open-source traffic simulation,” in *4th Middle East Symposium on Simulation and Modelling*, A. Al-Akaidi, Ed., 2002, pp. 183–187.

[91] C. Sommer, R. German, and F. Dressler, “Bidirectionally coupled network and road traffic simulation for improved IVC analysis,” *IEEE Trans. Mobile Comput.*, vol. 10, no. 1, pp. 3–15, 2011.

[92] G. Nardini, A. Virdis, G. Stea, and A. Buono, “SimuLTE-MEC: extending SimuLTE for multi-access edge computing,” in *Proc. OMNeT++ summit*, 2018.

[93] R. Rajamani, *Vehicle Dynamics and Control*, 2nd ed. Springer, 2012.

[94] R. K. Jain, D.-M. W. Chiu, W. R. Hawe *et al.*, “A quantitative measure of fairness and discrimination,” *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984.