AI-driven Zero-Touch solutions for resource management in cloud-native 5G networks

Menuka Perera Jayasuriya Kuranage

▶ To cite this version:

Menuka Perera Jayasuriya Kuranage. AI-driven Zero-Touch solutions for resource management in cloud-native 5G networks. Networking and Internet Architecture [cs.NI]. Ecole nationale supérieure Mines-Télécom Atlantique, 2024. English. NNT : 2024IMTA0427 . tel-04929866

HAL Id: tel-04929866
https://theses.hal.science/tel-04929866v1

Submitted on 5 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

COLLEGES BRETAGNE LOIRE | SCIENCES POUR L'INGENIEUR ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
MINES-TÉLÉCOM ATLANTIQUE BRETAGNE PAYS DE LA LOIRE –
IMT ATLANTIQUE

ÉCOLE DOCTORALE Nº 648
Sciences pour l'Ingénieur et le Numérique
Spécialité : *Télécommunication*

Par
Menuka PERERA JAYASURIYA KURANAGE

AI-Driven Zero-Touch Solutions for Resource Management in Cloud-Native 5G Networks

Thèse présentée et soutenue à IMT Atlantique, Rennes, 18 Novembre 2024
Unité de recherche : Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA)
Thèse Nº : 2024IMTA0427

Rapporteurs avant soutenance :

Nadjib AIT SAADI	Professor, Université Paris-Saclay / UVSQ
Jean-Louis ROUGIER	Professor, Télécom Paris

Composition du Jury :

Président :	Salah-Eddine EL AYOUBI	Professor, Centrale Supélec
Examinateurs :	Nadjib AIT SAADI	Professor, Université Paris-Saclay / UVSQ
	Jean-Louis ROUGIER	Professor, Télécom Paris
	Kandaraj PIAMRAT	Maître de conférences , Université de Nantes
	Philippe BERTIN	Chargé de recherche, Orange/ b<>com
Encadrants :	Ahmed BOUABDALLAH	Maître de conférences, IMT Atlantique/ IRISA
	Elisabeth HANSER	Chargé de recherche, b<>com
Dir. de thèse :	Loutfi NUAYMI	Professor, IMT Atlantique/ IRISA

ACKNOWLEDGEMENT

First and foremost, I would like to express my heartfelt gratitude to my thesis director, Loutfi Nuaymi, for his unwavering guidance, encouragement, and invaluable expertise throughout my PhD journey. His support and constructive feedback have been instrumental in shaping both the theoretical and practical aspects of this work. I am equally grateful to my thesis supervisors, Ahmed Bouabdallah, Elisabeth Hanser, Philippe Bertin, and Thomas Ferrandiz, for their insightful advice, technical guidance, and constant encouragement. Their diverse perspectives and expertise have significantly enriched this thesis, helping me navigate complex challenges and achieve meaningful results.

I would like to extend my sincere thanks to EXFO for funding this research and for closely following the progress of my work throughout the project. Their support has been vital in enabling this study and ensuring its successful completion.

I am deeply appreciative of my friends and colleagues at IRT b<>com for their camaraderie, encouragement, and practical support during this journey. Their willingness to collaborate, provide feedback has made this experience both productive and enjoyable.

Finally, and most importantly, I would like to thank my family for their unwavering love and support. To my parents, who have always believed in me and encouraged me to pursue my dreams, and to my sibling, who have provided endless moral support—thank you for your patience, understanding, and belief in my abilities. This thesis would not have been possible without your constant reassurance and faith in me.

Table of Contents

List of Figures **9**

List of Tables **13**

Résumé en français **15**

Abstract **23**

Abbreviations **25**

1 Introduction **31**
Motivations . . . 32
Contributions . . . 34
Organization of the Manuscript . . . 36

2 Background **39**
2.1 5G network & main challenges . . . 40
2.1.1 Service-based architecture . . . 40
2.1.2 5G Core network . . . 41
2.2 Network automation . . . 44
2.2.1 ETSI ZSM Framework . . . 47
2.2.2 Closed loop automation . . . 50
2.3 Cloud computing . . . 51
2.3.1 Cloud models . . . 52
2.3.2 Cloud service models . . . 53
2.3.3 Cloud pricing . . . 55
2.3.4 Cost model . . . 56
2.4 Kubernetes for CNF management . . . 56
2.4.1 System architecture . . . 57
2.4.2 Resource management . . . 60
2.4.3 Dynamic resource allocation . . . 62

2.5 Summary 64

3 State of the Art 67

3.1 Introduction 67
3.2 Rule based autoscaling 73
3.3 Queuing theory based autoscaling 82
3.4 Control based autoscaling 87
3.5 Reinforcement learning based autoscaling 92
3.6 Prediction based autoscaling 99
3.7 Summary 106

4 Resource usage forecasting for CNFs in Kubernetes environment 109

4.1 Introduction 109
4.2 Research challenge 110
4.2.1 Resource usage profiles of 5GC network functions 111
4.2.2 High level vs low level metrics 116
4.3 Proposed system model 120
4.3.1 Resource usage forecasting 121
4.3.2 Model selection 122
4.3.3 Data collection 123
4.3.4 Data pre-processing, model design and training 125
4.4 Results 126
4.4.1 Evaluation metrics 126
4.4.2 Prediction accuracy 127
4.5 Summary 130

5 AI-assisted proactive autoscaling solution for CNFs 133

5.1 Introduction 133
5.2 Research challenge 134
5.3 Dynamic scaling model 136
5.3.1 Scaling-up 136
5.3.2 Scaling-down 138
5.3.3 No scaling 140
5.3.4 Decision timing 140
5.3.5 Testbed 142

5.4 Results . . . 143
5.4.1 KPIs for the evaluation . . . 143
5.4.2 Benchmarking the autoscaling solution . . . 144
5.4.3 Autoscaler comparison . . . 145
5.5 Summary . . . 150

6 CPU Throttling aware autoscaling 151
6.1 Introduction . . . 151
6.2 Research challenge . . . 152
6.3 CPU throttling aware triggering . . . 161
6.3.1 CPU throttling forecasting . . . 162
6.3.2 Trigger module integration . . . 164
6.4 Results . . . 166
6.4.1 Forecasting model evaluation . . . 166
6.4.2 Autoscaling solution evaluation . . . 167
6.5 Summary . . . 170

7 Conclusion 173
Summary . . . 173
Future Directions . . . 174

Publications 177

Bibliography 179

LIST OF FIGURES

1.1 Annual average cloud expenditure in top cloud service providers - AWS, Azure, and Google - 2019. (from [2]) 33
1.2 Network automation maturity model (based on [3]) 34

2.1 5G Architecture. (based on [10]) 41
2.2 Network slicing with 5G network. (from [23]) (Copyright © 2017 IEEE) . . 45
2.3 Multi domain, multi tenant network slicing. (from [24] - modified) (Copyright © 2018 IEEE) 46
2.4 ETSI ZSM reference architecture. (from [33]) 48
2.5 E2E Communication service automation with ETSI ZSM framework. (from [33] - modified) 49
2.6 ETSI Closed loop stages within ZSM framework compared to OODA model. (from [33] - modified) 50
2.7 Comparison between different cloud service models. (based on [48]) 54
2.8 Kubernetes architecture. (based on [60]) 58

3.1 The taxonomy for autoscaling web applications in clouds. (from [72]) . . . 68
3.2 High level overview of three tier architecture. (from [76] - modified) (Copyright © 2016 IEEE) 69
3.3 MEC enabled 5G IoT architecture. (from [86]) (Copyright © 2020 IEEE) . 74
3.4 (a) KHPA in Kubernetes-based edge computing architecture and (b) THPA in Kubernetes-based edge computing architecture. (from [89]) (Copyright © 2022 IEEE) 76
3.5 ELASTICDOCKER architecture. (from [80]) (Copyright © 2017 IEEE) . . 77
3.6 Combining vertical and horizontal scaling. (from [92]) (Copyright © 2020 IEEE) 79
3.7 A simplified queuing model of the system. (from [96]) (Copyright © 2018 IEEE) 82

3.8 (a) Single-service application composed of one service which may contain different modules. (b) Multi-service application whose invoking relationships of services. (from [97]) (Copyright © 2021 IEEE) 83
3.9 The relationship between supply of resource and service performance. (from [98]) (Copyright © 2021 IEEE) 85
3.10 LTE CP queuing model. (from [99]) (Copyright © 2018 IEEE) 86
3.11 PID feedback control loop. (from [76]) (Copyright © 2016 IEEE) 88
3.12 Autoscaling architecture with the PID controller. (from [101]) (Copyright © 2022 IEEE) 89
3.13 Overview of QoS management control system. (from [102]) (Copyright © 2018 IEEE) 90
3.14 Custom membership function to define the status of the input metrics. (from [103]) (Copyright © 2018 IEEE) 91
3.15 SQLR Block diagram: 'LB' represents the Load Balancer agent, while 'AC' denotes the Admission Control agent. (from [106]) (Copyright © 2021 IEEE) 93
3.16 Proposed system architecture.(from [107]) (Copyright © 2022 IEEE) 95
3.17 The system architecture of autoscaling framework. (from [109]) 96
3.18 Proposed autoscaler system data flow. (from [112]) (Copyright © 2019 IEEE) 100
3.19 Requested CPU for each scaling approach. (from [113]) 101
3.20 Experimental results. (from [78]) (Copyright © 2018 IEEE) 103
3.21 System architecture for the under-study cloud-native network. (from [114]) (Copyright © 2021 IEEE) 104
3.22 Prediction-based autoscaling system architecture. (from [115]) 105

4.1 Kubernetes testbed with 5G CN. 113
4.2 5G CN resource usage profiles. 114
4.3 Resource usage profile of different applications. (from [125]) (Copyright © 2021 IEEE) 117
4.4 CPU usage profiles of the web application under different t_{base} values. (from [4]) (Copyright © 2022 IEEE) 119
4.5 High level architecture of the proposed solution. 120
4.6 Proposed solution with closed loop architecture. (from [5]) (Copyright © 2023 IEEE) 121
4.7 Data collection testbed. (from [4] - modified) (Copyright © 2022 IEEE) . . 124

4.8 RMSE values for forecasting horizons from 1 to 12. (from [4]) (Copyright © 2022 IEEE) . . . 127
4.9 MAE values for forecasting horizons from 1 to 12. (from [4]) (Copyright © 2022 IEEE) . . . 129

5.1 Selecting the time step to scale in the CPU usage forecast. (from [5]) (Copyright © 2023 IEEE) . . . 135
5.2 Dynamic scaling model decision space. (from [5]) (Copyright © 2023 IEEE) 137
5.3 Average CPU usage post scale down must not surpass the remaining replica resource capacity. . . . 139
5.4 Autoscaling decision-making process. . . . 140
5.5 Testbed for AI-assisted scaling solution. (from [5]) (Copyright © 2023 IEEE) 142
5.6 Service response time during scaling. (from [5]) (Copyright © 2023 IEEE) . 148
5.7 Pod count and pod operational time. (from [5]) (Copyright © 2023 IEEE) . 149

6.1 Effects of CPU throttling on application processing time – single-threaded application. (based on [144]) . . . 153
6.2 Comparison of processing time variation with and without CPU throttling. 156
6.3 Influence of incoming request rate, CPU usage on CPU throttling. (from [6]) (Copyright © 2024 IEEE) . . . 159
6.4 Proposed CPU throttling aware autoscaling architecture. (from [6]) (Copyright © 2024 IEEE) . . . 165
6.5 RMSE and MAE values of CPU throttling forecasting models. (from [6]) (Copyright © 2024 IEEE) . . . 167
6.6 Service response time during scaling. (from [6]) (Copyright © 2024 IEEE) . 170
6.7 Operational time comparison. (from [6]) (Copyright © 2024 IEEE) . . . 171

LIST OF TABLES

3.1 Rules in fuzzy database. (from [103]) (Copyright © 2018 IEEE) 92

4.1 Cluster resource specifications. (from [4]) (Copyright © 2022 IEEE) 111

RÉSUMÉ EN FRANÇAIS

L'industrie des télécommunications a connu des transformations profondes depuis le début du XIXe siècle, encommençant par la commercialisation du télégraphe. Cette innovation pionnière a posé les bases de l'exploitation dans un contexte économique de la communication longue distance , redéfinissantfondamentalement les méthodologies de transmission de l'information. L'inventionultérieure du téléphoneaaccéléré le développement du secteurenpermettant la communication vocale, favorisantuneconnectivitéinterpersonnelle sans précédent.

Une avancée majeure s'estproduite à la fin des années 1980 avec l'avènement de l'internet, facilitantl'échangerapided'informations à l'échellemondiale et remodelant les paradigmes de communication. Aujourd'hui, les infrastructures modernes de télécommunicationsontévoluéen réseaux sophistiqués , offrantune large gamme de services à l'ensemble des secteurs des activités humaines (santé, finance, divertissement, . . .). Cessystèmes constituent la clé de voute de l'infrastructuremondiale de l'économie numérique en permettantuneconnectivitémondiale systématique.

Le déploiement actuel des réseaux de cinquièmegénération (5G) devrait considérablementélargir les capacités de service et les scénariosd'application, amplifiantainsil'impactéconomique et societal des infrastructures numériques. La technologie 5G promet notamment des vitessesmontantesallantjusqu'à 10 Gbps et des vitessesdescendantesallantjusqu'à 20 Gbps, enexploitant les fréquencesmillimétriques (au-dessus de 6 GHz) et enincorporant des technologies avancéestelles que le Massive Multiple Input Multiple Output (MIMO) et le beamforming au sein du réseau d'accès radio (RAN).

En plus de sesavancéestechnologiques, la 5G intègre des systèmes de gestion automatisés entièrementnatifs pour le cloud , introduisant de nouvelles architectures réseau. Ces innovations permettentunecapacité accrue, unelatenceréduite, des performances améliorées et des coûtsd'exploitationinférieurs par rapport aux générationsprécédentes, positionnant la 5G comme un élémentfondamental des infrastructures de télécommunications futures.

Les capacitésaméliorées de la 5G sontappelées à transformer de nombreuxsecteurs, notamment la diffusion multimédia à haut débit, les systèmes de transport intelligents,

l'agriculture de précision, les solutions de santé avancées, les environnements de travail à distance, l'automatisationindustrielle (Industrie 4.0) et l'industrie des jeux enligne. Ces applications devraientapporter des avantagessubstantiels au bien-êtrehumain tout engénérantune contribution significative à la croissanceéconomiquemondiale.

Les projections estiment qu'environ 70 % du trafic mobile sera consacré au streaming vidéo, nécessitant des réseaux à haute capacité et une connectivité ininterrompue pour offrir une expérience utilisateur fluide. De plus, l'adoption croissante d'objets connectés (Internet des objets - IoT)—utilisés dans les maisons intelligentes, les environnements de travail intelligents et la surveillance environnementale agricole— entraîne une augmentation significative des communications machine-à-machine (M2M).

Les applications avancées telles que la conduite autonome et la chirurgie à distance imposent des contraintes très fortes , nécessitant une communication à ultra-faible latence pour répondre aux exigences strictes de performances en temps réel. Ces cas d'utilisation variés imposent des demandes complexes et diversifiées sur les performances des réseaux, soulignant la nécessité de solutions de communication personnalisées. Ces solutions doivent répondre efficacement aux besoins spécifiques en matière de qualité d'expérience (QoE), de qualité de service (QoS) et de sécurité robuste pour garantir des performances optimales dans toutes les applications.

L'adoption de l'architecture nativement basée sur le cloud marque une avancée significative dans l'industrie des télécommunications, en particulier avec la mise en œuvre de la 5G. Traditionnellement, les réseaux télécoms reposaient sur une infrastructure matérielle dédiée et ainsi que des logiciels propriétaires, limitant leur adaptabilité.

En adoptant les technologies natives du cloud, les fournisseurs de services de communication (CSP) peuvent tirer parti de la scalabilité, de l'élasticité et de l'agilité inhérentes à l'infrastructure cloud. Cette transition non seulement rationalise la prestation de services, mais améliore également l'efficacité opérationnelle et la flexibilité réseau.

Bien que la transition vers l'infrastructure cloud offre de nombreux avantages, elle ne garantit pas nécessairement une réduction des coûts. Sans stratégie efficace de gestion et d'optimisation, l'adoption du cloud peut entraîner une augmentation des dépenses opérationnelles. De nombreuses industries ayant migré leurs opérations informatiques vers le cloud indiquent des coûts plus élevés que prévus, souvent en raison d'inefficacités dans l'utilisation des ressources. Les études sur la gestion des coûts du cloud révèlent qu'en moyenne, 35 % des ressources cloud allouées aux trois principaux fournisseurs de services cloud sont gaspillées chaque année. Ces ressources inutilisées ou sous-utilisées contribuent

à des coûts opérationnels disproportionné et à une rentabilité réduite.

Les principaux facteurs à l'origine de ce gaspillage de ressources comportent une compréhension limitée des besoins en charge de travail, une visibilité insuffisante sur les déploiements cloud et des difficultés à identifier avec précision les besoins critiques en ressources telles que le calcul, la mémoire et le stockage. Ce manque de visibilité entraîne souvent une allocation de ressources sous-optimale. Des ressources insuffisantes dégradent les performances des applications, affectant directement la QoS. Pour atténuer cette dégradation, les organisations surprovisionnent fréquemment les ressources, ce qui conduit à d'autres inefficacités et à une augmentation des coûts. Cette dynamique illustre le compromis inévitable entre l'optimisation des coûts et la QoS dans les environnements cloud.

L'optimisation des coûts dans les infrastructures cloud est un défi complexe et multidimensionnel. Elle est influencée par divers facteurs, notamment les types de modèles de déploiement cloud (public, privé ou hybride), les structures tarifaires variées, la conception architecturale des applications côté client et l'intégration des composants middleware. Chaque implémentation cloud est unique et nécessite des stratégies d'optimisation des coûts sur mesure pour équilibrer efficacité opérationnelle et rentabilité. En l'absence d'une approche universelle, les organisations doivent évaluer et prendre en compte les variables et contraintes spécifiques à leurs déploiements cloud.

La gestion manuelle des ressources et l'optimisation des coûts sont des processus exigeants en termes de ressources et sujets à des inefficacités et des erreurs. La nature dynamique et évolutive des environnements cloud complique encore ces tâches, rendant l'automatisation essentielle pour une gestion efficace de ces coûts . Les solutions automatisées permettent une surveillance en temps réel, une allocation intelligente des ressources et des ajustements proactifs pour mettre en adéquation usage des ressources et besoins des charges de travail. En s'appuyant sur l'automatisation, les organisations peuvent minimiser le gaspillage des ressources, améliorer la QoS et réaliser des réductions significatives des dépenses d'investissement (CapEx) et des dépenses opérationnelles (OpEx), maximisant ainsi la valeur de leurs investissements cloud.

La progression vers une automatisation complète dans les télécommunications implique l'adoption de cadres d'automatisation autonomes dotés de capacités d'auto-réparation et d'auto-optimisation. Ces systèmes avancés permettent aux réseaux d'identifier, diagnostiquer et résoudre les problèmes de manière autonome tout en optimisant les performances sans intervention externe. Cependant, le déploiement d'une telle automatisation sophis-

tiquée dans le paysage complexe et dynamique des réseaux 5G présente des défis importants en raison des complexités architecturales et des exigences de performance diversifiées de la 5G.

Les organismes de normalisation, notamment le 3rd Generation Partnership Project (3GPP) et l'Institut européen des normes de télécommunication (ETSI), abordent ces défis en promouvant une gestion de réseau entièrement automatisée et en normalisant de nouvelles architectures . Ces efforts visent à garantir l'interopérabilité, la fiabilité et la cohérence des déploiements 5G à l'échelle mondiale.

Les avancées en intelligence artificielle (IA) et en apprentissage automatique (ML) sont essentielles à la réalisation de réseaux entièrement automatisés. L'IA et le ML fournissent des analyses intelligentes, des capacités prédictives et une prise de décision autonome, constituant la base des opérations réseau automatisées. Ces technologies améliorent les performances en permettant une allocation dynamique des ressources, une détection des anomalies en temps réel et une optimisation proactive.

Dans les environnements cloud, les techniques d'IA/ML jouent un rôle crucial pour résoudre le compromis entre coût et QoS. En prévoyant les charges de travail et en estimant les coûts, l'IA/ML facilite une allocation dynamique des ressources, garantissant des performances optimales et une efficacité des coûts. Cette intégration soutient le double objectif de maintenir une QoS élevée tout en minimisant les coûts opérationnels. Grâce à l'automatisation pilotée par l'IA/ML, les réseaux 5G peuvent atteindre des niveaux inégalés d'efficacité, de scalabilité et d'adaptabilité, pour devenir ainsi les fondements des infrastructures de télécommunications de prochaine génération.

La réduction du gaspillage des ressources cloud, un facteur important de l'augmentation des dépenses cloud, nécessite la mise en œuvre de stratégies d'allocation dynamique de ressources, améliorées par l'automatisation. Une gestion efficace des ressources implique de comprendre les besoins des applications et d'allouer les ressources de manière précise et rapide. Cette approche permet de réduire les risques de surallocation et de sous-allocation, atteignant ainsi un équilibre optimal entre coûts opérationnels et QoS.

Pour répondre au compromis coût-QoS dans les environnements cloud, en particulier pour les fonctions réseau 5G (NFs), cette thèse propose une nouvelle solution proactive d'autoscaling adaptée aux fonctions réseau natives pour le cloud (CNFs) déployées dans des environnements Kubernetes. L'étude a identifié plusieurs défis inhérents à ce domaine, notamment la nécessité d'analyser et de prévoir les futurs modèles d'utilisation des ressources des NFs basées sur le cloud, de permettre une prise de décision proac-

tive et éclairée, et d'évaluer l'influence du matériel et des middleware sous-jacents sur les stratégies de mise à l'échelle.

La première étape de la solution proposée, met l'accent sur uneprévisionprécise de l'utilisation des ressources des CNFs, essentielle pour éviter à la fois la surestimation et la sous-estimation des besoins. Une surestimationentraîneune allocation de ressources inutile et des coûtsaccrus, tandisqu'une sous-estimation peutdégrader les performances des applications, affectantnégativement la QoS. La prévision des ressourcesen temps réelfournit des informationsexploitablesessentielles pour l'autoscaling. Cependant, prédire les demandesenressources à l'avance,reste un défimajeur. Cette complexitédécoule de la relation complexe entre les charges de travail entrantes et la consommationréelle des ressources des CNFs, qui peut varier considérablementenfonction des caractéristiques des charges de travail et du comportement des applications.

Pour relever ce défi, une nouvelle méthodologie de prévision de l'utilisation des ressources, spécialement conçue pour les CNFs déployées dans des environnements Kubernetes, est introduite. Cette méthodologie prend en compte l'interaction complexe entre les exigences des charges de travail et les modèles de consommation des ressources. Elle utilise une technique de prévision des séries temporelles multivariées exploitant des modèles d'apprentissage profond pour capturer et prévoir avec précision l'utilisation future des ressources des CNFs.

L'approche proposée fournit des prévisions fiables, de la consommation des ressources sur des horizons temporels étendus, permettant des analyses en temps réel qui soutiennent des décisions d'autoscaling proactives. Les évaluations empiriques montrent que cette méthode de prévision améliore les approches traditionnelles, qui se basent généralement uniquement sur des données au niveau du système ou de l'application, en termes de précision et de fiabilité.

La deuxième phase de la solution d'autoscaling consiste à analyser les prévisions d'utilisation du CPU générées lors de la phase initiale afin de déterminer les actions de mise à l'échelle appropriées. Cela inclut le calcul du nombre optimal de répliques et la planification du moment de ces opérations de mise à l'échelle pour garantir une allocation efficace des ressources tout en maintenant les niveaux de QoS souhaités. En intégrant une prévision précise avec des actions de mise à l'échelle stratégiques, la solution équilibre efficacement rentabilité et performance, répondant ainsi au compromis critique entre coût et QoS dans les environnements 5G basés sur le cloud.

Le processus de prise de décision pour l'autoscaling dans les environnements cloud est

intrinsèquement complexe et pose des défis importants. Même avec des prévisions précises de l'utilisation du CPU, les décisions de mise à l'échelle peuvent perturber l'équilibre délicat entre coût et QoS. Malgré l'utilisation de valeurs prévues et de stratégies proactives, des problèmes similaires à ceux observés dans les approches réactives basées sur des seuils peuvent encore survenir. Ces problèmes incluent l'identification incorrecte des actions de mise à l'échelle ou une mauvaise estimation du nombre requis de répliques, ce qui peut entraîner une surallocation ou une sous-allocation des ressources.

Un autre défi réside dans le moment des décisions de mise à l'échelle, qui impacte de manière critique l'équilibre entre coût et QoS. Les décisions de mise à l'échelle reposent sur l'utilisation future prévue du CPU, et la latence inhérente entre l'initiation d'une réplique et sa disponibilité opérationnelle, introduit des sensibilités temporelles. Des décisions prises prématurément peuvent entraîner une allocation de ressources inutile, tandis que des actions retardées risquent de ne pas prévenir la dégradation des performances. Ainsi, garantir un moment optimal est essentiel pour maintenir à la fois l'efficacité des coûts et la QoS.

Le cœur de cette phase réside dans le développement d'un mécanisme avancé d'autoscaling combinant des stratégies basées sur des seuils statiques et dynamiques pour différencier divers scénarios de mise à l'échelle. Ce mécanisme intègre des techniques expérimentales visant à optimiser le moment des décisions de mise à l'échelle, garantissant que les actions sont exécutées aux moments les plus efficaces. En surmontant les limitations des capacités natives d'autoscaling de Kubernetes et en atténuant les oscillations décisionnelles, problème courant dans les systèmes traditionnels basés sur des seuils, la solution proposée renforce la stabilité et la robustesse du processus d'autoscaling.

Cette méthodologie permet des décisions de mise à l'échelle en temps réel, basées sur l'utilisation prédite des ressources, minimisant les coûts opérationnels et réduisant la dégradation de la QoS pendant les événements de mise à l'échelle. Les résultats empiriques montrent que cette approche proactive d'autoscaling atteint un compromis coût-QoS supérieur par rapport aux méthodes alternatives. La solution équilibre efficacement l'exploitation des ressources et la qualité de service, surmontant les limitations traditionnelles et faisant progresser les capacités de mise à l'échelle dans les environnements natifs pour le cloud.

Malgré ses performances supérieures à la solution d'autoscaling par défaut de Kubernetes , le mécanisme d'autoscaling proposé a rencontré une augmentation des temps de réponse des services lors des événements de mise à l'échelle ascendante. Ce problème était

principalement dû à un moment sous-optimal des décisions au sein du mécanisme de mise à l'échelle. Bien que les actions de mise à l'échelle ascendante aient été initiées de manière proactive, les délais d'exécution ont conduit à une dégradation de la QoS. L'analyse a révélé que, bien que les répliques aient été provisionnées avant que le pod affecté atteigne sa capacité CPU, les temps de réponse des services s'étaient déjà détériorés.

La cause sous-jacente de cette dégradation de la QoS a été attribuée au mécanisme de limitation du CPU inhérent à la gestion des ressources au niveau du système. Ce mécanisme, mis en œuvre dans les environnements Linux pour réguler et limiter la surutilisation des ressources par les pods, a involontairement impacté les calculs de temporisation nécessaires à des décisions de mise à l'échelle efficaces. En conséquence, il a perturbé l'équilibre entre les coûts opérationnels et la QoS, mettant en évidence les limites de la solution d'autoscaling initiale.

Pour pallier cette limitation, un mécanisme de déclenchement amélioré a été développé en tant qu'extension de la solution d'autoscaling proposée. Ce mécanisme amélioré ajuste dynamiquement le moment des décisions de mise à l'échelle en tenant compte des effets de la limitation du CPU. En prévoyant les futurs événements potentiels de limitation du CPU, il fournit des informations exploitables permettant des ajustements proactifs de la mise à l'échelle, atténuant ainsi les augmentations des temps de réponse des services causées par cette limitation.

Cette approche améliorée vise à minimiser la dégradation de la QoS lors des événements de mise à l'échelle, améliorant considérablement les performances des CNFs dans les environnements 5G natifs pour le cloud. La solution a été validée dans un déploiement Kubernetes en conditions réelles, avec des résultats comparés à ceux des expériences précédentes. Les résultats ont démontré l'efficacité du mécanisme amélioré pour maintenir un compromis coût-QoS plus optimal tout en réduisant la dégradation des temps de réponse des services lors des processus de mise à l'échelle ascendante.

Les conclusions de cette thèse représentent une contribution significative au domaine de la gestion des réseaux 5G natifs pour le cloud, en mettant particulièrement l'accent sur la résolution du compromis critique entre coût et QoS. En explorant les subtilités de l'allocation des ressources, des mécanismes d'autoscaling et de l'optimisation des performances dans des environnements basés sur Kubernetes, cette recherche présente un cadre solide et complet. Ce cadre est conçu pour être adopté par les fournisseurs de services de communication (CSP) afin d'améliorer l'efficacité opérationnelle et la scalabilité de leurs pratiques de gestion de réseau.

Les contributions de ce travail font progresser l'état de l'art dans la gestion des architectures natives pour le cloud, en dotant les CSP d'outils et de méthodologies innovants pour relever efficacement les défis complexes posés par les réseaux 5G. En introduisant des solutions équilibrant stratégiquement les coûts opérationnels et la QoS, cette thèse facilite la création d'infrastructures de réseau résilientes, performantes et économiquement durables. Ces avancées sont essentielles pour façonner la prochaine génération de connectivité mondiale, garantissant que les réseaux restent adaptables et efficaces dans un paysage technologique de plus en plus dynamique.

ABSTRACT

Recent technological advancements have spurred significant enhancements across sectors such as healthcare, media, and agriculture, compelling wireless network operators to elevate their offerings with highly dependable, low-latency, large-scale networking solutions. This transformation is driven by the advent of the 5th Generation (5G) network, designed to meet escalating demands and usher in novel applications like Augmented Reality (AR), Virtual Reality (VR), digital twins, Industry 4.0, etc. Consequently, 5G has fundamentally reshaped network traffic patterns and intensified the complexities of network management.

A pivotal feature of 5G is its shift from traditional monolithic network architectures dependent on Physical Network Functions (PNFs) to Cloud-native network functions (CNFs) in cloud-native environments, promising superior performance and flexibility. This evolution has underscored the growing complexity of network management, necessitating a shift towards automation. Recognizing this need, the European Telecommunications Standards Institute's (ETSI) Zero-touch Network and Service Management (ZSM) working group has proposed a standardized framework to automate modern telecom network operations using closed-loop architecture.

One of the primary challenges in managing 5G networks lies in dynamically scaling network functions within cloud-native settings. Effective network management tools, coupled with automation can optimize resource allocation by ensuring Quality of Service (QoS) and reducing deployment costs. Kubernetes, a leading container management and orchestration tool, has gained popularity among telecom providers for its adaptability and robust performance in 5G environments. Its Horizontal Pod Autoscaler (HPA) scales CNFs based on workload demands to maintain high availability and performance, yet aggressive cost-saving measures like this could impact deployment efficiency.

This research tackles the critical trade-off between cost and QoS in resource autoscaling for cloud-native 5G networks. We achieve this by developing advanced algorithms that enhance scaling actions within the Kubernetes ecosystem. By leveraging deep learning, real-time analytics, and predictive modeling, our approach improves the responsiveness and precision of autoscaling mechanisms within the framework of ZSM's closed-loop ar-

chitecture. Utilizing closed-loop systems and adaptive control strategies, our methodology significantly elevates autoscaling performance under real-world operational scenarios.

Abbreviations

5G 5th Generation

3GPP 3rd Generation Partnership Project

AI Artificial Intelligence

AMF Access and Mobility Management Function

API Application Programming Interface

AR AutoRegressive

ARIMA AutoRegressive Integrated Moving Average

AUSF Authentication Server Function

AWS Amazon Web Services

BiLSTM Bi-directional Long Short-Term Memory

CaaS Container as a Service

CapEx Capital Expenditures

CD Continuous Deployment

CFS Complete Fair Scheduler

CI Continuous Integration

CN Core Network

CNCF Cloud-Native Computing Foundation

CNF Cloud-Native Network Function

CNN Convolutional Neural Networks

CPU Central Processing Unit

CRIU Checkpoint Restore in User space

CSP Communication Service Provider

DDoS Distributed Denial of Service

DNN Deep Neural Network

DPDK Data Plane Development Kit

DRL Deep Reinforcement Learning

E2E End-to-End

EC2 Elastic Compute Cloud

EKS Elastic Kubernetes Service

EMA Exponential Moving Average

EPC Evolved Packet Core

ETSI European Telecommunications Standards Institute

FFRLS Forgetting Factor Recursive Least Squares

GCP Google Cloud Platform

GKE Google Kubernetes Engine

GNN Graph Neural Network

GRU Gated Recurrent Units

GTP-U General Packet Radio Service Tunneling Protocol-User Plane

HPA Horizontal Pod Autoscaling

HTTP Hypertext Transfer Protocol

HW Holt-Winters

IaaS Infrastructure as a Service

IoT Internet of Things

IP Internet Protocol

JQN Jackson Queuing Network

KPI Key Performance Indicators

LSTM Long Short-Term Memory

LTE Long-Term Evolution

MA Moving Average

MAE Mean Absolute Error

MANO Management and Orchestration

MAPE-K Monitor, Analyze, Plan, Execute, Knowledge

MARIMA Multivariate AutoRegressive Integrated Moving Average

MCMC Markov Chain Monte Carlo

MIMO Multiple Input Multiple Output

ML Machine Learning

MME Mobility Management Entity

NaaS Network as a Service

NAS Non-Access Stratum

NF Network Function

NFV Network Functions Virtualization

NFVO Network Functions Virtualization Orchestrator

NGAP Next Generation Application Protocol

NG-RAN Next Generation Radio Access Network

NRF Network Repository Function

NSA Non-Standalone

NSSF Network Slice Selection Function

NWDAF Network Data Analytics Function

ONAP Open Network Automation Platform

ONF Open Network Foundation

OODA Observe, Orient, Decide, Act

OOM Out-Of-Memory

OpEx Operational Expenditures

OS Operating System

PaaS Platform as a Service

PCF Policy Control Function

PDCP Packet Data Convergence Protocol

PFCP Packet Forwarding Control Protocol

PGW Packet Data Network Gateway

PID Proportional-Integral-Derivative

QoE Quality Of Experience

QoS Quality of Service

QPS Queries Per Second

RAN Radio Access Network

REST Representational State Transfer

RL Reinforcement Learning

RMSE Root Mean Square Error

RNN Recurrent Neural Network

SaaS Software as a Service

SARIMA Seasonal AutoRegressive Integrated Moving Average

SARIMAX Seasonal AutoRegressive Integrated Moving Average with Exogenous Regressors

SBA Service-Based Architecture

SCTP Stream Control Transmission Protocol

SDN Software-Defined Networking

SGW Serving Gateway

SLA Service Level Agreements

SMA Simple Moving Average

SMF Session Management Function

SON Self-Organizing Network

UDM Unified Data Management

UDR Unified Data Repository

UE User Equipment

UP User Plane

UPF User Plane Function

VAR Vector AutoRegressive

VM Virtual Machine

VNF Virtual Network Function

VPA Vertical Pod Autoscaling

VPP Vector Packet Processing

VU Virtual User

ZSM Zero Touch Network and Service Management

Chapter 1

Introduction

The telecommunications sector has undergone substantial transformation since the early 19th century, marked by the commercialization of the first telegraph service. This foundational development established the basis for commercial long-distance communication, fundamentally altering information transmission methodologies. The subsequent invention of the telephone further propelled the industry, introducing voice communication and creating unprecedented connections among individuals.

A pivotal advancement occurred in the late 1980s with the commercialization of the internet, which enabled rapid global information exchange and revolutionized communication paradigms. Presently, modern telecommunication systems have evolved into highly complex networks, delivering a diverse array of services across multiple sectors, including healthcare, finance, and entertainment. These systems form a critical component of global infrastructure, facilitating the digital age and ensuring seamless global connectivity.

The telecommunications industry is currently in the process of deploying 5th generation (5G) networks, a significant progression expected to greatly extend services and use cases, thereby amplifying its economic impact. 5G networks are projected to achieve uplink speeds of up to 10 Gbps and downlink speeds of up to 20 Gbps, utilizing millimeter-wave frequencies (above 6 GHz) and integrating advanced technologies such as massive Multiple Input Multiple Output (MIMO) and beamforming within the Radio Access Network (RAN).

In addition to these enhancements, 5G is adapting fully cloud-native and automated management systems, introducing new network architectures in both core and edge networks. These innovations will result in increased capacity, reduced latency, enhanced performance, and lower operational costs compared to previous generations, solidifying 5G's role as a cornerstone of future telecommunications infrastructures. These advancements position 5G to revolutionize various sectors, including high-speed media delivery, smart transportation, smart agriculture, smart healthcare, remote work environments, industrial automation (Industry 4.0), and the online gaming industry. Such applications are

expected to significantly contribute to human well-being and the global economy.

Current projections suggest that 70% of mobile traffic will be dedicated to video streaming [1], necessitating high-capacity and uninterrupted communication to ensure seamless user experiences. Additionally, the proliferation of Internet of Things (IoT) devices managing daily activities—such as smart homes, smart offices, and environmental monitoring in agriculture—significantly increases the volume of machine-to-machine communications. Moreover, emerging applications like autonomous driving and remote surgery demand ultra-low latency communications to meet stringent time-sensitive requirements. This diverse array of use cases imposes varying demands on network performance, necessitating customized communication solutions that address specific needs in terms of Quality of Experience (QoE), Quality of Service (QoS), and security levels.

Motivations

Transitioning from proprietary hardware and software to a cloud-native approach marks a significant technological advancement for the telecommunications industry, particularly with the advent of 5G. Traditionally, telecom networks have relied heavily on specialized hardware and proprietary software solutions, often resulting in higher costs and limited flexibility. By adopting cloud-native technologies, Communication Service Providers (CSPs) can leverage the inherent scalability, elasticity, and agility of cloud infrastructures for their services, while simultaneously reducing both Capital Expenditures (CapEx) and Operational Expenditures (OpEx).

However, moving to the cloud does not inherently guarantee lower costs if not managed properly. Many industries have migrated their IT operations to the cloud, yet experience higher operational costs due to a lack of cost optimization. Reports on cloud cost managing indicate that on average, 35% of allocated cloud resources in top three cloud service providers are wasted each year [2] as illustrated in Figure 1.1. In this context, wasted resources refer to those not fully utilized, leading to increased operational costs. Common reasons for this include a lack of understanding of workloads, insufficient observability into cloud deployments, and an inability to identify which resources (e.g., computing, memory, storage) are critical for day-to-day operations.

The lack of sufficient resources can degrade application performance, adversely affecting the QoS. To prevent this, many cloud users over-provision resources, further contributing to resource waste and cost increases. This highlights a major trade-off between cost

Figure 1.1 – Annual average cloud expenditure in top cloud service providers - AWS, Azure, and Google - 2019. (from [2])

and QoS in cloud-based applications. Optimizing operational costs is a complex challenge due to numerous independent variables, including different cloud models, pricing strategies, client-side application architectures, and various middleware components. Each case requires a customized cost-optimized solution, as there is no one-size-fits-all approach. Manual management of these factors is resource-intensive and often suboptimal, necessitating the need for automation.

Automation can significantly reduce cloud resource waste by dynamically adjusting resource allocation based on workload demands. This approach helps clients avoid over-provisioning, thereby lowering operational costs, while also preventing under-provisioning, which can degrade QoS and violate Service Level Agreements (SLAs). From the client's perspective, idle resources increase operational costs, while insufficient resources can lead to performance issues and SLA violations. From the cloud service provider's perspective, idle resources increase the energy consumption of underlying hardware. By leveraging automation, CSPs can achieve efficient, scalable, and cost-effective cloud operations, thereby fully capitalizing on the advantages of cloud-native technologies.

The evolution of network automation aims to achieve a fully automated network environment, significantly minimizing repetitive tasks that require human intervention. This reduces the risk of human errors, leading to enhanced operational efficiency and reliability. The path to full automation involves leveraging self-aware automation frameworks, characterized by self-healing and self-optimizing capabilities. These frameworks enable the network to autonomously detect, diagnose, and resolve issues, as well as optimize performance without external input. However, implementing such advanced automation in the

Figure 1.2 – Network automation maturity model (based on [3])

intricate domain of 5G mobile networks presents substantial challenges due to its complexity. Achieving this goal necessitates a phased approach, as outlined in the maturity model described in the report [3]. Each stage of the maturity model introduces incremental capabilities, laying the foundation for the subsequent stages and ensuring robust and scalable automation within the 5G ecosystem. Standardization bodies in the 5G domain are addressing this challenge by advocating for fully automated network management and actively developing and standardizing new concepts, architectures, and frameworks.

The realization of fully automated networks is largely driven by advancements in Artificial Intelligence (AI) and Machine Learning (ML) technologies. These technologies facilitate intelligent analytics and decision-making, optimizing network management operations. In the context of the cost-QoS trade-off in cloud environments, AI/ML techniques contribute to capacity planning by forecasting workloads and cost estimations. This enables dynamic allocation of cloud resources, ensuring optimal performance and cost efficiency.

Contributions

The objective of this thesis is to explore and enhance the efficiency of resource management for 5G Network Functions (NFs) deployed in cloud-native environments. A primary approach to addressing this issue involves the dynamic scaling of NFs in conjunction with network automation. This investigation uncovered several challenges, including analyzing and forecasting resource usage behavior of cloud-based applications, facilitating real-time decision-making, and assessing the impact of underlying hardware and middleware on the

scaling decisions. The contributions of this thesis are as follows:

Resource Usage Forecasting in CNFs

This contribution [4] addresses the cost-QoS trade-off in cloud-native 5G networks through dynamic autoscaling by emphasizing the necessity for proactive and precise scaling decisions. Accurately forecasting the resource usage of CNFs (Cloud-native network functions) is critical to prevent under- or overestimation, which can disrupt the balance between cost and QoS. Real-time resource forecasting offers valuable insights for autoscaling; however, predicting resource requirements in advance is challenging due to the complex relationship between incoming workloads and actual CNF resource usage.

To tackle this issue, a novel resource usage forecasting method specifically designed for CNFs in Kubernetes environments is proposed. This approach integrates the intricate relationship between workload demands and resource consumption. A multivariate time series forecasting approach, leveraging deep learning models, is introduced to capture these complexities and accurately predict future resource usage of CNFs. The method provides reliable forecasts of resource consumption patterns over extended time horizons, thereby enabling real-time insights that support proactive autoscaling decisions.

Dynamic autoscaling decision-making

This contribution [5] addresses the challenges of proactive horizontal autoscaling decision-making within Kubernetes environments. Specifically, it tackles the complexities inherent in making scaling decisions, even when resource usage is accurately forecasted. Our research introduces a novel approach that enhances the cost-QoS trade-off by preventing under-provisioning and over-provisioning through intelligent differentiation of scaling events (scaling up, scaling down, or no scaling) and the strategic timing of these actions.

The core of this contribution lies in the development of an advanced autoscaling mechanism that leverages both static and dynamic thresholds to distinguish between different scaling scenarios. This method incorporates experimental techniques to fine-tune the timing of scaling decisions, ensuring that actions are taken at the most opportune moments. By addressing Kubernetes' limitations and mitigating decision oscillation effects—common issues in threshold-based scaling systems—the proposed solution enhances the stability and effectiveness of the autoscaling process. The proposed methodology enables real-time scaling decisions based on predicted resource usage, thereby minimizing operational costs and reducing QoS degradations during scaling events. This innovative

approach significantly improves the balance between cost efficiency and service quality, providing a more robust and responsive autoscaling system for Kubernetes environments.

CPU throttling aware autoscaling

We identify Central Processing Unit (CPU) throttling as a critical challenge in cloud computing, particularly in scenarios where CNFs operate near their allocated CPU resource limits, leading to QoS degradation. This challenge is exacerbated by the need to balance the trade-off between cost and QoS, as autoscalers are designed to maximize resource utilization while minimizing operational costs. When CNFs approach their CPU allocated capacity, CPU throttling mechanism is triggered, resulting in increased service response times and adversely impacting performance-sensitive applications. Accurately predicting the onset of CPU throttling and executing timely scaling actions is crucial to mitigate its impact. However, this task is complicated by the intricate inter-dependencies between Operating System (OS) mechanisms, Kubernetes orchestration, workload variability, and CPU usage patterns. These complexities make it challenging to identify the optimal timing for autoscaling decisions, which are vital for achieving a balance between cost efficiency and QoS.

This contribution [6] introduced a novel trigger mechanism that leverages deep learning to predict CPU throttling events and facilitate real-time scaling decisions. To the best of our knowledge, this work is among the first to investigate CPU throttling prevention in the context of Kubernetes using deep learning techniques. By proactively addressing the effects of CPU throttling on service response times, this approach aims to minimize response time degradation during scaling events, thereby significantly enhancing the performance of CNFs, particularly within cloud-native 5G network environments.

Organization of the manuscript

Chapter 1 presents a general introduction to the thesis context and outlines its direction.

Chapter 2 provides a comprehensive background on different domains addressing the cost-QoS trade-off in cloud-native 5G networks. This chapter delves into the various components, standards, and concepts crucial to addressing this trade-off, including the functional architecture of 5G networks, principles of network automation, cloud computing paradigms, and the role of Kubernetes in managing containers within cloud environments.

Chapter 3 conducts an in-depth analysis of state-of-the-art autoscaling solutions within cloud environments. The chapter begins by identifying the key considerations in developing effective autoscaling mechanisms. It then categorizes and evaluates existing autoscaling solutions, examining their respective advantages, disadvantages, and their compatibility with cloud-native 5G architectures.

Chapter 4 presents the first contribution of this thesis: the development of a novel proactive two-stage autoscaling solution. This chapter is dedicated to the initial phase, which centers on resource usage forecasting for CNFs. A deep learning-based forecasting method is introduced, specifically focused on cloud-native 5G environments. The chapter concludes with a comprehensive evaluation of the proposed forecasting model, assessing its accuracy and effectiveness in predicting resource demands where results demonstrate that the proposed forecasting approach accurately predicts resource usage over extended time horizons.

Chapter 5 presents the second contribution of this thesis, which addresses the second stage of the proposed autoscaling solution. This chapter is centered on the design and implementation of a dynamic scaling mechanism. It provides a detailed account of how the forecasting stage is integrated with dynamic scaling to create a cohesive autoscaling solution. It also includes detailed experimentation of the proposed solution on a real-world cloud platform. The chapter concludes with a evaluation of the complete solution, and the results indicate that our proposed auto-scaling solution achieves a superior balance between cost and QoS compared to Kubernetes HPA and other state-of-the-art proactive solution.

Chapter 6 provides the third contribution, addressing a critical issue that affects the cost-QoS trade-off in the proposed autoscaling solution: CPU throttling during autoscaling operations. This chapter introduces a novel autoscaling strategy specifically designed to mitigate the adverse effects of CPU throttling. The efficacy of this strategy is evaluated, with a focus on its impact on maintaining the balance between cost and QoS. The results demonstrate that our proposed solution effectively minimizes QoS degradation caused by CPU throttling during scaling, achieving a better balance between cost and QoS.

Chapter 7 concludes the thesis by summarizing the key findings and contributions. It provides a forward-looking perspective, suggesting potential areas for future exploration in balancing cost-QoS trade-off, energy-QoS trade-off, security aspects etc.

Chapter 2

Background

The rapid evolution of 5G networks, particularly in cloud-native environments, has introduced a new set of challenges and opportunities in managing the balance between cost and QoS. As these networks become increasingly integral to the digital infrastructure, the ability to optimize resource allocation dynamically is critical. To address this cost-QoS trade-off effectively, it is essential to gain a comprehensive understanding of the various components influencing the balance between cost and QoS, as each plays a significant role in determining the overall efficiency and performance of the network.

The first step in tackling this challenge is to focus on the application itself: the cloud-native 5G network. This involves delving into the intricate details of the 5G architecture, understanding its communication protocols, and exploring the inner workings that define its capabilities and limitations. A deep comprehension of these aspects is crucial to devising customized solutions that can optimize resource allocation, thereby improving both cost efficiency and QoS.

Automation emerges as a key enabler in managing this balance. The telecommunications industry has long leveraged automation to streamline operations and enhanced performance, and this trend is now extending into cloud-native 5G networks. Various standardizing organizations have been working on establishing frameworks and guidelines for automation in this context. Therefore, it is important to understand the current industry standards and how they can be adapted and applied to cloud-native 5G networks. Aligning with these standards not only ensures greater interoperability but also positions these networks for future scalability and innovation.

Another critical factor to be considered in the cost-QoS trade-off is the type of cloud environment used to deploy the 5G network. Different cloud models, whether public, private, or hybrid, come with their own sets of pricing structures and operational characteristics. These choices can significantly impact the cost-effectiveness and service quality of the network. Thus, understanding the implications of these cloud environments is essential for making informed decisions that optimize resource allocation.

Lastly, the software tools used to manage cloud-native 5G networks are crucial in this dynamic landscape. Kubernetes, as a leading container management platform, offers valuable features for optimizing resource allocation, though it also presents certain limitations and challenges. A comprehensive assessment of these tools is essential to identify the most effective strategies for balancing cost and QoS. By capitalizing on the strengths of these tools and addressing their shortcomings, it is possible to devise a resource allocation strategy that improves both cost efficiency and QoS in cloud-native 5G networks. This chapter, will explore these critical components in detail, providing the foundational knowledge required to navigate the complex landscape of cost and QoS management in cloud-native 5G networks.

2.1 5G network & main challenges

In response to the exponential growth in mobile network connections, a significant advancement in 5G technology is the adoption of a cloud-native approach. This paradigm shift facilitates networks to exhibit high scalability, resilience, and adaptability to fluctuating demands. In the 5G architecture, both Radio Access Network (RAN) and Core Network (CN) functions have embraced this cloud-native methodology, extending its benefits to edge networks as well [7] [8] [9].

2.1.1 Service-based architecture

In accordance with the 3GPP (3rd Generation Partnership Project) specifications [10], the 5G architecture not only transitions towards a cloud-native approach but also adopts a Service-Based Architecture (SBA) [11]. This transition facilitates the decomposition of monolithic software into manageable, independent microservices. The modular nature of this microservice architecture enables 5G NFs to be updated, modified, scaled, migrated and dynamically placed across the network with ease.

Furthermore, these loosely coupled, NFs with isolated resource contribute to cost reduction as only the necessary services need to be altered based on demand. The lightweight, containerized nature of these network functions, coupled with their hardware independence, enables fine-tuned resource allocation and facilitates rapid recovery [12]. In the event of faults, services can be swiftly redirected, enhancing network resilience. This also enables integration of Continuous Integration/Continuous Deployment (CI/CD) processes

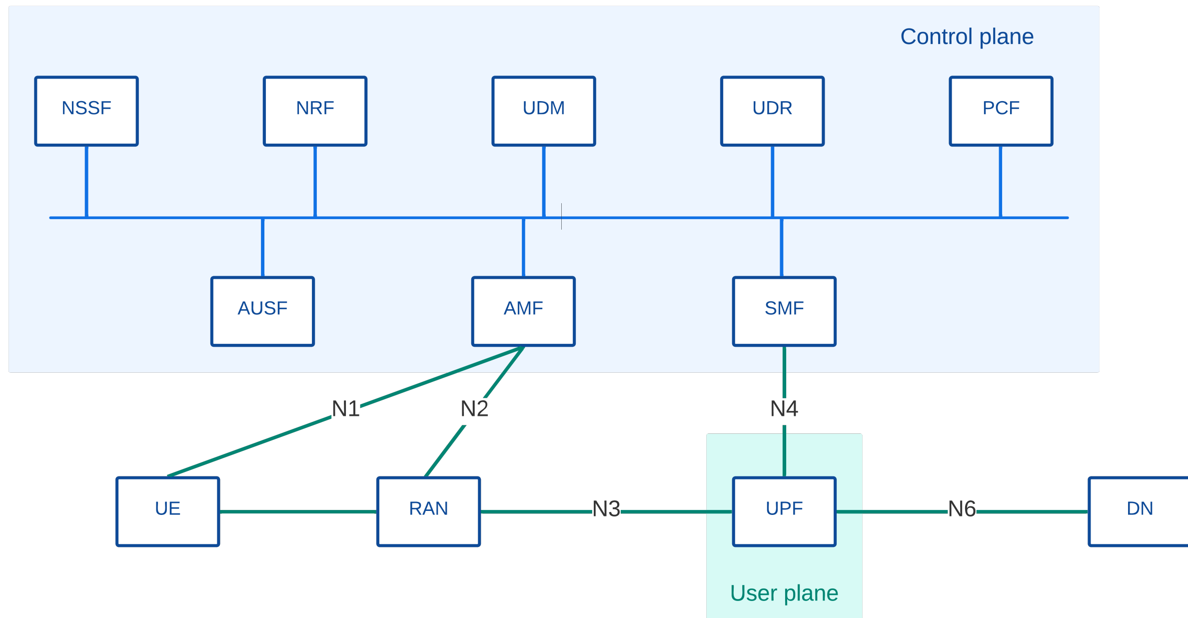


Figure 2.1 – 5G Architecture. (based on [10])

[13], thereby improving the lifecycle management of network functions. This shift towards a cloud-native, SBA simplifies the automation of network function and service lifecycle management, marking a significant advancement over previous generations.

2.1.2 5G Core network

In the architecture of 5G, the 5G Core Network is a crucial component that plays a significant role in managing communication services on the network. Beyond the adoption of a SBA, a significant enhancement is the separation of Control Plane (CP) NFs and User Plane (UP) NFs [14]. This separation allows the 5G CN to be more flexible and efficient, enabling it to support a wide range of use cases. For instance, the UP can be deployed closer to the users to improve latency, while the CP is deployed separately. This architecture also allows for the provision of customized UPs for different services, all while utilizing the same CP. These are just a few examples of the capabilities of the 5G CN. The 5G architecture, as illustrated in the Figure 2.1, shows that User Equipment (UEs) connected to the Next Generation Radio Access Network (NG-RAN) can access the CN for its functionalities both prior to and during connection with the data network. Below are some of the main network functions and their corresponding functionalities:

Access and Mobility Management Function (AMF)

The Access and Mobility Management Function is responsible for handling registration, connection, and mobility management. As the CP's access point from the NG-RAN, it also oversees authentication and authorization processes. The UE connects to the AMF via the N1 signaling interface, while the NG-RAN connects to the AMF via the N2 interface. The N1 interface uses the Non-Access Stratum (NAS) signaling messages to manage initial registration, mobility management, and other functions. The N2 interface uses the Next Generation Application Protocol (NGAP) over Stream Control Transmission Protocol (SCTP) for CP communications. Additionally, the AMF manages the termination process for both the N1 and N2 interfaces.

Session Management Function (SMF)

The Session Management Function is responsible for managing session-related processes, including session establishment, modification, and release. It interfaces with the User Plane Function (UPF) via the N4 interface, overseeing various UPF operations such as policy enforcement, UP traffic management, and UP selection when multiple UPFs are available. The N4 interface uses the Packet Forwarding Control Protocol (PFCP) to facilitate this communication.

Network Repository Function (NRF)

The Network Repository Function is responsible for service discovery, registration, and management within the 5G CP. It maintains profiles and instances data of NFs along with related information.

Unified Data Management (UDM)

The Unified Data Management function is responsible for generating user authentication keys, managing subscription data, handling user identification, and exposing subscriber data to authorized external entities.

Unified Data Repository (UDR)

The Unified Data Repository is responsible for delivering centralized data repository services to other NFs through UDM. It maintains comprehensive storage of subscriber

data, policy data, and other network-related information.

Authentication Server Function (AUSF)

The Authentication Server Function oversees the UE authentication process by verifying the UE's authentication request using credentials and data retrieved from the UDM. It ensures that the authentication key provided by the UE matches the subscriber data maintained by the UDM, thereby granting access to the network.

Policy Control Function (PCF)

The Policy Control Function provides policies based on user profiles, network status, and service types, utilizing subscription information to manage the CP.

User Plane Function (UPF)

The User Plane Function is responsible for packet traffic management, QoS management, packet inspection, and policy enforcement. The UPF communicates directly with the SMF via the N4 interface. Additionally, it interfaces with the NG-RAN through the N3 interface, utilizing the General Packet Radio Service Tunneling Protocol-User Plane (GTP-U) protocol to encapsulate and transport user data. Subsequently, the UPF connects to external data networks via the N6 interface, employing Internet Protocol (IP) protocols for communication.

All NFs in the CP utilize Representational State Transfer (REST) Application Programming Interface (API) over Hypertext Transfer Protocol (HTTP) for intercommunication. In addition to the aforementioned NFs, the 3GPP specification defines several other NFs, each serving specific use cases. For instance, the Network Slice Selection Function (NSSF) is dedicated to managing network slicing within the 5G CN, overseeing all slicing-related processes. A significant addition to the 5G CN is the Network Data Analytics Function (NWDAF), which facilitates network automation. NWDAF's primary functions include data collection, insight generation, and performance optimization. It also lays the foundation for integrating data-driven AI/ML technologies into 5G CN performance management. While these NFs are crucial for service management in 5G, a separate framework is required for lifecycle and resource management in a cloud environment. The European

Telecommunications Standards Institute (ETSI) has standardized the Management and Orchestration (MANO) architecture for this purpose, known as NFV MANO [15]. To ensure efficient management and orchestration of cloud-native 5G networks, network automation is essential. This automation necessitates its own standardization to facilitate coordinated large-scale network management and enhance interoperability.

2.2 Network automation

The advent of the 5G network heralds a new era of high-performance connectivity, supporting innovative business models such as Network as a Service (NaaS), and enabling a wide array of use cases [16]. These advancements are not only transformative, but also introduce significant complexity to network management.

Network slicing, a fundamental concept in 5G networks [17] [18], involves the creation of isolated logical networks operating on a shared physical infrastructure as shown in the Figure 2.2. These logical networks are tailored to support diverse vertical use cases and business models [19]. This paradigm introduces a comprehensive set of challenges, including but not limited to service lifecycle management, NF lifecycle management within slices and service isolation, resource isolation, security between slices [20] [21] [22]. Furthermore, network slicing is expected to extend across multiple domains, accommodate multiple tenants, and support various services, as illustrated in the Figure 2.3.

This expansion underscores the complexity inherent in managing such a system, which involves coordinating and integrating multiple isolated network slices, each with unique requirements and performance criteria. Given this complexity, traditional manual network management methods are becoming increasingly impractical and inefficient, and they are prone to human error. Therefore, the need for more sophisticated, automated network management solutions is evident.

With the development of Software-Defined Networking (SDN), which decouples the network control plane from the data plane, and Network Functions Virtualization (NFV), which virtualizes network functions by separating software from dedicated hardware components, key advancements have been made in enabling network automation [23]. The rapid improvement in general-purpose hardware performance and the cost-efficiency of cloud operations, coupled with advancements in AI/ML technologies, further facilitate the implementation of automated network management within advanced networking paradigms [25].

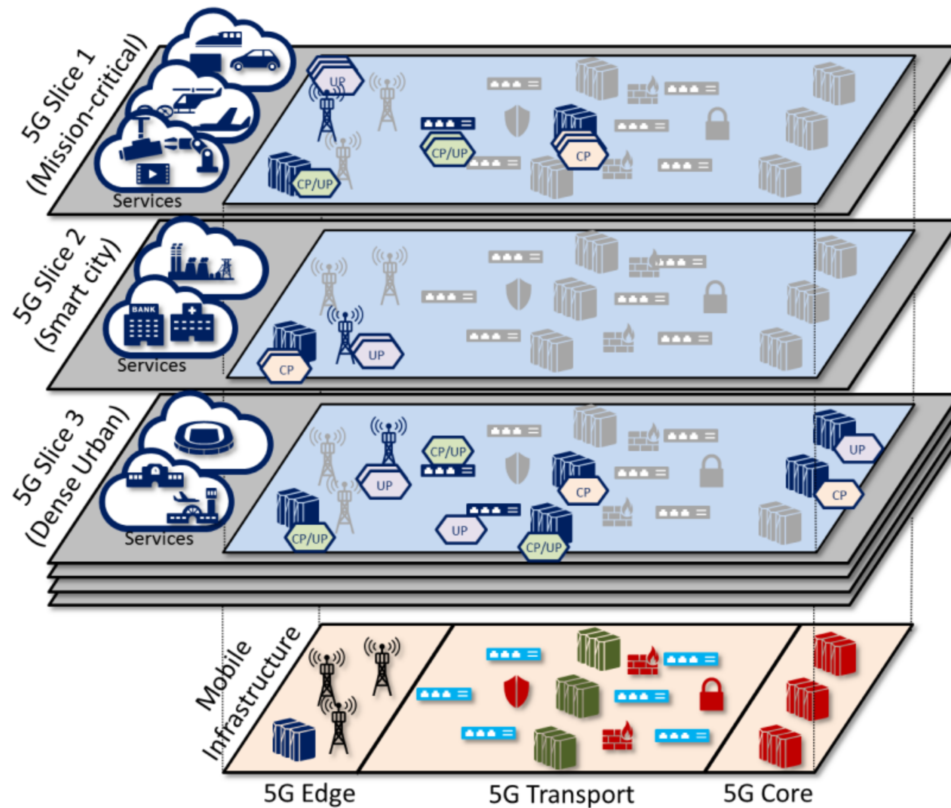


Figure 2.2 – Network slicing with 5G network. (from [23]) (Copyright © 2017 IEEE)

In the past, various approaches have been proposed over the years to enhance and automate network management. Among these, the Observe, Orient, Decide, Act (OODA) [26] loop and the Monitor, Analyze, Plan, Execute, Knowledge (MAPE-K) [27] model stand out. Both models encapsulate the principles of closed-loop automation, which are pivotal for modern network management strategies. These methodologies aim to streamline the management process by automating routine tasks, thus reducing the potential for errors and improving overall efficiency.

A significant milestone in the journey toward standardized network automation is the Self-Organizing Network (SON) concept introduced by the 3GPP [28]. SON encompasses a suite of technologies and techniques designed to automate network configuration, optimization, and troubleshooting, thereby alleviating the burden on human operators and enhancing network performance and reliability [29] [30]. However, this early standardization has significant progress to make before achieving fully autonomous networks and supporting large-scale, cross-domain modern networking as expected in 5G and beyond [31].

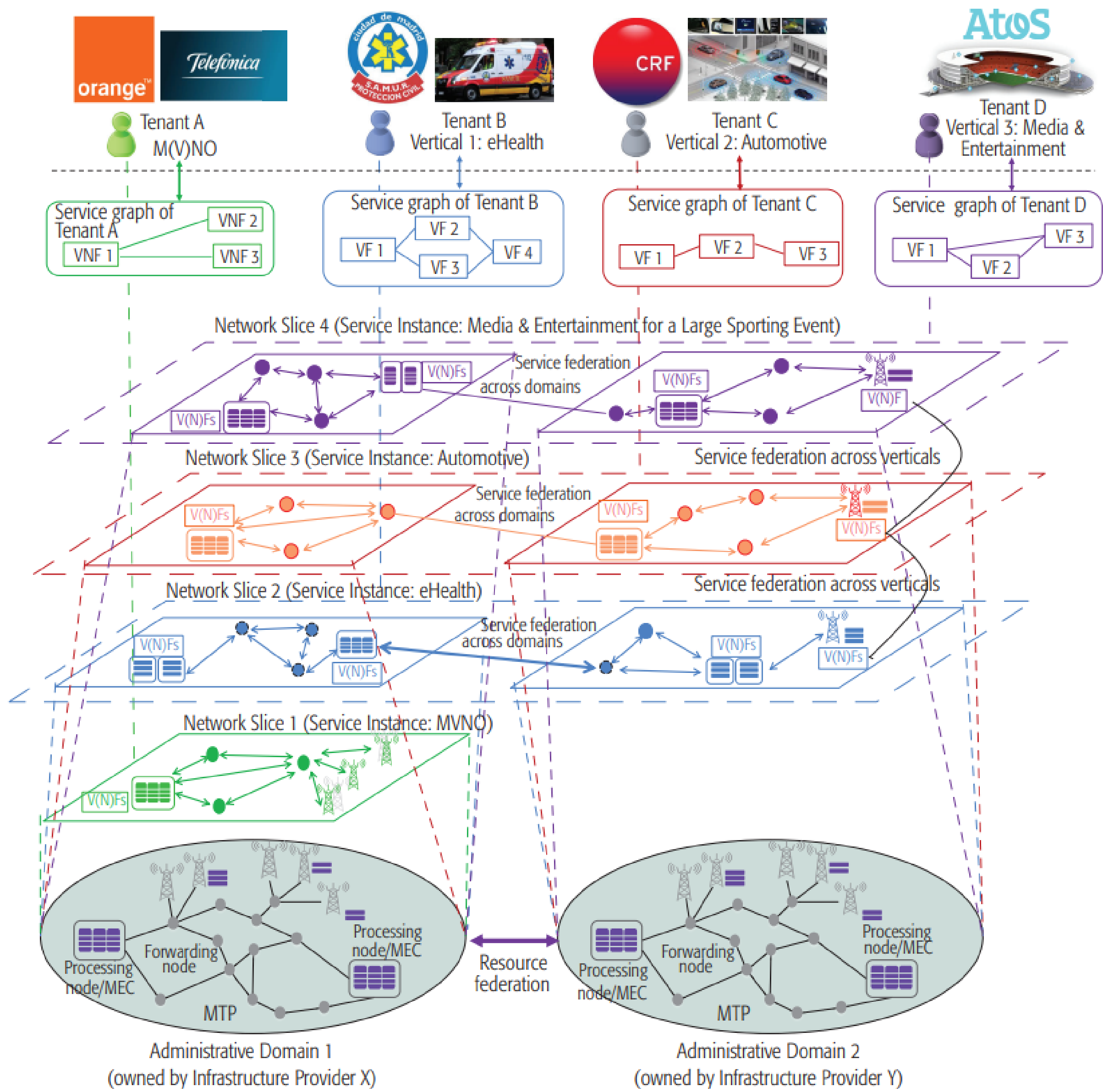


Figure 2.3 – Multi domain, multi tenant network slicing. (from [24] - modified) (Copyright © 2018 IEEE)

2.2.1 ETSI ZSM Framework

Recently, the Zero Touch Network and Service Management (ZSM) working group of the ETSI has adopted the concept of a closed-loop framework aimed at standardizing telecom network automation processes [32] [33]. This framework addresses various shortcomings found in previous standardization attempts and incorporates the latest technological advancements, accommodating modern 5G and beyond network architectures. It offers a flexible, modular, and data-driven architecture, providing End-to-End (E2E) automation through a unified framework and aiming to achieve fully automated network management [34] [35].

The ZSM project has garnered significant attention from industry stakeholders, including network operators, vendors, cloud providers, academic partners, and open-source communities. Notably, the Open Network Automation Platform (ONAP) [36], an open-source network orchestration and management platform, has already aligned its automation processes with the ETSI ZSM framework [37].

The proposed framework, illustrated in the Figure 2.4, comprises several management blocks designed to achieve E2E automation. Key architectural principles embedded within this framework include:

— Modularity: to prevent monolithic structures and tight coupling between entities.
— Extensibility: to facilitate the seamless integration of new services without compatibility issues.
— Scalability: to dynamically adjust management services according to demand.
— Model-Driven Design: to ensure portability, reusability, and vendor-neutral management.
— Closed-Loop Automation: to enable feedback-driven automation.
— Stateless Management: to decouple management functions from data storage services.

Management domains are entities that can be technical or organizational, partitioned based on deployment, functional, operational, or governance constraints. Each domain is responsible for domain-level lifecycle management processes by integrating closed loops within the domain.

The E2E management domain oversees all domains involved in E2E services, coordinating seamless management processes. It is responsible for E2E service-level lifecycle management and supporting E2E service-level closed-loop automation processes.

An entity used in both the management domain and the E2E management domain

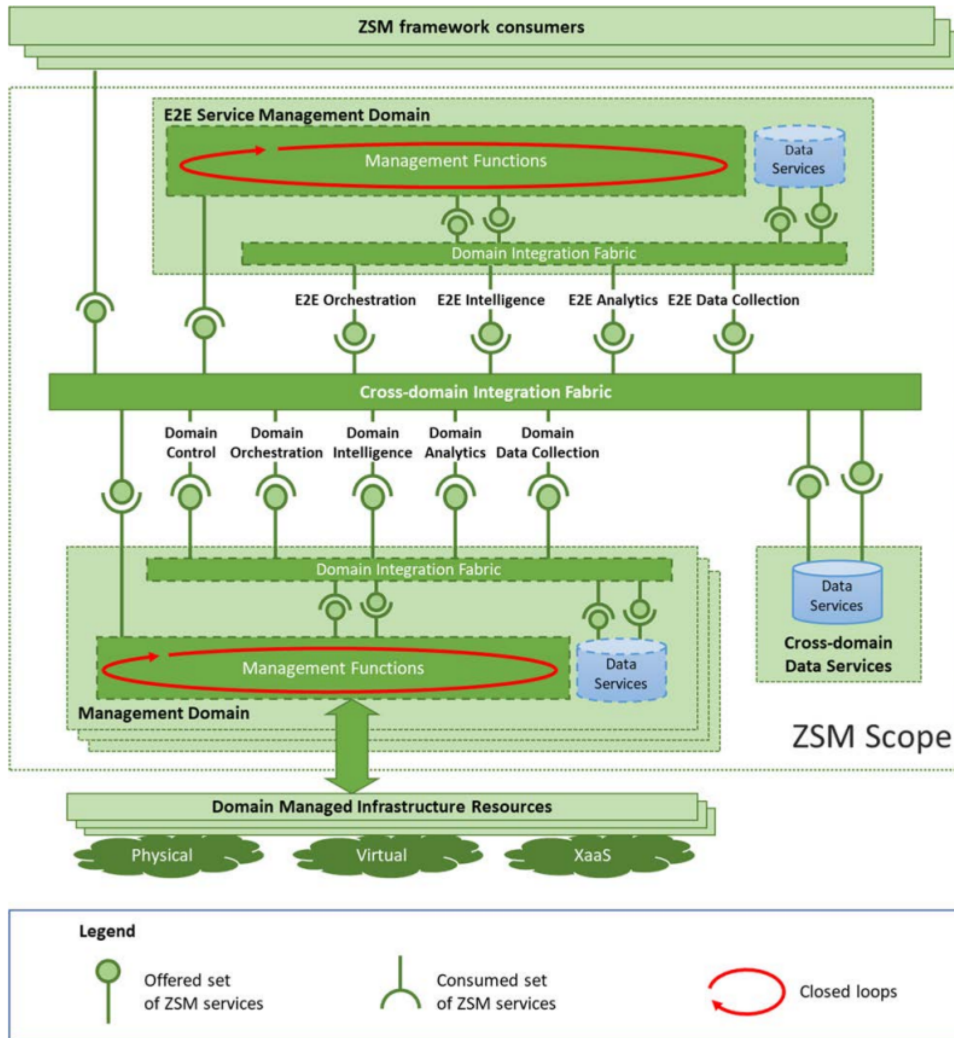


Figure 2.4 – ETSI ZSM reference architecture. (from [33])

seperatly within the ZSM framework is data services. Each management domain and E2E management domain maintains a dedicated data service to store domain-level data, which is kept separate from management functions. The stored data is accessible to other authorized domains and/or E2E management domains. Besides the individual domain-level data services, the ZSM framework also includes a cross-domain data service for centralized storage of all management and E2E domain-level data.

Furthermore, each management domain and E2E management domain includes an integration fabric, serving as the communication interface outside the domain. This fabric supports service registration, access, routing invocation, and service exposure. Additionally, the ZSM framework maintains a cross-domain integration fabric, interconnecting all

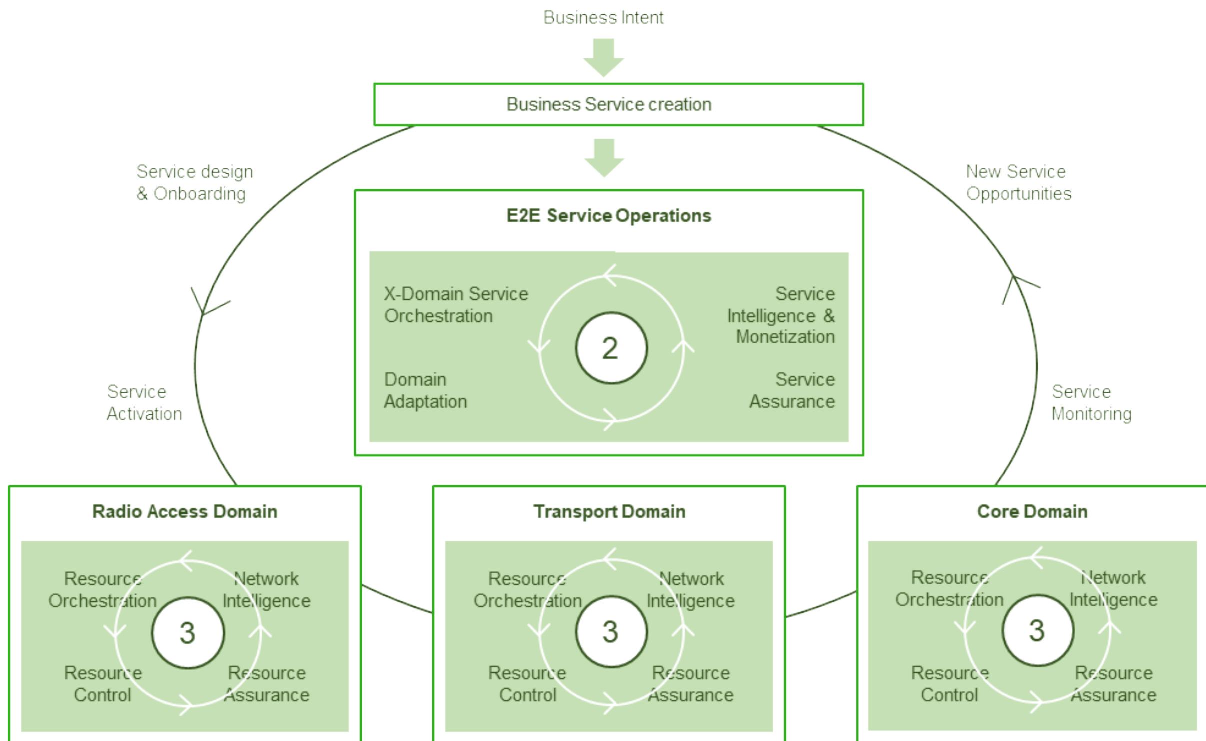


Figure 2.5 – E2E Communication service automation with ETSI ZSM framework. (from [33] - modified)

components to enable seamless communication across domains.

To further elucidate the framework, an illustrative case depicted in the Figure 2.5, considers an E2E communication system comprising three management domains dedicated to network segments (RAN, transport and CN). Each of these networks constitutes a management domain, automated through closed-loop mechanisms. Each management domain is tasked with managing various functions including, but not limited to, resource orchestration, control assurance, and the delivery of intelligent services. These functions are automated within their respective closed loops to ensure efficient management and operation. In addition to these management domains, there exists an overarching E2E management domain designed to meet customer requirements and oversee all aspects of E2E service-level management processes. This domain encompasses service orchestration, domain adaptation, service assurance, and the provision of comprehensive E2E service intelligence. These processes are also automated using closed-loop mechanisms, ensuring seamless integration and intelligent management across the entire E2E communication system.

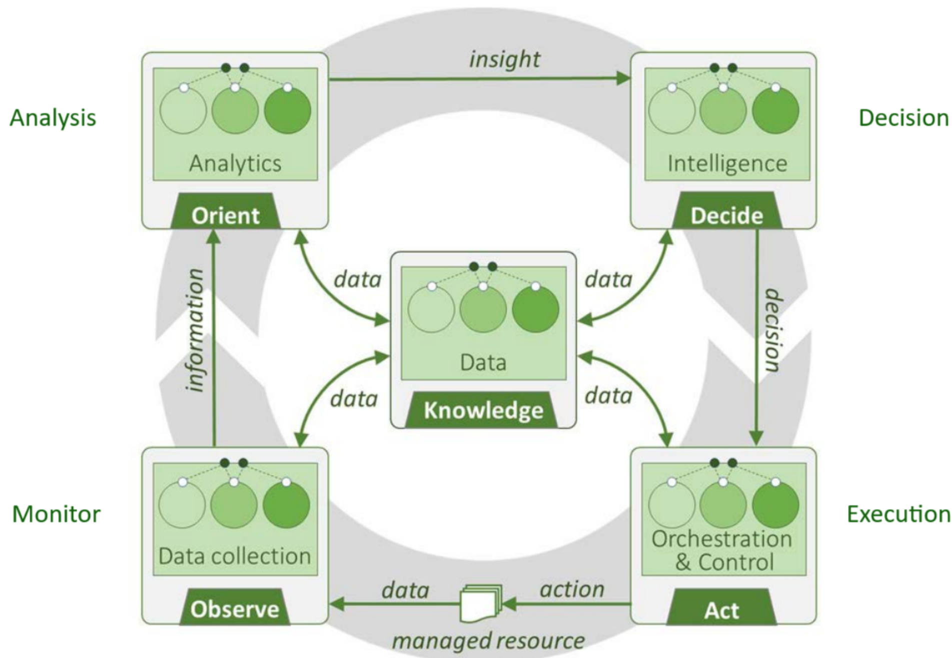


Figure 2.6 – ETSI Closed loop stages within ZSM framework compared to OODA model. (from [33] - modified)

2.2.2 Closed loop automation

As previously mentioned, each component within the proposed framework can be automated using closed-loop automation. A closed-loop mechanism functions autonomously, making precise decisions based on continuous monitoring and analysis of the target entity. This mechanism includes a feedback loop that provides information on the subsequent impact of decisions on the target entity and suggests follow-up actions necessary to maintain the target entity at desired operational levels [38] [39]. To achieve this, ETSI ZSM employs the following closed-loop stages, illustrated Figure 2.6.

Monitor

This stage is responsible for data collection and data preprocessing from the target entity. In complex systems, data can come in various formats, such as time series, categorical data, etc. and from different sources, where data can be real-time data streams, historical data from databases, etc. Additionally, the collected data will be preprocessed before being forwarded to the next stage for analysis.

Analysis

In this stage, the preprocessed data is used to generate intelligent insights. The objective is to explain the reasons behind data behaviors. These insights are critical for precise decision-making in the subsequent stage. This mechanism continuously refines its analysis of the target entity based on the data.

Decision

Insights from the analysis stage are used to develop workflows that govern the behavior of the target entity, addressing issues identified from the insights. This stage makes corrective adjustments to maintain the desired system levels, which can be either reactive or proactive.

Execution

This final stage is responsible for implementing the decisions made in the previous stage. It interprets the corrective decisions and determines the necessary actions to actualize those decisions.
The Knowledge state, although not an active stage in the closed loop, stores all data flowing between stages and facilitates data sharing with other closed loops. Additionally, authorized external entities can retrieve information or intervene at each stage.

Some tasks, which can be divided into several elements, require more than a single closed loop for automation. In such cases, multiple closed loops can coexist, sometimes hierarchically or in a nested form. A separate closed loop management process is necessary to manage closed loop coordination and prevent conflicts [40]. Furthermore, optimizing the network automation process to achieve high performance and cost-efficient management in cloud-native 5G networks requires a thorough understanding of the deployment environment, including cloud models, processing models, and other relevant factors.

2.3 Cloud computing

Cloud computing is a paradigm that delivers on-demand computing resources, data storage and networking services, etc., over the internet. This concept, now a well-established industry, is increasingly adopted across various sectors due to its numerous advantages,

including cost reduction, scalability, reliability, lack of maintenance requirements, accessibility from any location and ease of management [41] [42]. The advent of virtualization and containerization of network functions has made cloud computing particularly appealing to telecommunications operators. Software-based network functions can now deliver high-performance services without the dependency on dedicated hardware. Presently, a diverse array of cloud models, service models, and pricing models are available, providing customers with the flexibility to select options that align best with their business requirements [43].

2.3.1 Cloud models

Public clouds

Public clouds provide computing power, storage options, networking, and other services over the internet, with infrastructure owned and managed by third-party operators such as Google Cloud Platform (GCP) [44], Microsoft Azure [45], and Amazon Web Services (AWS) [46]. In the context of cloud services, clients are absolved from the responsibility of infrastructure and maintenance costs associated with the deployment of their applications. Instead, they incur a fee contingent upon the specific hardware and software requirements of their applications. This fee is subject to variation, influenced by several factors including the pricing model, the use case, and the cloud provider. These elements will be elaborated upon in subsequent sections. A significant advantage of public clouds is their extensive infrastructure and the ubiquity of their data center facilities across diverse geographical locations, which ensures high scalability and reliability. Nevertheless, despite the robust security options available for customer applications within public clouds, the shared infrastructure and significant exposure to the internet increase the potential for cyber threats [47].

Private clouds

Private clouds deliver cloud services via the internet, similar to their public counterparts. However, the infrastructure resides on-premises within the organization and is managed by the same entity. This arrangement allows for a high degree of customization for applications hosted in the cloud, as all elements are owned and controlled by the same organization. An additional advantage of private clouds is the enhanced security they offer, stemming from their limited exposure to the public. However, the responsibility of

managing the infrastructure can result in substantial initial costs for hardware and software, as well as ongoing maintenance expenses. These maintenance costs can manifest in various forms, including infrastructure repair, labor costs, and energy consumption associated with the infrastructure. Contrary to public clouds, the infrastructure of private clouds is typically smaller and less geographically dispersed, which may limit scalability.

Hybrid clouds

Hybrid cloud architectures represent an amalgamation of public and private cloud infrastructures. In this model, the client retains the discretion to determine the distribution of deployment between the public cloud and on-premises infrastructure. This methodology offers superior flexibility compared to other models, as it allows the client to leverage the extensive scalability and reliability inherent to public clouds, while simultaneously enhancing customization and security through on-premises deployment.

2.3.2 Cloud service models

In the domain of public cloud computing, providers must accommodate a diverse array of client service requirements for their applications. To address this, providers offer a selection of cloud service models, enabling clients to opt for the model that aligns optimally with their needs. These service models are not mutually exclusive; clients have the flexibility to utilize combinations of these models to optimize their solutions [48].The Figure 2.7 illustrates the comparison of component management and ownership across various service models.

Infrastructure as a Service (IaaS)

In Infrastructure as a Service (IaaS) service model, providers supply only hardware resources, including compute, storage, networking, etc. and virtualization services. Clients are responsible for deploying and operating their applications by utilizing their own OS, middleware, and managing runtime operations.

Container as a Service (CaaS)

In the Container as a Service (CaaS) model, providers offer hardware resources, virtualization services, and operating systems needed to execute containerized applications.

Graphic removed to respect copyright

Figure 2.7 – Comparison between different cloud service models. (based on [48])

However, clients must supply additional software components for deploying applications and assume responsibility for managing runtime operations.

Platform as a Service (PaaS)

Platform as a Service (PaaS) is a cloud service model that provides hardware resources, virtualization services, OS, and middleware components necessary for running applications. In this model, clients are responsible solely for managing their application's runtime operations.

Software as a Service (SaaS)

In the Software as a Service (SaaS) model, providers deliver a complete application service that includes hardware resources and all necessary software components ready for immediate use. The cloud provider manages all aspects of hardware and software management, as well as runtime operations.

2.3.3 Cloud pricing

In the realm of public cloud pricing models, several factors influence pricing, including the chosen cloud model, client specifications, market dynamics, and the cloud provider itself. Nevertheless, there exist several prominent cloud pricing models widely adopted by the industry [49] [50] [51].

Pay-as-you-go

In this model, clients are billed based on the actual usage of cloud resources or services. Billing can be granular, ranging from hourly to per-second usage. This pricing structure is optimal for highly variable demand scenarios where dynamic scaling of cloud-based applications is required. Typically, these pricing models do not necessitate upfront payments.

Reserved instances

In this pricing model, clients reserve cloud resources or services for a predefined time period and pay a predetermined cost regardless of actual usage. This model is typically suited for environments with stable demand, as scaling beyond the reserved capacity is prohibited. Clients are generally required to make an upfront payment and commit to a long-term agreement.

Spot instances

Cloud providers offer unallocated resources or services to clients with the understanding that there is a risk of interruption. These resources are typically suited for non-critical applications where interruptions are tolerable. This method is generally more cost-effective and useful for handling sudden bursts of demand that exceed reserved instances.

Subscription based

In this pricing structure, clients are required to pay monthly or annually for access to cloud services. This model is predominantly used in SaaS cloud models. Typically, there are no upfront fees or long-term commitments associated with this model.

2.3.4 Cost model

5G networks can be deployed in any cloud model, and when deployed in the public cloud, they can be implemented under most cloud service models or pricing model. Given the containerized nature of NFs, one effective configuration is deploying 5G networks in the public cloud using a CaaS model. Considering the large volume of connections and highly variable workloads associated with 5G networks, a pay-as-you-go pricing model can be an optimal choice for cost efficiency. Typically, major cloud providers offering CaaS, such as Google Kubernetes Engine (GKE) [52] and Amazon Elastic Kubernetes Service (EKS) [53], set prices for client application containers based on their resource configurations, primarily CPU and memory. Later, under the pay-as-you-go pricing model, these providers charge clients based on the actual usage period of the application. Hence,

$$Cost \propto Container\ operational\ time \tag{2.1}$$

Consider the hourly rate per container for a specified resource configuration, consisting of CPU and memory per service, denoted as β_j where j represents the service (or network function). Utilizing a pay-as-you-go pricing model, the cost is contingent upon the container usage. If dynamic horizontal scaling is enabled, which involves adjusting the replication of containers to manage the workload, for a given time duration T, the cost can be calculated as,

$$Cost = \beta_j \times \sum_{i=0}^{N} T_{(i,j)} \tag{2.2}$$

where N is the maximum number of replicas used within T, $T_{(i,j)}$ is the operational time for each replica i for a given service j and $T_{(i,j)} \leq T$.

When deploying cloud-native 5G networks, it is essential to understand the capabilities and limitations of the various software tools available to effectively manage the cost-QoS balance in the cloud.

2.4 Kubernetes for CNF management

Kubernetes [54] is an advanced container orchestration platform that has gained widespread adoption among cloud service providers. Originally developed by Google in the early 2000s as a containerized application management tool within its internal data

centers, known as Borg [55], it was later refined into a more robust and scalable system named Omega [56], addressing the constraints of its previous version. In 2014, Google made the strategic decision to open-source this technology under the name Kubernetes. Subsequently, in 2015, the governance of the project was transferred to the Cloud Native Computing Foundation (CNCF) [57], a part of the Linux Foundation [58]. Since its inception, Kubernetes has undergone significant enhancements through contributions from the open-source community, establishing itself as a pivotal entity in the cloud-native ecosystem.

2.4.1 System architecture

Kubernetes is designed for deployment on bare metal servers or Virtual Machines (VMs) on top of IaaS platforms such as OpenStack [59]. Its architecture as shown in the Figure 2.8, is divided into two main components: the Kubernetes control plane (also known as the master node) and the worker nodes. In this context, a node refers to either a physical server or a VM, depending on the deployment environment. A typical Kubernetes deployment consists of at least one master node and one or more worker nodes, collectively referred to as a cluster. The master node hosts all the cluster management components, while the worker nodes host all the workload container deployments.

In Kubernetes, containerized applications are deployed within entities called "Pods." A Pod is the smallest deployable unit in Kubernetes and can contain one or multiple containers. The resources allocated to a Pod are shared among its containers. In the event of scaling, these Pods can be replicated, a process managed by ReplicaSets, which will be explained later in this chapter. To access containerized applications within Pods, Kubernetes provides a stable endpoint known as a "Service." This Service ensures reliable communication in the cluster, allowing clients to access applications without needing to know the IP addresses of individual Pods. When a Pod is replicated during scaling, all replicas belong to the same Service. Additionally, Services can be configured for internal cluster communication or to connect with external systems.

The control plane in Kubernetes architecture comprises four primary components: the API server, etcd, scheduler, and controller manager.

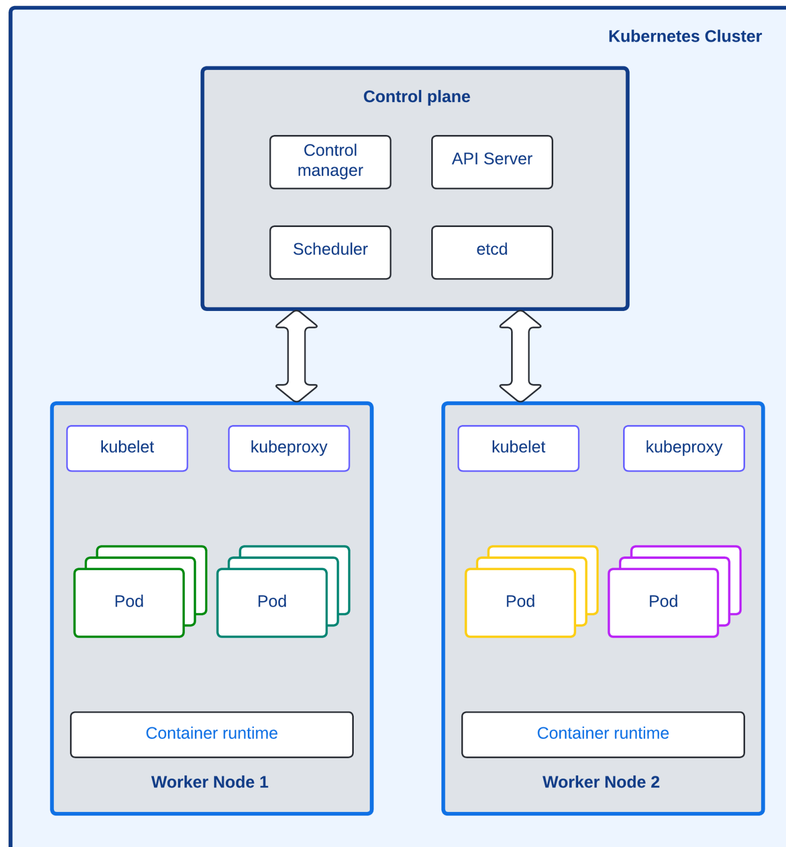


Figure 2.8 – Kubernetes architecture. (based on [60])

API Server

The API server functions as the front end of the control plane, providing an interface for clients and other Kubernetes components to interact with the cluster. It is responsible for managing the cluster state, handling authentication processes, performing admission control, and validating resources.

Etcd

Etcd serves as the distributed data store for the Kubernetes cluster, storing configurations, states, and Kubernetes objects. It ensures data consistency and fault tolerance across the cluster. A Kubernetes cluster can have one or more etcd instances, which can be deployed either within or outside the cluster.

Scheduler

The scheduler is responsible for assigning pods to nodes based on resource availability, for balancing workloads, collecting metric data, and enforcing scheduling policies. It plays a key role in resource mapping and ensuring efficient utilization of cluster resources.

Controller manager

The controller manager oversees the operations of various controllers within the cluster. This includes monitoring the health of other components, managing resources, handling autoscaling, managing StatefulSets and persistent volumes, and executing other processes essential for maintaining cluster functionality.

The worker nodes in a Kubernetes cluster are equipped with three primary components essential for container deployment and interaction with the control plane: kubelet, kube-proxy, and container runtime.

Kubelet

The kubelet functions as an agent on each worker node, responsible for maintaining the node's operational state. It communicates with the Kubernetes API server to receive instructions and report node status. Key responsibilities include pod management, monitoring pod and node status, collecting metrics and logs, and managing the lifecycle of containers on the node.

Kube-proxy

Kube-proxy manages the networking aspects of the worker node. It oversees network traffic routing, maintains IP tables to facilitate communication between pods and external clients, ensures session affinity for client connections, supports service discovery mechanisms, and contributes to cluster-wide high availability through load balancing and failover configurations.

Container runtime

The container runtime handles the low-level operations necessary for running containers. This includes resource allocation and management, container image handling, and ensuring container security. Kubernetes supports multiple container runtimes such as Docker [61], containerd [62], and CRI-O [63], allowing flexibility in runtime selection based on specific deployment requirements and operational preferences.
It is important to note that the tasks listed for each component in the master node and worker node are primary responsibilities but are not exhaustive.

2.4.2 Resource management

Each application deployed in Kubernetes requires resources such as CPU, memory, and storage to operate effectively. Within a Kubernetes cluster, these resources are aggregated from all available nodes and allocated by the Kubernetes scheduler. The scheduler evaluates the resource requests specified by the client against the available resources in the cluster to determine appropriate pod placement. It is crucial to note that resources are not shared across nodes; therefore, the maximum resource allocation for a pod must not exceed the resources available on a single node.

Pod resource allocation

For pod resource allocation, Kubernetes utilizes two key parameters: resource requests and resource limits.

- **Resource request** :

  This parameter specifies the amount of resources guaranteed by Kubernetes for a pod's deployment. When the scheduler places a pod on a node, it first considers the resource request parameter. While it is not mandatory to define this parameter during deployment, if it is omitted and the node experiences resource starvation, the scheduler may terminate pods without defined resource requests.

- **Resource limit** :

  Kubernetes allows Pods to consume resources beyond their requested allocation. However, to restrict additional resource consumption, the resource limit parameter defines the maximum amount of resources a Pod can consume. Typically, the resource limit is set equal to or higher than the resource request. Like the resource request, defining a resource limit is not obligatory during deployment. The

primary purpose of the resource limit is to ensure that pods adhere to their allocated resources, preventing excessive resource consumption by certain pods that could impair the performance of other pods in the cluster. Kubernetes enforces resource limits using Linux kernel mechanisms. If a pod attempts to exceed its resource limit, Kubernetes restricts its usage according to these mechanisms. The enforcement methods differ depending on the resource type. For the commonly used resources, CPU and memory, Kubernetes employs two distinct mechanisms available in the Linux kernel, which will be detailed in a subsequent section.

QoS classes

In the Kubernetes ecosystem, QoS classes play a pivotal role in orchestrating resource allocation and pod placement processes. The scheduler utilizes these classes to prioritize deployments based on their resource request and limit specifications.

— **Guaranteed QoS class** :
  This class encompasses deployments where resource requests and limits are identical. The scheduler is responsible for ensuring that the resources allocated are in perfect alignment with the requested amounts. Pods belonging to this class are accorded high priority during the scheduling process.

— **Burstable QoS class** :
  Deployments falling under this class have a defined resource request. However, the resource limit, although not mandatory, must exceed the request. The scheduler commits to the requested amount of resources and permits pods to utilize resources beyond the request up to the defined limit. Pods in this class are assigned a moderate priority level.

— **Best effort QoS class** :
  This class includes pods that lack defined resource requests or limits in their deployment specifications. Despite their ability to function, these pods are not allocated any guaranteed resources. They are accorded the lowest priority during pod scheduling. In situations of resource contention, these pods are most susceptible to throttling or eviction.

In Kubernetes, the initial dimensioning of resources during deployment presents a significant challenge. Resource request and limit parameters are typically assigned based on historical resource consumption profiles and data analysis. The resources of the pod, namely CPU and memory, must be judiciously allocated to prevent frequent resource

starvation and avoid costly resource wastage. The calculation of requests and limits necessitates the consideration of baseline resource usage, which represents the minimum amount of resources required to run the pod in idle mode. Even in the absence of incoming requests to process, an application maintains operational background processes such as service discovery, session management, communication line maintenance, monitoring, and logging.

In addition to understanding the baseline resource usage, it is crucial to comprehend the average (σ_{avg}) and maximum (σ_{max}) resource usage. As an example, if the application is not sensitive to QoS and cost is a priority, the resource request and limits can be selected to align with the Burstable QoS class as shown in equation 2.3 and equation 2.4.

$$Resource\ request = \sigma_{avg} \tag{2.3}$$

$$Resource\ limit = \sigma_{max} \tag{2.4}$$

Conversely, if QoS is a priority, the request and limit can be assigned to align with the Guaranteed QoS class as shown in the equation 2.5.

$$Resource\ request = Resource\ limit = \sigma_{max} \tag{2.5}$$

2.4.3 Dynamic resource allocation

Despite the assignment of request and limit parameters, the workload for pods does not always reach maximum or average resource usage. In fact, the workload fluctuates due to temporal, spatial, and other factors, leading to variations in resource demand [64]. This variability can lead to resource wastage, particularly during periods of low workload demand. To address this issue, Kubernetes offers dynamic resource allocation for pods in the cluster, enabling both horizontal and vertical scaling of pods.

Horizontal pod autoscaling

By default, Kubernetes provides the Horizontal Pod Autoscaling (HPA) [65] to automatically scale pods based on the current workload. The underlying principle is to increase the number of pod replicas to cope with an increasing workload and to terminate replicas as the workload decreases. This approach ensures that the initial resource allocation does

not need to match the maximum or average resource demand, thereby reducing resource wastage.

The HPA operates on a threshold-based mechanism. Static thresholds for selected performance metrics must be set at the beginning of deployments to enable HPA to scale up or down replicas during runtime. These performance metrics can be system metrics, such as pod CPU and memory consumption, which Kubernetes provides by default, or custom/external metrics that can be configured to work with HPA. Leveraging a control loop, the HPA controller continuously fetches the required metrics and compares them with the thresholds to make scaling decisions. For a given time t_i and a performance metric, HPA calculates the required number of replicas $P_{desired}[t_i]$ based on equation 2.6.

$$P_{desired}[t_i] = \left\lceil P[t_i] \times \frac{M[t_i]}{M_{desired}} \right\rceil \tag{2.6}$$

where current replica count denoted as $P[t_i]$, $M[t_i]$ denoted as the current metric value and $M_{desired}$ is the desired threshold for the selected metrics.

Vertical pod autoscaling

Vertical Pod Autoscaling (VPA) [66] is a mechanism designed to dynamically adjust the resource allocations (such as CPU and memory) for a single pod based on the current workload demands. When the workload increases, VPA automatically adds more resources to the pod. Conversely, it removes resources when the workload decreases. This allows for refined resource management, scaling only specific resources according to demand.

VPA is not included as a preinstalled feature in Kubernetes. Instead, it can be integrated as an optional plugin application. The VPA controller is responsible for monitoring resource usage and updating the resource requests accordingly. When scaling is necessary, the VPA controller updates the resource requests, which involves restarting the pod. This leads to temporary unavailability of the application in the pod, potentially impacting QoS.

Starting from Kubernetes version 1.27, it became possible to update certain resource fields of a pod specification without requiring a restart through features like Resource Policy. This improvement helps in maintaining better QoS for application without downtime. However, not all updates might be applicable without a restart, and the specifics depend on the nature of the resource changes. For instance, when scaling-up the resources of a Pod and there are insufficient resources available on the current node, the VPA may

relocate the Pod to another node with adequate resources. In this scenario, the VPA will need to restart the Pod to apply the new resource configuration.

VPA operates with a single replica, presenting a single point of failure risk. This can be particularly problematic for QoS-sensitive applications, such as 5G CN NFs, where high availability and connectivity are critical. Consequently, this research focuses on HPA to ensure redundancy and mitigate the risks associated with single point of failure.

Cluster autoscaler

Kubernetes extends its autoscaling capabilities beyond pod-level to include node-level autoscaling through the Cluster Autoscaler [67]. While pod-level autoscaling, using mechanisms like the HPA and VPA, effectively manages application performance amid workload fluctuations, the resources allocated to pods are limited by the node's resource capacity. To address this constraint, Kubernetes offers cluster autoscaling, which dynamically adjusts the number of nodes in a cluster by adding or removing nodes in response to workload demands. This approach prevents underprovisioning, which could lead to exceeding HPA or VPA limits, thereby affecting performance. It also avoids overprovisioning nodes that remain idle, thereby optimizing operational costs.

In addition to horizontal scaling of nodes, Kubernetes supports vertical node scaling, allowing nodes to adjust their capacity based on workload demands. However, this vertical node scaling feature is currently in beta. To leverage either horizontal or vertical node scaling, the cluster must be deployed on VMs within an IaaS platform, supplemented by external middleware.

2.5 Summary

This chapter comprehensively examines the technological aspects, industry standards, concepts, and software tools pertinent to the cost-QoS trade-off in cloud-native 5G deployments. Detailed are the 5G network architecture, its components, their respective functions, and the initiatives undertaken by telecom standardization organizations to facilitate cloud-native compatibility. Furthermore, existing network automation standards and frameworks applicable for automating complex tasks within 5G networks are reviewed. Subsequently, various cloud models, including service and pricing models, are explored to provide the foundational knowledge necessary for understanding the variability of costs associated with cloud integration. Finally, current software tools and methodologies aimed

at addressing the cost-QoS trade-off through dynamic autoscaling in cloud-based applications are analyzed. The next chapter explores the characteristics and requirements that must be considered in an autoscaling solution, as well as the types of existing autoscaling solutions presented in the literature.

Chapter 3

State of the Art

3.1 Introduction

Dynamic resource allocation in cloud-native environments enables applications to maintain performance across a wide range of workloads while simultaneously reducing operational costs. Prominent cloud service providers, including GCP, AWS, and Microsoft Azure, etc. have already adopted dynamic resource scaling strategies within their cloud infrastructure. These strategies range from basic rule-based autoscaling to more sophisticated custom scaling strategies, such as AI/ML based autoscaling. For instance, AWS Elastic Compute Cloud (EC2) service employs a threshold-based scaling strategy [68] while GKE has introduced an AI-driven autoscaling solution known as Autopilot [69] for its deployments. Recognizing the advantages of these autoscaling strategies, cloud service providers persistently broaden their research efforts in this particular field [70] [71].

Dynamic resource allocation is a well-researched and established concept within the field of cloud-based resource management. Various approaches exist to tackle the dynamic autoscaling problem in cloud environment, all with the shared objective of identifying the optimal cost-performance trade-off for a given use case.

The studies [72] [73] [74] [75] offer an exhaustive classification of autoscaling methodologies as illustrated in the Figure 3.1. These classifications are established considering various elements such as the employed strategies, the encountered challenges, and the essential characteristics within a cloud-based environment.

Identifying the need for dynamic resource scaling and optimizing resource allocation requires a comprehensive understanding of the application architecture. Applications of a monolithic nature, also referred to as single-tier applications, are characterized by the consolidation of all components within a singular physical environment. In the context of autoscaling, these applications are treated as a unified entity. Despite the advantages associated with single-tier applications, such as diminished communication overhead between components and the reduced monitoring services, their scalability is sub-optimal. For in-

Graphic removed to respect copyright

Figure 3.1 – The taxonomy for autoscaling web applications in clouds. (from [72])

stance, when applications necessitate scaling due to resource insufficiency , the autoscaling mechanism will indiscriminately augments resources (CPU, memory, storage, bandwidth, etc.), irrespective of the location of the bottleneck in the application. This approach can potentially lead to an inefficient allocation of resources affecting the operational cost.

Contrarily to single-tier , certain applications possess multiple tiers that can be segregated based on their respective physical environments. This is commonly referred to as multi-tier or n-tier architecture. The most prevalent tiers include the presentation tier, which interacts with the end user, the business logic tier, which delivers the core functionality of the application, and the data tier, responsible for data storage, as depicted in Figure 3.2.

From an autoscaling perspective, this architecture offers greater scalability efficiency compared to single-tier applications. This efficiency is achieved by isolating and scaling only the bottleneck component.

Recently, most modern applications, including 5G networks, are transitioning to SBA, where in applications are broken down into multiple components based on the service they provide. This enhances the granularity of the scalable component from an autoscaler's viewpoint and facilitates more precise resources allocation. Consequently, the autoscaler can identify and scale the targeted component without disrupting the application's functionality.

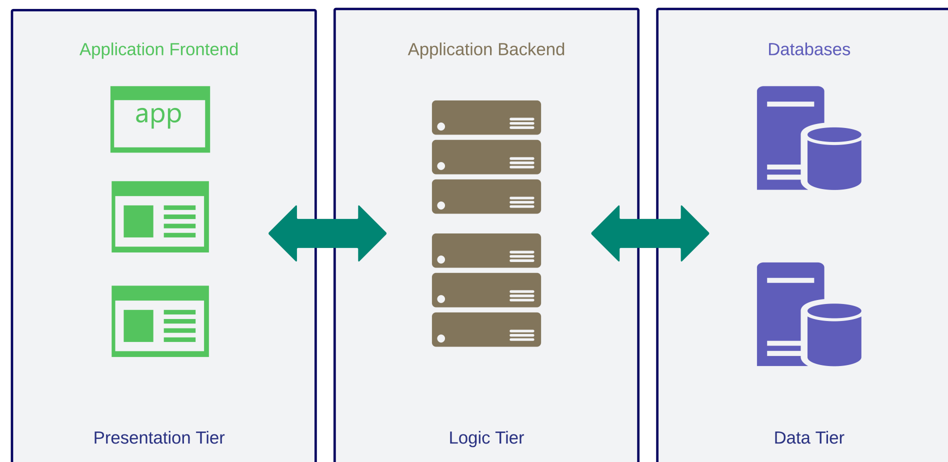


Figure 3.2 – High level overview of three tier architecture. (from [76] - modified) (Copyright © 2016 IEEE)

Despite the numerous advantages associated with both multi-tier and SBA, they also present certain challenges. For instance, both architectures necessitate monitoring for each component, which can be complex and costly to manage.

In cloud environments, session stickiness, or session affinity, which ensures a consistent connection between a user and the same application instance, becomes a critical factor in the implementation of autoscaling methodologies. When an application necessitates the preservation of user session affinity throughout the connection, the act of scaling the application could potentially disrupt this session affinity. Maintain session affinity imposes constraints on horizontal scaling, inhibiting the addition or removal of replicas. It also obstructs vertical scaling by preventing restarts with new resource configurations. Therefore, an effective autoscaling strategy must incorporate awareness of application sessions if it is mandated to maintain session affinity.

Furthermore, stateful applications, which require the preservation of application state, necessitate maintaining consistency across instances during scaling. This requirement differs from stateless applications, where state information is either not retained or is managed externally. Stateful applications also demand additional mechanisms to manage backups and the data recovery process in the event of scaling. Consequently, the cost and complexity associated with implementing dynamic autoscaling differ between stateful and stateless applications. This difference must be taken into account when devising autoscaling strategies.

Autoscaling solutions can be categorized based on their adaptability. Non-adaptive

strategies, such as rule-based autoscaling, operate according to a predefined set of conditions. Scaling actions are triggered solely when these conditions are met, regardless of the application's current status or performance metrics. These strategies are typically suited for applications with less volatile workloads, characterized by lower variability in resource demands. In contrast, adaptive scaling strategies, including queuing theory-based, control theory-based, and advanced techniques such as RL-based methods, offer superior adaptability. These strategies can dynamically respond to the application's current state and performance, potentially enhancing overall efficiency and performance more effectively than non-adaptive strategies. As elaborated in [73], self-adaptive scaling strategies are equipped with prior knowledge and a certain degree of self-awareness, enabling the system to adjust to volatile workloads.

Autoscaling mechanisms operate on a continuous monitoring paradigm, dynamically scaling the target application in response to varying workload and performance status. The efficacy of this autoscaling process is essentially tied to the choice of performance indicators that are tracked for the application. These performance metrics should accurately reflect the application's status, identify which resources require scaling, and the implications of the scaling decision to optimize resource allocation. Certain applications necessitate a focus on low-level or infrastructure-level performance indicators, such as CPU utilization, memory consumption, and bandwidth usage [77], to facilitate accurate autoscaling of the deployment. Conversely, other applications may be more sensitive to service-level performance indicators, such as response time and throughput. It is noteworthy that a subset of applications may derive enhanced autoscaling performance from a hybrid approach, incorporating a mix of both infrastructure-level and service-level performance indicators. This combination of performance indicators can potentially optimize the performance of the autoscaling algorithm. Nonetheless, the choice of performance metrics for the autoscalers strictly depends upon the specific application.

In the domain of autoscaling, one of the most challenging issues is the oscillation of scaling decisions. This phenomenon occurs when a scaling strategy, in response to current conditions, triggers a scaling action, only for the conditions to change shortly thereafter, leading the strategy to reverse its decision within a brief time frame. This cyclical behavior, often referred to as the 'ping-pong effect' in some research literature [78] [79], can adversely impact application performance, including stability and load balancer consistency, as well as result in delays in service initialization. The root cause of this issue is generally attributed to suboptimal configuration of the scaling strategy, with non-

adaptive methods such as rule-based scaling being particularly susceptible. The problem is exacerbated in scenarios involving volatile workloads. To mitigate the oscillation effect, several approaches have been proposed. One such method is the implementation of a cooldown period, during which the system prohibits the reversal of scaling decisions for a predefined time. However, without a thorough understanding of workload patterns, this approach may lead to inefficient resource allocation. Another potential solution is to increase the monitoring interval, which helps to smooth out fluctuations in the monitoring data and thereby reduce the oscillation effect in decision-making. Nonetheless, extending the monitoring interval can reduce visibility into the cluster's status, potentially impeding optimal resource provisioning.

The timing of autoscaling decisions plays a pivotal role in the performance of a deployment. Autoscaling can be implemented in two distinct manners: reactively or proactively. Reactive scaling processes, such as rule-based scaling, control-based scaling,etc. are suitable for cloud applications with slow-moving workloads. These methodologies are reactive in nature, responding to workload fluctuations post-occurrence. Consequently, they are unsuitable for applications with performance sensitivity. On the other hand, cloud applications with volatile workload patterns necessitate proactive scaling. This approach anticipates changes in workload and scales resources accordingly before performance issues arise. By detecting events early, proactive scaling can prevent performance degradation, thereby ensuring the smooth operation of the deployment. Thus, the choice between reactive and proactive autoscaling hinges on the nature of the workload and the criticality of decision timing for the specific deployment. Nevertheless, reactive strategies may offer simplicity in implementation, eliminating the need for prior data collection or training. Conversely, proactive scaling necessitates advanced training for event prediction based on historical data, which may introduce complexity and impose significant implementation costs.

Autoscaling within a cloud context can be executed via horizontal or vertical scaling. The choice between horizontal and vertical scaling is contingent on a multitude of factors. Both horizontal and vertical scaling present their unique advantages and disadvantages. Horizontal scaling, due to its inherent replication, engenders additional redundancy and high availability, thereby circumventing a single point of failure.

Conversely, vertical scaling offers a finer granularity compared to horizontal scaling, as it allows for the reconfiguration of any resource values, while horizontal scaling uniformly increases or decreases a predetermined amount of resources in the original pod. Vertical

scaling is applicable to both stateful and stateless applications, whereas horizontal scaling may necessitate additional software components to synchronize the states of replicas in stateful applications. Certain cloud platforms or versions necessitate application restarts to incorporate new resource configurations in vertical scaling, whereas horizontal scaling is devoid of such requirements. Vertical scaling eliminates the need for internal load balancers, whereas horizontal scaling necessitates them for load distribution among replicas. Not all cloud-based softwares are compatible with horizontal scaling due to its complex implementation nature. Certain softwares may require additional licensing for instance replication, which could impose significant costs for horizontal scaling [80].

In the contemporary era, customers are presented with the flexibility to deploy their applications in public clouds, private clouds, or hybrid environments. This is a critical consideration in the context of autoscaling. For instance, public clouds propose diverse pricing models that enable cost optimization through dynamic scaling [49]. They also operate numerous data centers distributed across various geographical locations, enabling them to scale customer applications effectively and minimize response times and latency.

Conversely, most private clouds lack this degree of cost flexibility and multi-datacenter scalability. If the private cloud is hosted on-premises, dynamic scaling does not incur additional costs but is limited by the system's capacity. In this context, energy consumption becomes a critical factor, necessitating the development of an autoscaling solution that incorporates energy efficiency considerations [81]. This includes employing cluster autoscaling techniques designed to minimize the number of idle nodes, thereby reducing overall energy consumption [82] [83] [84]. Furthermore, customers have the option to adopt a hybrid approach, where application deployment is distributed across private and public clouds, and potentially across different public cloud providers. In such cases, it becomes imperative to consider factors such as pricing models and cost differences among cloud providers in the autoscaling approach to optimize costs.

The majority of cloud service providers offer the capability to deploy applications on either VMs or containers [85]. VMs operate on the host OS and necessitate a guest OS for application deployment. Consequently, during a scaling events, a new VM requires time to load the guest OS and the application, resulting in a significant delay before it is ready for operation. In contrast, containers are lightweight entities that only require the necessary OS dependencies and application images to deploy on the cloud. During a scaling events, a new container also requires some time to become operational, but this delay is significantly less than that of VMs. Therefore, it is crucial to consider these

delay differences and technology differences when implementing autoscaling strategies to optimize both performance and cost.

In the context of this research investigation, autoscaling strategies are systematically classified according to their resource estimation methodologies and decision-making processes. It is crucial to note that despite the aforementioned characteristics inherent to autoscaling techniques, the manner in which these strategies manage constraints while simultaneously maintaining satisfactory performance levels and optimizing cost is of significant importance. This aspect forms a critical component of our study and is examined in detail.

3.2 Rule based autoscaling

The research study [86] underscores the pivotal role of NFV and SDN in the structure of 5G networks. It highlights the requirement for a suitably configured orchestrator to enhance performance of the RAN and CN. To address this requirement, the authors propose a unique NFV orchestrator designed to provide multiple Virtual Network Function (VNF) life cycle functionalities, including scheduling, migration, and scaling as illustrated in Figure 3.3 . In terms of autoscaling aspect, the authors propose an autoscaling mechanism that employs a threshold-based scaling algorithm which is based on the CPU utilization of the VNFs and was evaluated in a Multi-access Edge Computing (MEC)-enabled 5G testbed. The operational logic of the algorithm is to scale up the VNFs when the CPU utilization exceeds a predefined threshold. Conversely, it scales down the VNFs when the CPU utilization falls below a certain threshold. This approach offers the advantage of simplicity in implementation and considers the resource availability of the MEC node prior to executing the scaling decision. However, the authors note that low-level metrics such as CPU usage alone do not offer sufficient detail about the incoming workload, which is vital for decision-making. Furthermore, the VNFs can be CPU-intensive, Memory-intensive, or both. In these scenarios, the decision must take into account which resource is critical for the VNF performance. A significant feature of this solution is the integration of the autoscaling solution into the Network Functions Virtualization Orchestrator (NFVO), providing centralized control over the network. Despite having a simple rule-based scaling solution, making a scaling decision necessitates communication through multiple layers of components in the proposed centralized NFVO architecture, which may not always be ideal for time-sensitive applications. This observation underscores the complexity and

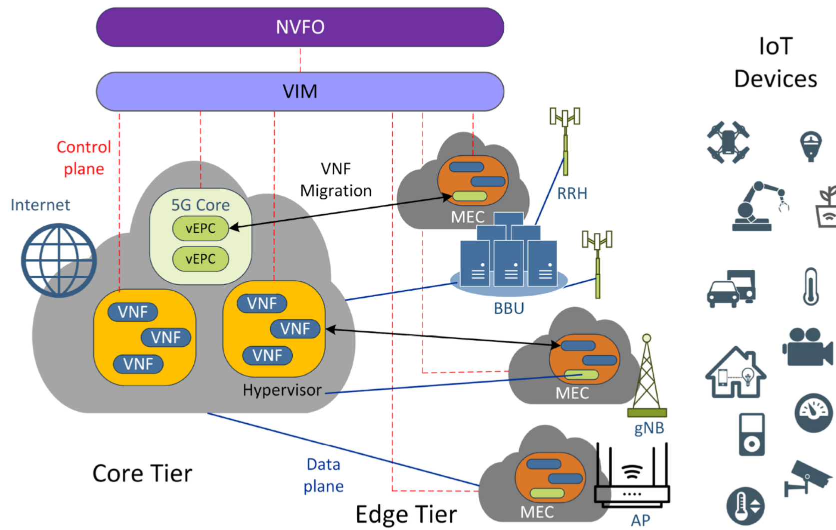


Figure 3.3 – MEC enabled 5G IoT architecture. (from [86]) (Copyright © 2020 IEEE)

challenges inherent in managing and optimizing 5G network performance.

In the work [87], the authors introduce a dynamic autoscaling solution designed to optimize resource allocation in Kubernetes, thereby enhancing QoS. The solution is proposed in response to identified challenges in resource estimation, which arise due to the diverse nature of applications and their associated workloads. The authors critique the existing HPA in Kubernetes, stating that its singular focus on resource consumption is inadequate for managing the complexities of modern workloads. They propose an enhancement to the Kubernetes HPA that implements dual thresholds for scaling the number of replicas both up and down, contingent on their respective CPU and memory usage. A key distinction between the Kubernetes HPA and the authors' solution lies in the incorporation of a smart load balancer in the latter. This component prioritizes assigning new workloads to new replicas as long as they are underutilized compared to existing ones. This strategy not only improves the QoS but also complements the autoscaling solution, thereby enhancing overall system performance. Nevertheless, the scaling process is designed to be dependent on resource consumption, and the scaling-up/down threshold selection does not take into account fundamental operations such as CPU throttling which could affects the QoS during the scaling process.

In the paper [88], the authors articulate the need for dynamic scalability of stateful databases in the context of large data streams. They observe that under high load conditions, the performance of database services can be compromised due to excessive resource consumption. To mitigate this issue, they propose a smart agent, designated as

"SCAL-E". This agent interfaces with the Kubernetes metric server to monitor and scale the repository subsystem components of a database in a GKE environment.

The authors assert that this approach ensures appropriate resource allocation and enhances efficiency in data storage and forwarding. Scaling events are triggered based on the CPU usage of the nodes. Specifically, a scaling-up event is initiated if the CPU usage of any node exceeds the scaling-up threshold, thereby preventing performance degradation of the database. Conversely, if the CPU usage of all nodes falls below the scaling-down threshold, a scaling-down event is initiated. These thresholds are determined through empirical testing on CPU usage and system performance. Given the nature of stateful applications, during periods of high traffic, the database is replicated onto a new node when scaling-up. The authors' evaluation indicates that this approach significantly improves response times compared to a deployment without the SCAL-E agent. However, it is important to note that this approach prioritizes application performance over cost optimization. The replication of a database onto a new node can be costly, which is a factor that needs to be considered in the overall evaluation of this approach.

The paper [89] presents a novel approach to Kubernetes HPA, termed 'traffic-aware' horizontal pod autoscaling (THPA). This approach is specifically tailored to facilitate real-time, traffic-aware resource autoscaling for IoT applications within an edge computing context as demonstrated in the Figure 3.4. The proposed solution carries out both upscaling and downscaling operations, which are informed by network traffic data from nodes. The primary objective of these operations is to enhance the quality of IoT services provided within the edge computing infrastructure. The authors identified a specific issue with Kubernetes' handling of scaling-up events: it distributes replicas evenly and fails to account for the network delay between edge nodes. This problem becomes more significant when the edge nodes are dispersed over a wide geographical area. To mitigate this issue the proposed solution allocates replica proportions in accordance with the traffic load present in the node and implements a scale down operation when the nodes are experiencing low demand.

For time-sensitive IoT applications, consideration of the network delay induced by edge nodes can significantly enhance the performance of the applications. However, it is important to note that the strategy of distributing replicas across nodes solely to minimize network delay could potentially degrade service quality. This is particularly true if the resource capacity of the nodes and the resource configuration of the pod are not taken into account in the autoscaling strategy.

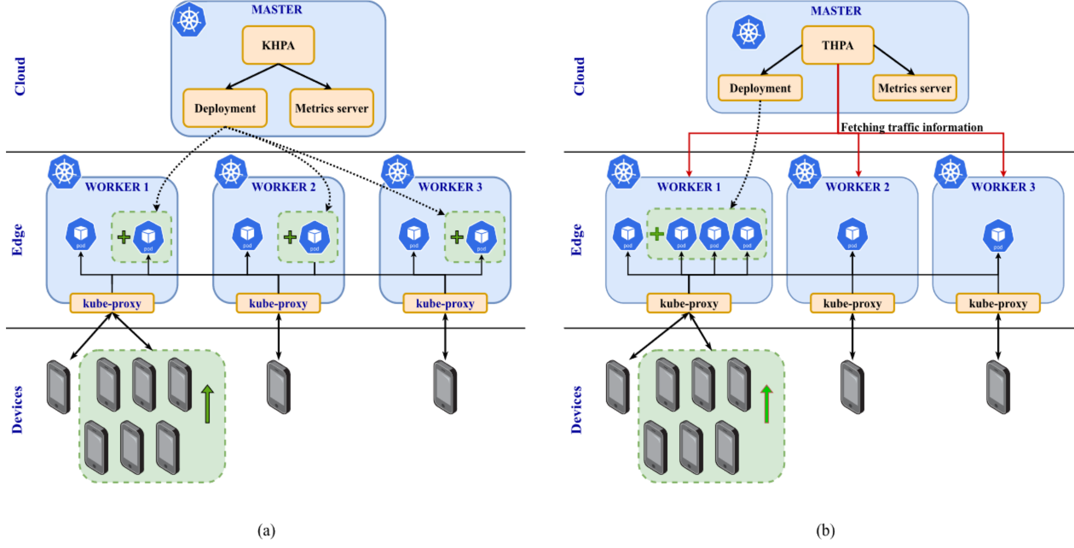


Figure 3.4 – (a) KHPA in Kubernetes-based edge computing architecture and (b) THPA in Kubernetes-based edge computing architecture. (from [89]) (Copyright © 2022 IEEE)

In the paper [90], the authors pinpoint a significant shortcoming in the Kubernetes HPA: its inefficiency in managing scaling during traffic spikes. This problem emerges due to the HPA's default scaling decisions being solely predicated on resource usage, without taking into account the number of requests that can be served while maintaining the requisite QoS levels. This oversight can lead to a QoS degradation.

To rectify this issue, the authors put forth a service-aware autoscaling solution that incorporates user request measurements. This solution retrieves the number of requests from the load balancer via monitoring services. The solution computes the necessary number of replicas using equation 3.1 by adopting a threshold for the maximum number of users that can be served per pod without impacting QoS.

$$N_{pods} = \left\lceil \alpha \cdot \frac{N_{req}}{M_{req}} \right\rceil \tag{3.1}$$

Here, N_{pods} represent the required number of pods, N_{req} represent the current number of requests and M_{req} represent the number of requests a pod can handle. One of the advantage of their approach is authors introduced α parameter to tune the calculation to avoid resource over dimension or under dimension.

However, this approach is only effective in scenarios where there is a proportional relationship between traffic rate and resource consumption, and each pod serves a single

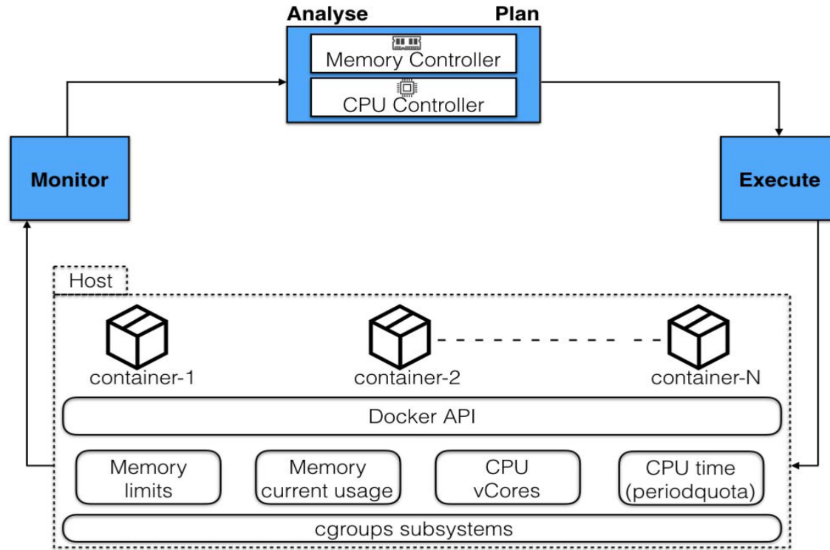


Figure 3.5 – ELASTICDOCKER architecture. (from [80]) (Copyright © 2017 IEEE)

type of service. In situations where a single pod provides multiple services, and each service necessitates different resource consumption per request, relying exclusively on traffic rate for resource scaling may not be optimal. A mix of requests with varying resource consumption per request arriving at the pod does not provide accurate timing for scaling if the decisions are based on the number of requests.

In the research work [80], the authors present an autoscaling mechanism, denoted as ELASTICDOCKER, explicitly engineered for web applications encapsulated and deployed within Docker containers. The authors propose an alternative solution to horizontal scaling, given its inherent disadvantages. This solution autonomously modifies the resources allocated to the containers in a vertical manner, achieved through the integration of the MAPE-K concept as illustrated in Figure 3.5.

The scaling of resources, specifically CPU and memory, is dictated by the workload of the containers. To ensure efficient resource allocation, the solution implements two distinct thresholds for both CPU and memory, with the scaling-up threshold set at 90% and the scaling-down threshold at 70%. To maintain container stability post-scaling, a 'breathing period' is incorporated, allowing the containers to stabilize their state following the scaling process.

However, both vertical and horizontal scaling possess their respective advantages and disadvantages. Therefore, the selection of an approach is contingent upon the application requirements. Moreover, the selection of thresholds should be predicated on the applica-

tion's historical behavior. In this study, the authors establish a relatively high scaling-down threshold, beneficial for reducing operational costs in consumption-based pricing models. However, this may not always be conducive to maintaining application performance in volatile workloads, as performance can degrade in high-resource usage situations. Furthermore, the scaling is executed through static increments and decrements and does not dynamically adapt to the workload, potentially affecting performance under high resource usage.

In the research [91], the authors investigate the feasibility of non-disruptive autoscaling within the Kubernetes environment. They identify a significant issue with Kubernetes VPA in versions 1.26 and earlier, which necessitates an application restart during autoscaling. This disruption can negatively impact the current state of the application, particularly for those that are performance-sensitive. As a solution, the authors propose a novel vertical autoscaling approach, termed as RUBAS (Resource Usage Based Autoscaling). RUBAS leverages Checkpoint Restore in Userspace (CRIU) technology to maintain the application's state during the scaling process. A key advantage of RUBAS over Kubernetes VPA is its ability to migrate applications to a new node during scaling if the current node lacks sufficient resources for the new resource configuration.

RUBAS operates by monitoring resource usage and determining the required resource configuration based on the median resource usage plus a buffer, which is defined as the positive deviation of the observation. This method offers an advantage over the peak resource usage calculation used in Kubernetes VPA, as it helps to prevent resource overestimation. Furthermore, RUBAS exhibits adaptability to the current container situation, dynamically adjusting resources as needed. This dynamic adjustment provides a significant improvement over traditional threshold-based solutions, which are static and may not adequately respond to changes in resource demand. Thus, RUBAS presents a promising alternative for efficient and non-disruptive autoscaling in Kubernetes.

Despite the superior performance of the proposed solution in comparison to Kubernetes VPA, the authors have identified several potential limitations. First, the container may continuously migrate from one node to another without any processing during scaling. Secondly, multiple restarts can occasionally lead to the generation of corrupt container images. The autoscaling model relies on median resource usage, making the monitoring interval a critical factor in estimation accuracy. The authors suggest that smaller monitoring intervals may lead to inaccurate estimations and increased runtime. In Kubernetes, container migration involves several steps, which may delay the new container becoming

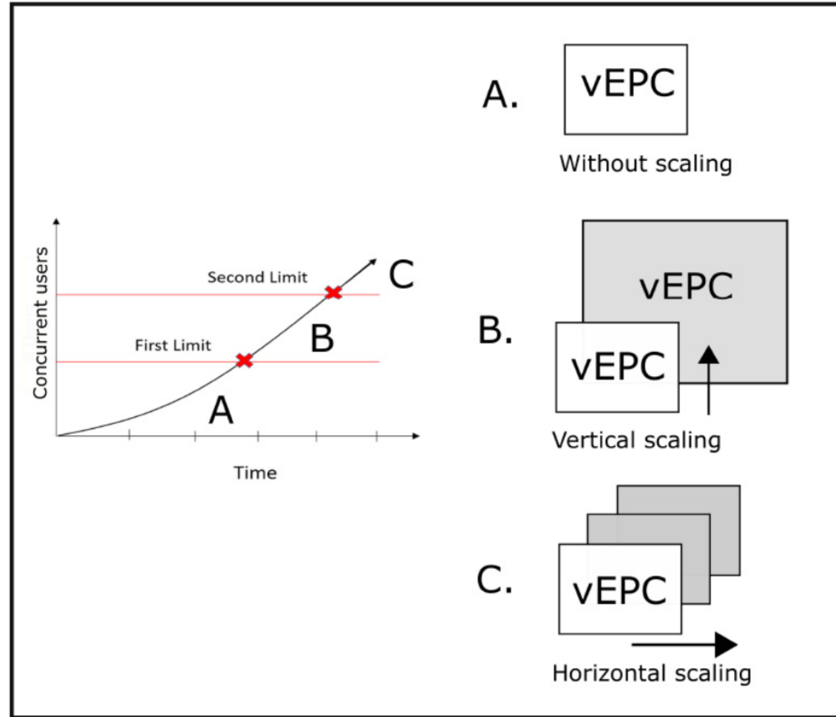


Figure 3.6 – Combining vertical and horizontal scaling. (from [92]) (Copyright © 2020 IEEE)

operational. This delay can impact QoS. This is particularly significant in VPA, where there is only one operational replica, compared to multiple replicas in HPA.

In the article [92], the authors explore the integration of horizontal and vertical scaling strategies for the EPC within a cloud-based environment. The proposed methodology is predicated on a threshold-based approach, utilizing the number of concurrent user connections to the EPC as a metric for scaling the Mobility Management Entity (MME), Serving Gateway (SGW), and Packet Data Network Gateway (PGW). Two thresholds T_1 and T_2 are defined where $T_1 < T_2$ as shown in the Figure 3.6. These thresholds serve as decision parameters for determining the necessity of horizontal or vertical scaling for the EPC. If the number of concurrent users is below T_1, scaling is deemed unnecessary. Conversely, if the number of concurrent users falls between T_1 and T_2, vertical scaling is implemented to adjust the resources. However, if the number of users exceeds T_2, horizontal scaling is initiated. The selection of these thresholds was based on previous experiments, taking into account the CPU usage per concurrent user and the number of registrations. The proposed solution was deployed and evaluated in AWS cloud envionment by implementing EPC functions in VMs and utilizing eNodeB simulator instances to generate UE traffic.

This approach is able to increase the user registration per second and reduces the latency compared to EPC without scaling capability. However, in high-traffic scenarios, the application of two distinct thresholds for horizontal and vertical scaling can lead to uncertainty in determining the appropriate scaling strategy. The startup delay associated with VM initialization further complicates this issue. Specifically, if the vertical scaling threshold is reached—necessitating a pod restart—and the number of concurrent users exceeds the horizontal scaling threshold before the new pod becomes fully operational, conflicts can arise in the autoscaling decisions.

The paper [93] presents a scaling process for VNF within the 5G data plane based on the bit rate of incoming traffic across various network slices. The solution is implemented by deploying multiple data planes across different network slices, each with distinct bit rate configurations. These deployments are housed within a lightweight container in a LXC [94] environment and the solution monitors the throughput of each data plane. The proposed horizontal scaling mechanism operates on a threshold basis, scaling-up when the threshold is reached. Following the scale up, the mechanism waits for the completion of processes within the VNF before allowing a scale down. The authors conducted experiments to test the scalability of containerized MME, SGW, and PGW and evaluated their scaling solution by injecting traffic using the iperf3 traffic generator.

With carefully selected thresholds this approach can effectively counteract any decreases in throughput, thereby ensuring that the data planes maintain the anticipated levels of throughput. However, it is important to note that scaling strategies based solely on throughput, without taking into account the resource utilization of the network functions, may not always yield optimal results. This is because such an approach oversee which specific resource deficiencies that impacting performance. Additionally, scaling the entire data plane solely based on throughput, without identifying the specific bottleneck causing performance issues, may result in over-provisioning resources across the entire data plane, thereby significantly increasing costs.

Within the 5G architectural framework, the decoupling of the UP from the CP enables distributed deployment and allows telecom operators to independently autoscale UPs. The authors of [95] assess this architecture utilizing their innovative autoscaling solution, which are predicated on the Vector Packet Processing (VPP) rate. VPP is a method employed to augment hardware performance, thereby enhancing the efficacy of software-based networks.

The authors suggest that the VPP vector rate is an appropriate metric for autoscaling

data plane functions. The proposed autoscaling solution utilizes thresholds to scale the SGW and PGW network functions in a 5G Non-Standalone (NSA) network, integrating the Monitor, Analyze, Plan, Execute over a shared Knowledge base (MAPE-K) concept to automate the process. However, the authors dismiss the idea of employing CPU utilization as a decision-making indicator, arguing that the polling mode in the Data Plane Development Kit (DPDK)/VPP environment leads to high CPU utilization, even when packet processing is not occurring. Unlike the CP, the UP is required to process the data traffic flowing through it. Therefore, monitoring the VPP vector rate in a softwarized UP is essential to comprehend the volume and patterns of traffic to be processed.

Nevertheless, determining the required resources for processing solely based on monitoring the VPP vector rate does not always produce accurate results. This limitation arises because the UP supports various service types, each with distinct resource demands per request. Consequently, the VPP vector rate alone does not accurately reflect the actual resource consumption of the UP. As a result, scaling the network function based solely on this metric may not be efficient. There can be scenarios where the VPP vector rate is low, yet resource consumption is sufficiently high to affect performance, which would go undetected by an autoscaling solution that relies exclusively on VPP vector rate monitoring.

Inherent to their design, rule-based autoscaling strategies operate on a reactive basis. The autoscaling mechanism is triggered by events that have already transpired or are currently in progress. During the scaling-up process, a primary challenge is the latency between the initiation of the scaling mechanism and the point at which the new replicas are prepared to manage requests. This latency can fluctuate based on the specific network function. Throughout this latency period, the operational but overloaded VMs or containers persist in receiving user requests (unless there is an additional smart load balancing mechanism that is aware of the scaling decisions). As a result, due to resource scarcity, the service response time may be adversely affected. While this method may offer an adequate scaling solution for applications with slow-moving workloads, applications with volatile workloads may experience difficulties. However, by carefully selecting rules that take this delay into account, it is possible to mitigate resource starvation for the incoming workload. A majority of rule-based autoscaling solutions employ a buffer zone to avoid this issue, which involves setting the lower thresholds below the maximium resource capacity of the NF. This approach inhibits the auto scaler from optimizing the operational cost associated with resource usage. However, even with an adequate buffer

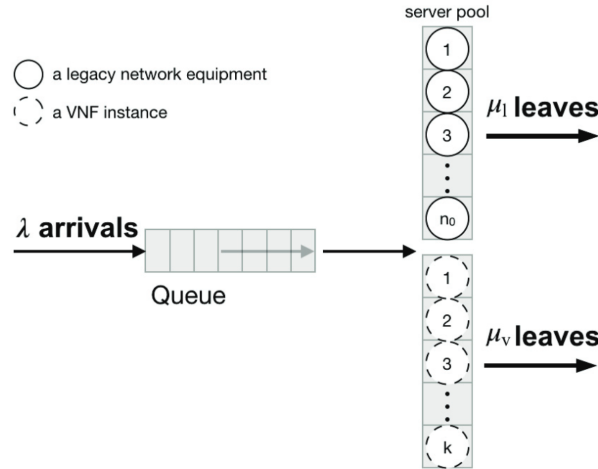


Figure 3.7 – A simplified queuing model of the system. (from [96]) (Copyright © 2018 IEEE)

zone, rule-based autoscaling cannot always ensure that it can accommodate all workload variations. This underscores the necessity for a more dynamically adjustable and proactive scaling solution.

3.3 Queuing theory based autoscaling

In the work [96], the authors address the challenge of integrating legacy network equipment with VNFs for network operation. They argue that hardware-based legacy networks offer superior cost-effectiveness and performance compared to VNF-operated networks. In the context of 5G, they emphasize the importance of integrating both types of networks, taking into account cost and performance considerations.

The authors propose leveraging the dynamic scalability of VNFs to enhance network performance in conjunction with legacy networks, while being mindful of cost implications. To this end, they introduce an adaptive scaling algorithm for VNFs, which is based on a system model grounded in queuing theory as illustrated in Figure 3.7. In their system model, they quantify the cost-performance trade-off based on several parameters: the number of active legacy servers, the number of active VNFs, the average service response time in both types of networks, and the associated costs. These parameters are used to calculate the service rates for both legacy networks and VNFs. VNF scaling is performed based on predefined thresholds tied to the number of waiting requests in the system queue. This process is simulated using the NS2 simulator.

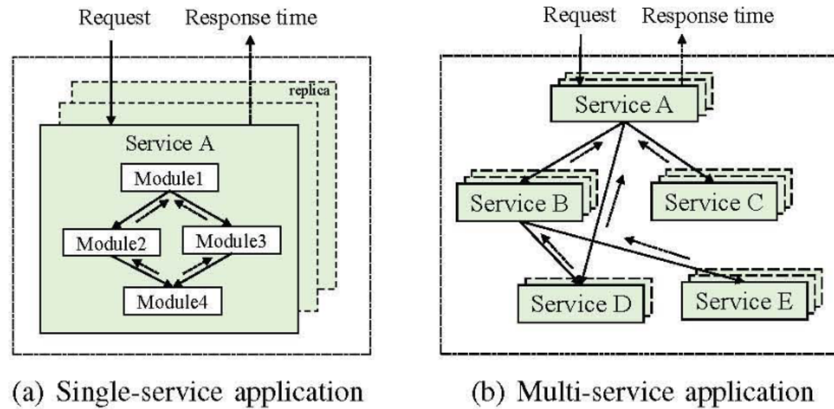


Figure 3.8 – (a) Single-service application composed of one service which may contain different modules. (b) Multi-service application whose invoking relationships of services. (from [97]) (Copyright © 2021 IEEE)

Despite the ongoing transition of 5G networks to the cloud, telecom operators are not expected to phase out legacy networks in the immediate future. In such scenarios, the proposed autoscaling solution offers a valuable means of balancing the cost-performance trade-off. However, this solution does not account for the resource consumption of the VNFs or the application architecture, and relies solely on the system queue length to scale the VNFs. Consequently, if the root cause of the service response time is resource scarcity or starvation, such scenarios remain undetected by this autoscaling approach.

In [97], the authors tackle the challenges associated with autoscaling multi-service applications. These challenges stem from the complex architecture of such applications as shown in the Figure 3.8, the intricate correlations between services, and the bottlenecks that arise due to untimely decisions. These factors complicate the optimization of cost and performance, often leading to direct violations of SLAs.

While many autoscaling solutions attempt to address these issues by independently scaling each service, most fail to optimize cost and performance effectively. They also struggle to prevent bottlenecks arising from untimely scaling decisions, largely due to their lack of consideration for the relationships between services.

The authors underscore that autoscaling multi-service applications can be particularly challenging due to several factors. For instance, maintaining a balance between cost and performance becomes difficult when the workload is dynamic, especially during sudden bursts. Additionally, identifying the application's bottleneck becomes complex when there are intricate correlations between different services.

To mitigate these issues, the authors propose a scaling solution that models application services based on the Jackson Queuing Network (JQN), thereby reducing the decision space. They argue that commonly used queuing models, such as M/M/n or M/G/1, in multi-tier applications assume that the service arrival process and service rate are independent of services. However, the JQN model is deemed superior as it considers service correlations among services and views the multi-service application as a whole.

The authors acknowledge the difficulty of implementing a generalized autoscaling model based on this approach for all multi-tier applications. To simplify the solution, they impose certain constraints: all the services form a directed acyclic graph, there is only a single request access point, no external service has access to any internal services, and all external services accessed by the internal services guarantee service quality.

For modeling the autoscaling based on the queuing model, the authors propose a proportionality coefficient measurement strategy. This strategy maintains a balanced traffic stream among services and calculates the required number of instances based on thresholds implemented on response time, as per the queuing model. The authors consider CPU utilization, the number of requests, and service time in the modeling process.

The authors evaluated their solution in a Kubernetes cluster by deploying a multi-service application. Their solution collects traffic and resource usage metrics, as well as response time metrics, through monitoring services and makes real-time scaling decisions. The solution is benchmarked against other solutions, such as M/M/n/PS queuing model-based autoscaling, RL-based autoscaling, and threshold-based Amazon autoscaler. According to the authors, their solution excels in reducing cost and SLA violations.

One of the main advantages of this solution is its incorporation of both service-level and infrastructure-level data to model the multi-service application autoscaler. However, as the application architecture becomes more complex and correlations increase, the modeling of the application also becomes more complicated to implement. Additionally, certain metrics, such as service rates, can vary widely due to factors such as application language and compiler optimization, which can potentially mislead the autoscaler.

In the paper [98], the authors assert that most cloud-based autoscalers employ either horizontal scaling or vertical scaling, but not both simultaneously. The authors explain that horizontal scaling does not always ensure high resource utilization due to the added buffer zone when selecting scale up thresholds, which impacts cost optimization. On the other hand, vertical scaling encounters a performance ceiling issue, implying that the continuous addition of resources does not lead to a corresponding increase in performance,

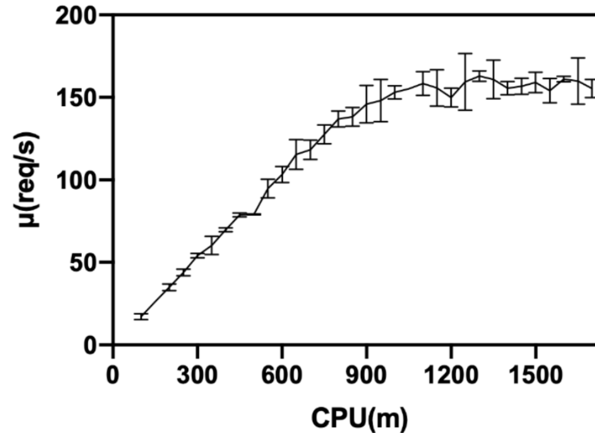


Figure 3.9 – The relationship between supply of resource and service performance. (from [98]) (Copyright © 2021 IEEE)

as depicted in the Figure 3.9.

As a solution, the authors propose an autoscaling solution based on queuing theory that utilizes both HPA and VPA in a Kubernetes environment.

The proposed solution collects both infrastructure-level and service-level metrics using monitoring services in Kubernetes. Based on the collected metrics, the solution first calculates the required number of pods and new resource configurations for each pod based on a mapping system. This mapping system maps high-level metrics to low-level metrics under different resource configurations to calculate the required number of requests that can be processed per second. Subsequently, it calculates the resource cost based on the cloud provider's pricing model.

Using the M/M/c queuing model, the solution obtains the service availability parameter from SLAs. After these steps, the solution computes a score for the new scaling scheme. It then compares the score of the new scheme with the current one, and the scheme with the highest score is executed.

This solution integrates HPA and VPA, enabling it to capitalize on the strengths of both methodologies while mitigating their limitations, particularly in cost optimization. However, this solution is best suited for simple monolithic applications, as implementing it in more complex architectures may prove challenging. For instance, in multi-service applications, calculating service availability might not be straightforward with this queuing model method.

In the paper [99], the authors emphasize the necessity for dynamic resource allocation for CP NFs in the Long-Term Evolution Evolved Packet Core (LTE EPC). They propose

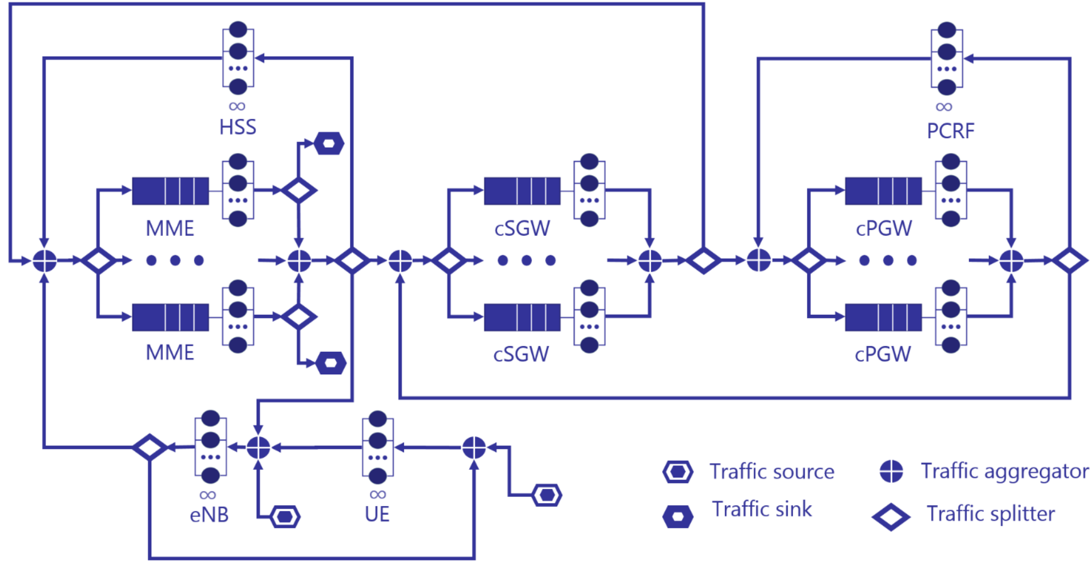


Figure 3.10 – LTE CP queuing model. (from [99]) (Copyright © 2018 IEEE)

an autoscaling solution based on a processing delay budget, which models the CP using the G/G/m queuing model as illustrated in Figure 3.10.

When calculating the delay budget, the authors utilize the 3GPP specification as the performance requirements for CP latency. This latency, as defined by the 3GPP, is the average time taken to transition a UE from an idle state to an active state.

The authors formulate the resource allocation problem as an optimization problem. The objective is to minimize the number of CPU cores allocated to each NF, subject to the condition that the service time is less than or equal to the CP latency specification. Additionally, there is a constraint on the maximum number of CPU cores that can be allocated to a single VNF.

Based on the modeled average response time, the solution calculates the processing delay budget and allocates CPUs to VNFs until the budget is fulfilled. The optimization algorithm iterates multiple times to find the optimal distribution of CPU resources. A key advantage of this approach is that it models the entire CP and customizes it to comply with the standards, thereby reducing resource wastage.

This approach is limited by its inability to directly pinpoint the bottleneck VNF, requiring multiple iterations to do so. Consequently, its computation-intensive nature may render it unsuitable for managing fluctuating traffic.

Just as rule-based autoscaling, the majority of queuing theory-based autoscaling approaches are reactive in nature. The primary objective of these methodologies is to as-

certain the response time of cloud-based applications. This is achieved by modeling the system based on a queuing model and subsequently allocating resources to uphold an acceptable response time.

However, these approaches face certain challenges. Once a discrepancy in response time is detected, a certain amount of time is required to stabilize it through scaling, due to factors such as start-up delay. Furthermore, these methods are computationally intensive, which could prolong the process of resource optimization.

3.4 Control based autoscaling

In the realm of control engineering, Proportional-Integral-Derivative (PID) control is a prevalent method employed to ensure system stability. The PID control mechanism enables the regulation of both the input and output of a system, even when the system model is not known. The PID controller operates by processing the difference between the actual output and the desired output, also known as the error using a feedback loop. This error is then used to adjust the system input $u(t)$ to achieve the desired output. The PID control function comprises three components where the proportional control component K_p aggregates terms that exhibit a high correlation between the error $e(t)$ and the desired output. The integral control component K_i takes into account the historical error data and the derivative control component K_d processes the rate of change of the error as expressed in equation 3.2. The PID controller utilizes these components to implement countermeasures that aim to minimize the error. This ensures that the system output aligns closely with the desired output, thereby enhancing system stability and performance as illustrated in Figure 3.11.

$$u(t) = K_p \cdot e(t) + K_i \int_0^t e(t)dt \ + K_d \cdot \frac{d}{dt}e(t) \tag{3.2}$$

In the study [76], the authors employ PID control theory to dynamically adjust the underlying resources, thereby maintaining the performance of the application. The authors utilize this approach to horizontally autoscale servers in a cloud environment, to maintain a predefined response time. The authors concentrate on scaling CPU-intensive applications by monitoring the application's CPU usage as an input parameter and evaluating the response time as the output metric. The authors posit that saturated CPU utilization results in increased service response times. By adjusting the number of VMs, they increase the CPU capacity, which in turn reduces the response time. To maintain a predefined

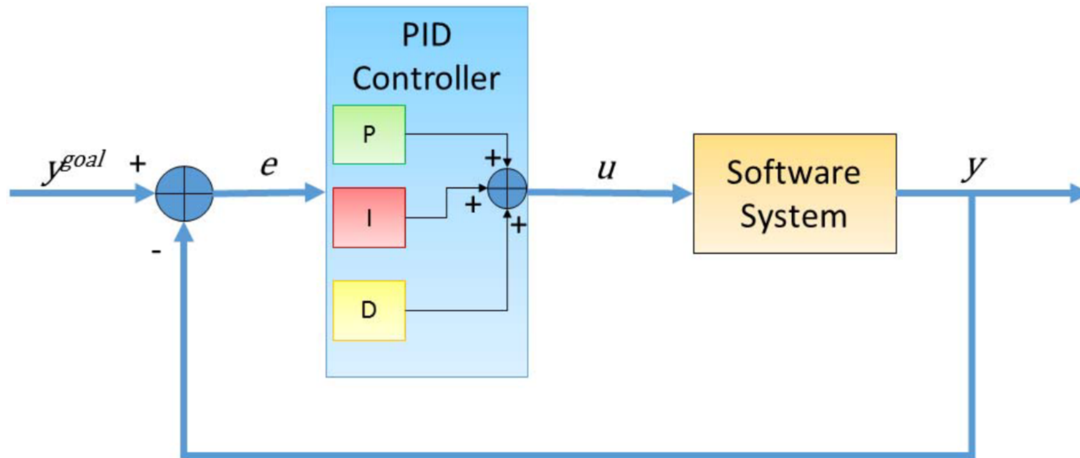


Figure 3.11 – PID feedback control loop. (from [76]) (Copyright © 2016 IEEE)

response time, the authors tune the PID controller to add or remove servers from the cloud environment. This dynamic adjustment of resources presents an advantage over static threshold-based scaling.

However, the solution is not immune to oscillation if the workload is volatile. The response time can change if the cause of the response time change is attributed to another tier, potentially misleading the algorithm to scale the deployment. Moreover, the PID controller requires online tuning, which can cause QoS degradation until the model stabilizes. In this study, the authors did not employ any auto-tuning mechanism, which can prolong the tuning process. Furthermore, if the traffic pattern or system configurations change drastically, the PID controller needs to be re-tuned to adjust to the new configuration.

Similar to the previous work, [100] proposed an autoscaling solution predicated on PID control employed to uphold QoS levels. This is achieved through the horizontal scaling of pods within a Kubernetes cluster. The research introduces three feedback signals, which are predicated on the average weighted workload, the request waiting time, and the CPU utilization. The augmentation of these feedback signals enhances the visibility of the cloud environment, thereby increasing the optimality of decision-making processes. However, this enhancement also escalates the complexity of the PID tuning process, given the multitude of features contributing to the error. Consequently, this increases the time required for online tuning, which may adversely impact the performance of the application.

In the research conducted by [101] , it is explicated that an application's resource consumption is intrinsically linked to the language in which the application is written. The

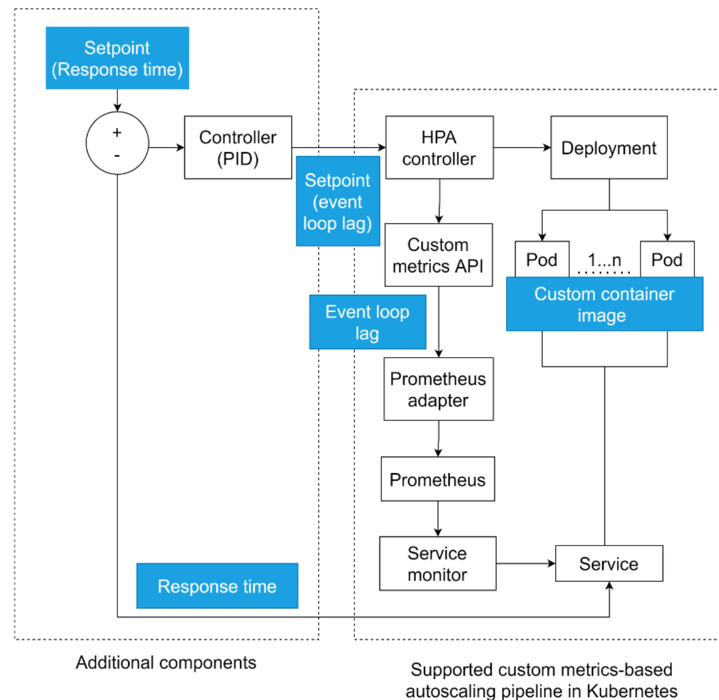


Figure 3.12 – Autoscaling architecture with the PID controller. (from [101]) (Copyright © 2022 IEEE)

resource consumption is influenced by several factors including, but not limited to, the number of threads employed, the garbage collection methodologies implemented, and the compiler optimizations applied. These factors necessitate consideration during the formulation of an autoscaling solution. The authors present an autoscaling solution tailored for Node.js applications, distinguished by their event-driven architecture. Contrary to conventional methods that depend on resource consumption - a metric that can be deceptive for applications such as Node.js - the proposed solution is predicated on response time. The fundamental principle is the utilization of a PID controller, which dynamically adds or removes pods from the cluster to uphold a predetermined response time. Oscillations are managed by instituting a cool-down period in Kubernetes, enhancing the stability of the system as shown in the Figure 3.12.

This methodology can potentially optimize the performance of the application by scaling pods based on application response time. However, it is noteworthy that for multi-tiered applications, this approach could potentially misguide the algorithm. The response time of one tier could be affected by another, resulting in imprecise scaling decisions. This caveat underscores the complexity of devising a universally applicable autoscaling

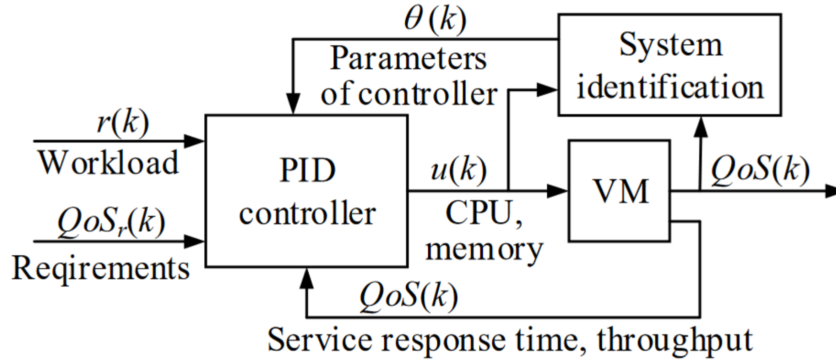


Figure 3.13 – Overview of QoS management control system. (from [102]) (Copyright © 2018 IEEE)

solution.

In the study [102], an autoscaling solution based on PID control is proposed. This solution serves as a mitigation strategy against the observed degradation of QoS in static threshold-based autoscaling solutions due to their inherent lack of adaptability. The primary goal of this solution is to dynamically adjust the allocation of CPU and memory resources in VMs deployed in a cloud environment, in response to changes in both workload and QoS parameters. The system model, illustrated in Figure 3.13, consists of two main components: the PID controller and the system identification module. The PID controller actively adjusts resources to maintain a predefined QoS level by adding or removing resources from the VM. In contrast, the system identification module is responsible for the online tuning of PID parameters.

For the tuning of PID parameters, the authors utilize the Forgetting Factor Recursive Least Squares (FFRLS) technique. This technique functions by comparing estimated QoS values with actual QoS values and selecting the appropriate coefficient values. It then applies the updated PID parameters to minimize the discrepancy between these estimated and actual response time values. The inclusion of this module eliminates the need for manual intervention in PID parameter tuning. However, despite the advantage of dynamic resource adjustment, the proposed method requires online tuning of the PID controller, which can potentially impact QoS during the tuning process.

Fuzzy logic, a variant of control theory utilized in control engineering, is capable of decision-making under conditions of uncertainty and ambiguity. It introduces the concept of partial truths, which exist between the extremes of absolute truth and falsehood, a stark contrast to threshold-based solutions. This is achieved through the implementation

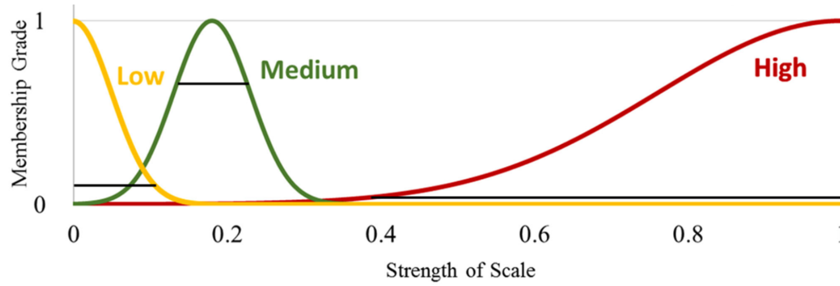


Figure 3.14 – Custom membership function to define the status of the input metrics. (from [103]) (Copyright © 2018 IEEE)

of a membership degree function.

Subsequently, these membership degrees are associated with fuzzy rules, which articulate the system's input-output relationship, thereby facilitating decision-making processes. This methodology enables the system to be controlled in a smooth and adaptive manner, a significant improvement over rule-based mechanisms, which can often result in abrupt decision impacts. This technical approach to control theory underscores the versatility and adaptability of fuzzy logic. In the paper [103], a horizontal autoscaling method for fog computing applications is proposed, leveraging the benefits of fuzzy logic in control systems. The authors underscore the necessity for an autoscaling solution capable of dynamically managing workloads. They note that while existing rule-based scaling is straightforward to implement, it lacks adaptability. To address this, the authors propose an autoscaling method that takes into account average CPU, memory, and network usages as inputs. The authors define the status of these input metrics (high, medium, and low) based on a custom membership function, as depicted in the Figure 3.14. Depending on the status of each metric, different fuzzy rules are set to scale the VNF instances, as stated in the Table 3.1. This approach offers a smoother transition between degrees compared to rule-based scaling and exhibits greater adaptability. However, the complexity of the fuzzy rules increases with the addition of managing features. To implement these fuzzy rules, prior knowledge of the application behavior in the environment is required. Furthermore, the selection of fuzzy rules must be done judiciously to optimize performance and operational cost, adding to the complexity.

Autoscaling solutions based on control theory offer a level of adaptability that is not achievable with rule-based autoscaling solutions. Techniques such as PID control and fuzzy logic are employed to dynamically adjust resource allocation in response to real-time conditions in the cloud environment. These methods calculate the optimal quantity

CPU Usage	Memory Usage	Network Usage	Scaling
High	High	High	High
High	-	-	High
Medium	-	-	Medium
Medium	High	Medium	Medium
Medium	Low	Low	Medium
Medium	Low	Medium	Medium
Low	-	Medium	Low
Low	-	-	Low
Low	-	Low	Low
Medium	Low	-	Low

Table 3.1 – Rules in fuzzy database. (from [103]) (Copyright © 2018 IEEE)

of resources to add or remove, potentially offering a more cost-effective solution compared to solutions that add or remove a predefined quantity of resources. Furthermore, these techniques are sensitive to even minor fluctuations in performance metrics and can take action to maintain performance within acceptable bounds. However, it is important to note that, akin to rule-based autoscaling solutions, control-theory-based autoscaling solutions are reactive in nature. They only take action in response to detected changes, which may render them ineffective for handling volatile workloads, particularly when considering the start-up delay associated with containers or VMs in the cloud.

3.5 Reinforcement learning based autoscaling

According to [104], the majority of open-source autoscaling solutions predominantly rely on either workload-based or resource usage-based approaches. In this study, the authors experiment with the application of Reinforcement Learning (RL) to resource-based autoscaling with the aim of enhancing scaling decisions by reducing response time and utilizing fewer resources.

The authors propose a solution to enhance Kubernetes HPA by dynamically adjusting the scaling threshold using a Q-learning-based RL model. The learning agent monitors CPU usage and response time, and decides whether to increase, decrease, or maintain the current scaling threshold. Given the continuous nature of CPU usage values, the authors employ a quantization method to convert CPU usage into discrete values for use in Q-learning models.

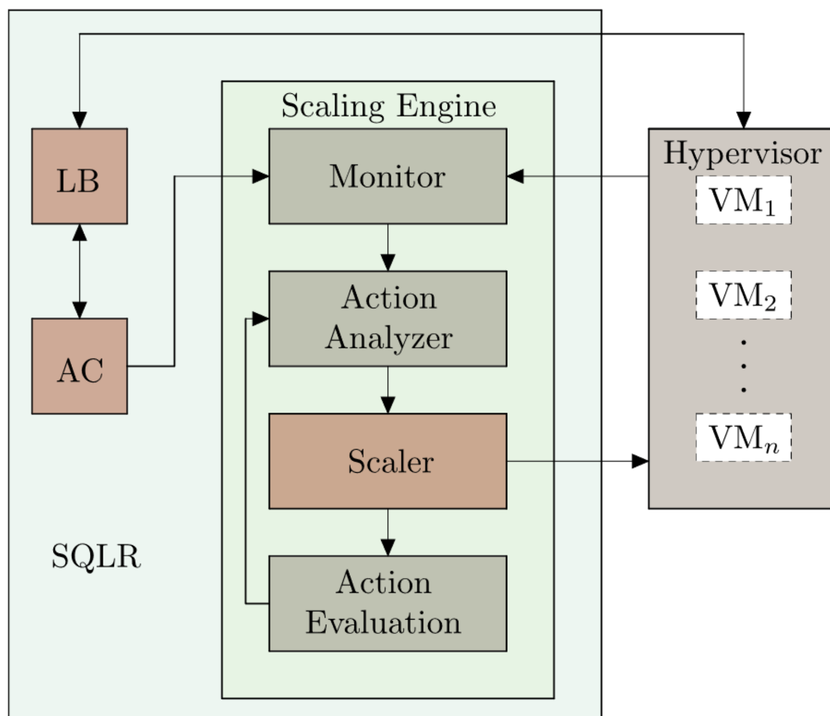


Figure 3.15 – SQLR Block diagram: 'LB' represents the Load Balancer agent, while 'AC' denotes the Admission Control agent. (from [106]) (Copyright © 2021 IEEE)

One of the primary advantages of this approach is its potential to minimize the effects of CPU throttling on QoS, which is primarily caused by CPU usage reaching limits in Kubernetes, by detecting increased response times and scaling replicas. Additionally, the quantization of CPU usage values to reduce the state space can expedite model convergence. However, this quantization process also diminishes visibility into the cloud environment, which can lead to suboptimal scaling decisions that impact performance and operational cost. Furthermore, the model relies solely on infrastructure-level data and lacks visibility into workload data, which could complicate resource allocation in volatile environments when there are complex correlations between resource consumption and workload.

Studies [105] and [106] highlight that most contemporary autoscaling models fall short in adapting to dynamically changing workloads in VMs, thereby impacting performance. As a countermeasure, the authors propose a staged horizontal autoscaling solution for VMs, leveraging multi-agent RL strategies as illustrated in the Figure 3.15.

In the first stage, the authors employ a generic load balancer that distributes traffic based on the CPU utilization of the VMs. The underlying rationale is to direct traffic to

the replicas with the least CPU usage, thereby averting performance degradation due to high resource consumption of the replicas.

In the second stage, a RL agent is introduced for admission control. If a VM can process an incoming request within an acceptable response time, the request is admitted; otherwise, it is dropped. In the third stage, another RL agent, referred to as the informant learning agent, makes scaling decisions based on metrics developed on the CPU utilization of the VMs. To tackle this, the authors propose a novel mapping method that converts continuous parameters into discrete values for integration into Q-learning models.

This solution decomposes the problem into multiple stages and employs a multi-agent method, which reduces the state-action space for each agent and could potentially increase the convergence rate. Additionally, the use of a load balancer that directs traffic based on CPU usage minimizes the immediate impact of scaling decisions on performance.

However, these three stages operate independently of each other, leading to challenges in reaching optimal decisions. For instance, even though the load balancer directs traffic based on CPU usage, it does not guarantee that requests will not be blocked by the admission controller. This could occur when all replicas have high CPU usage and require the third stage to scale up the deployment. Simultaneously, the penalty comparison of violating SLAs by dropping requests versus serving the request with a higher response time due to high resource consumption is not studied. This could indirectly impact the cost-performance trade-off.

In the paper [107], the authors underscore the limitations of threshold-based scaling mechanisms such as HPA in handling highly dynamic workloads. As a remedy, they propose an autoscaling solution based on Deep Reinforcement Learning (DRL). Despite the ability of RL models to learn from environmental interactions, the authors opted for DRL models over tabular-based RL models due to the continuous nature of the features involved in the scaling problem.

The primary objective of this study is twofold: to minimize SLA violations resulting from increased service response time, and to maximize resource usage, thereby reducing operational costs. The proposed solution monitors average CPU usage, the number of current replicas, and the user request rate of deployments, using these as inputs for the DRL model. However, the authors have restricted their action space to scaling-up or down by 1 or 2 replicas, or no scaling.

To optimize the model, the authors have designed a reward function based on CPU utilization and SLA violations due to increased response time. The function is designed

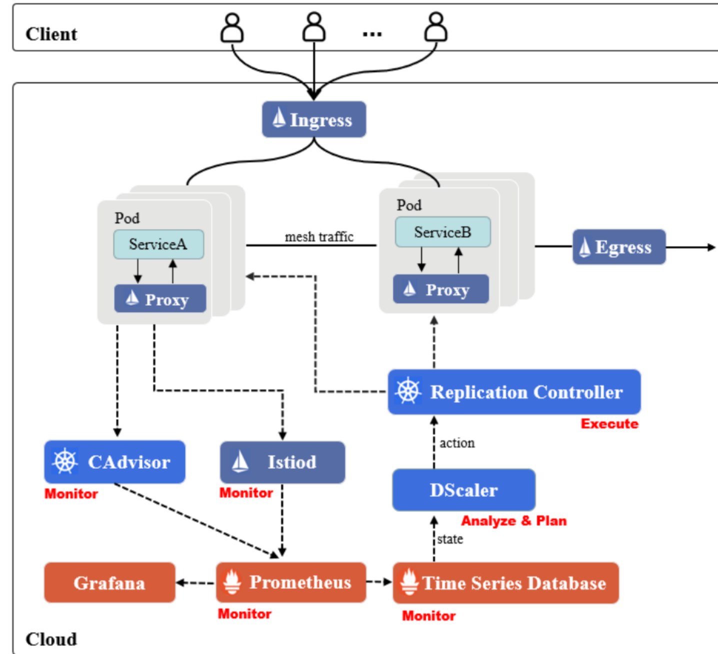


Figure 3.16 – Proposed system architecture.(from [107]) (Copyright © 2022 IEEE)

to maintain an acceptable response time while maximizing average CPU utilization. The authors tested their solution against HPA in a Kubernetes environment with two different workload patterns, achieving improved response time, higher average CPU usage, and a lower pod count as illustrated in the Figure 3.16.

One of the advantages of using RL in autoscaling is that it facilitates online training, eliminating the need for data collection, offline training, and regular model updating. However, this method has its limitations, particularly in the action space. If CPU demand significantly increases within a short period, the model cannot deploy more than two pods at a time, which can impact service response time. Increasing the action space would complicate the decision process, given the large state space. Additionally, the entire autoscaling process is handled by the DRL model, resulting in minimal transparency in the process.

In the paper [108], the authors identify the challenge of accurately selecting resource parameters under varying QoS requirements in standard autoscaling solutions. To address this, they propose a two-stage solution that dynamically adjusts resource parameters to maintain the performance of container-based cloud applications. The authors elucidate that each application's resource consumption and QoS requirements differ, necessitating an autoscaling solution capable of dynamic adjustment.

Graphic removed to respect copyright

Figure 3.17 – The system architecture of autoscaling framework. (from [109])

One of the most significant challenges in cloud deployments is the initial allocation of resources. This is typically accomplished by observing historical resource consumption data. However, in the absence of historical data, Kubernetes provides vertical autoscaling to recommend the necessary initial resources. In their proposed solution, the first stage utilizes this generic vertical scaling to determine the required amount of resources (CPU, memory limits). Subsequently, when the workload fluctuates, their solution relies on the second stage, which is a DRL-based solution, to horizontally scale pods.

The objective of this solution is to maximize resource utilization while minimizing response time. The ability to dynamically adapt to the current situation and reduce response time is a key advantage of the solution. The authors have tested this solution with different deep-learning agents. However, the state space, which consists of pod count, average CPU utilization, response time, minimum and maximum pod numbers, as well as the action space (scaling-up and down and no scaling), is large. This is similar to the issue in[107] and necessitates a longer online training time, which is a significant disadvantage.

In [109], the authors address the issue of extended training time for RL in cloud Horizontal Autoscaling by introducing a multi-agent approach. This approach aims to reduce the state space, thereby accelerating convergence. However, in this case, the authors employ multi-agents based on the type of VM, specifically spot instances. The authors categorize the types of VMs deployed in the cloud (e.g., based on application type, resource configurations, etc.) and deploy separate agents per type to manage autoscaling as illustrated in the Figure 3.17.

Each monitoring component observes the type of the VM, the number of VMs in the cloud, and the current workload. Given the continuous nature of these input data, the authors utilize DRL agents. The reward function is designed with consideration for operational cost, user payment, and compensation due to SLA violations. This approach covers both performance and cost, aiding in converging to a solution that can balance both aspects.

Furthermore, the authors implement a passive mechanism to train the DRL models, aiming to minimize the impact of the trial and error process of the DRL model. This is achieved by initiating on-demand VM capacity immediately if there is an interruption to the instance processing. Despite the authors' use of a multi-agent method per type of VMs to reduce convergence time on the DRL model, the state space to be covered remains large, which could potentially extend the training model.

In the contemporary landscape of cloud computing, multiple layers such as infrastructure, virtualization, and application are prevalent. The authors of the paper [110] have identified a research gap in the development of autoscaling solutions for this architecture. They note that most existing solutions do not incorporate visibility into this multi-layer cloud architecture, and most autoscaling solutions operate in isolation, without a comprehensive understanding of the dependencies that influence them.

The authors emphasize the necessity for an autoscaling mechanism to be cognizant of various multi layer correlations for accurate scaling. These include vertical correlations (e.g., container/VM level dependencies), horizontal correlations (e.g., topology and service architecture dependencies), and time correlations (e.g., time dependencies in component behavior).

The authors further underscore that an autoscaling solution that handles a large number of features across different layers can benefit from the use of AI models to extract complex dependencies. However, they caution that certain AI models, such as neural networks, can act as a "black box" in decision-making, leading to a lack of transparency and an inability to understand the root cause of issues.

To address these challenges, the authors propose a novel multi-stage autoscaling mechanism that incorporates multiple AI models in decision-making. In their proposed solution, the first stage involves monitoring and collecting data from each layer, which is then fed into a Dynamic Bayesian Network trained to estimate causal dependencies among features. This module aids in reducing the number of features used in the subsequent stage.

In the next stage, a pre-trained LSTM model predicts the future state of the cloud

environment, making the solution proactive. With the knowledge of SLAs, a Markov Chain Monte Carlo (MCMC) model in the subsequent stage maps appropriate events to adjacent states. A Q-learning model then assigns necessary actions based on the states provided by the MCMC model. This MCMC model enables the RL model to limit the state-action pairs, simplifying the decision-making process.

A key advantage of this solution is its ability to understand correlations between different layers, which is beneficial for accurate scaling decisions. For instance, application performance could suffer due to a bottleneck in another layer, not necessarily due to resource starvation. In such cases, isolated autoscaling solutions that make scaling decisions based on performance could lead to resource overprovisioning.

However, the implementation of this novel system is complex. The creation of multiple AI models, training of multiple models, and large-scale data collection can be cost-prohibitive for most cloud applications. Due to potential errors in AI model outputs, aggregating results from multiple models can amplify these errors, potentially leading to suboptimal scaling decisions.

As indicated in preceding studies, RL offers a fully automated process for decision-making analysis and self-learning. However, an increased state-action space, also known as the "state-action explosion" problem, is a well-known issue in RL. This problem is also evident in the application of RL in cloud autoscaling, where continuous state and action spaces, such as differences in resource utilization, number of replicas, response time, and actions like adding, removing, and determining the number of replicas, can expand these spaces.

This expansion can prolong the time required for model convergence. A significant issue is that during the training period, RL can be reactive, implying a trial-and-error approach. Consequently, if the convergence time is extensive, the cost will be higher during the online learning period due to unoptimized decisions. Additionally, the absence of a proper reward function and inadequate training could lead to suboptimal convergence, indicating potential model overfitting.

Furthermore, RL operates as a "black box," offering no explanation for its decisions, which complicates troubleshooting or pinpointing the root cause. However, some state-of-the-art papers have utilized clustering or quantification of the state space and action space to convert it into a finite problem, thereby reducing the time to convergence. However, this approach can lead to sub-optimal results due to reduced information in the clustering or quantification.

3.6 Prediction based autoscaling

In the domain of serverless computing, characterized by ephemeral workloads, the authors in [111], elucidate that performance can be compromised when resource allocation is based on predefined threshold based autoscaling. This highlights the necessity for dynamic autoscaling processes capable of accurately estimating the requisite resources.

The authors acknowledge that, while long-running applications can leverage machine learning models dependent on historical data, the serverless context necessitates a lightweight resource usage prediction mechanism for rightsizing serverless containers. As an alternative to Kubernetes VPA, the authors propose vertical autoscaling using Simple Moving Average (SMA) and Exponential Moving Average (EMA) methodologies to predict CPU usage and dynamically modify pod resource configurations.

These methodologies are particularly beneficial in the context of cold starts, where there is an absence or insufficiency of historical data for employing machine learning methods to predict CPU usage. Despite their results demonstrating that the proposed methods can anticipate the expected CPU load in advance, it does not always ensure the maintenance of acceptable performance levels due to CPU throttling.

CPU throttling is a Linux kernel mechanism utilized by Kubernetes to enforce pod CPU limits. Throttling is triggered when CPU usage reaches the defined limit, leading to a degradation in application performance. Therefore, to maintain acceptable QoS levels, scaling actions must be executed at the appropriate time to prevent the impact of CPU throttling. However, this particular factor is not addressed in proposed methodology. Furthermore, the authors contend that reducing the monitoring time interval enables more precise detection of CPU usage fluctuations. However, frequent retrieval of new metrics data increases monitoring cost, impacting the cost-QoS trade-off.

The paper [112] addresses the challenge of achieving an optimal trade-off between cost and QoS in cloud environments by leveraging widely used threshold-based autoscaling solutions, which are incapable of handling complex workload patterns.

The authors state the necessity of prediction based autoscaling to estimate the required resources and optimize the cost based on the pricing models used by public cloud providers. To address this issue they propose a predictive autoscaling solution that allocates resources proactively, preventing QoS degradation due to resource starvation.

The proposed architecture as illustrated in Figure 3.18 is designed for horizontal scaling of containers in a Kubernetes environment. It monitors the CPU usage of pods deployed in

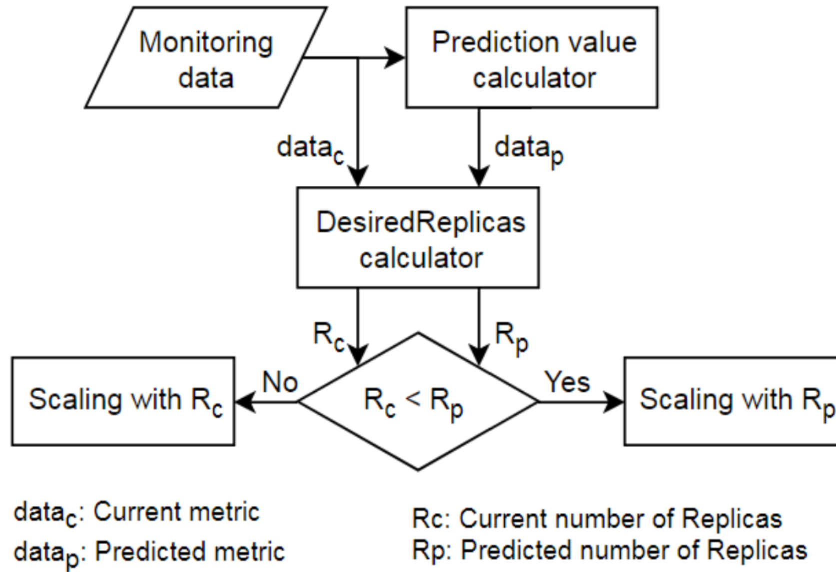


Figure 3.18 – Proposed autoscaler system data flow. (from [112]) (Copyright © 2019 IEEE)

the Kubernetes cluster and employs a double exponential smoothing technique to predict future CPU usage of the replicas. The autoscaler module then calculates the desired number of replicas based on a predefined CPU usage threshold. This calculated desired number of pods is compared with the current number of replicas to determine whether scaling-up is necessary.

However, their prediction method is not used when scaling-down replicas. Authors state that prediction of such situations could potentially indicate a service failure and mislead the autoscaling solution. Therefore, they delegate the scaling-down process to the Kubernetes HPA. The authors tested their solution and the HPA using the same traffic pattern. Their solution demonstrated better scaling performance compared to the HPA in terms of maintaining response time. Unlike machine learning or deep learning models, the double exponential smoothing technique does not require extensive training, making the prediction process lightweight. However, this prediction method can struggle with more complex patterns, such as nonlinear trends and seasonality. Additionally, the authors note that allowing the default HPA to handle scaling-down does not improve the cost-QoS trade-off, especially when it uses a cool-down period to avoid oscillations.

In the research [113], the authors identified a limitation in Kubernetes' vertical scaling capabilities, especially when dealing with periodic workload fluctuations. To address this, they proposed a predictive, autoscaling strategy that adjusts the CPU resource request to prevent CPU throttling and under-provisioning resources.

Figure 3.19 – Requested CPU for each scaling approach. (from [113])

The authors conducted experiments using two time-series prediction models to forecast the CPU usage of the target pod. They utilized the Kubernetes metric API to collect CPU usage data at 10-minute intervals and input these values into two models: Holt-Winters (HW) and Long Short-Term Memory (LSTM). With a historical window of two season lengths, both models were able to predict 24 steps into the future, with the primary consideration being weekly patterns.

Upon predicting the CPU usage using the best performing model, two static thresholds were established to determine whether to scale up or down. To mitigate sub-optimal scaling decisions due to prediction error, a buffer zone of 120 CPU milicores was implemented. Each scaling action determined the amount of CPU milicore based on the prediction plus the buffer. Furthermore, they employed a cool-down period of 18 time steps to prevent oscillations.

Predicting the required resources for longer horizons enables the scaling solution to determine the necessary number of resources in advance, thereby reducing resource wastage. However, the data extraction period is too extensive to prevent CPU throttling if the workload is volatile. With this method, as suggested, the selected parameters are more suitable for weekly adjustments. As depicted in the Figure 3.19 , the CPU usage in certain situations can exceed the allocated capacity and could lead to CPU throttling. However, the authors do not investigate the impact of CPU throttling on the response time with their solution, which could be a valuable indicator of whether the solution minimizes SLA

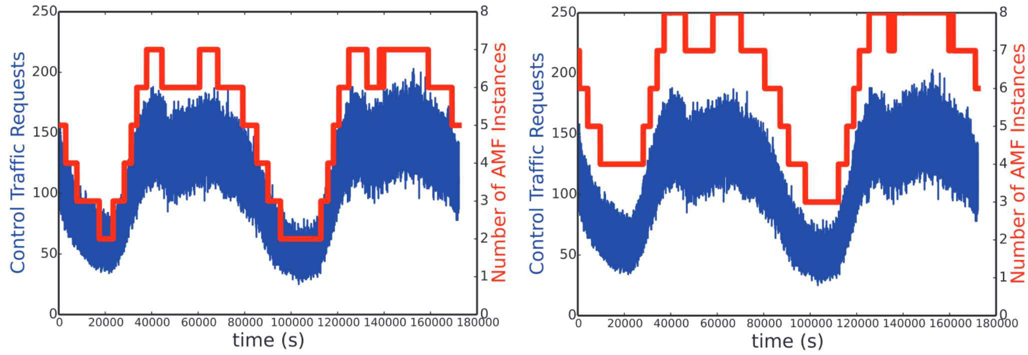
violations.

In the paper [78], the authors argue that the magnitude and volatility of the traffic expected in 5G networks cannot be managed effectively by traditional threshold-based scaling. One of the primary reasons for this is the potential impact of startup delay post-triggering on the performance of VNFs. The AMF, serving as the access point for the 5G CN, is identified by the authors as a potential performance bottleneck during periods of heavy traffic, necessitating dynamic autoscaling. To address this, the authors propose an autoscaling solution based on traffic load prediction for managing the scaling of AMF NF. They emphasize the utility of service-level metrics, such as the number of incoming users, in determining whether VNFs are overloaded. Deep Neural Networks (DNNs) and Recurrent Neural Networks (RNNs) are employed by the authors to predict the requisite number of AMF instances, with the ETSI NFV architecture being leveraged to deploy their solution in a cloud environment. The authors utilize a public telecom dataset, containing temporal traffic patterns of user connections, to train their forecasting models. These models are designed to analyze past values of incoming user connections and predict the expected number of AMF instances.

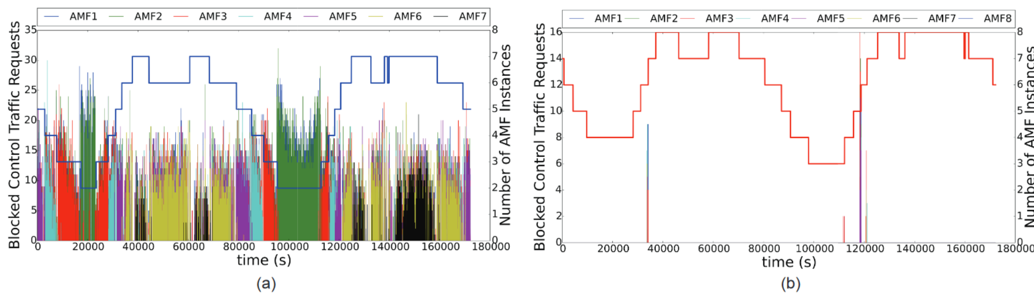
For this purpose, the original dataset is manually categorized into ten groups, with the required number of AMF instances being assigned to each. Following the training and testing of both forecasting models, the authors select the model demonstrating superior performance for their autoscaling experiments. The authors conduct comparative experiments between threshold-based and their autoscaling solutions to determine which approach optimally utilizes AMF instances and minimizes blocked requests. As illustrated in their Figure 3.20, their solution proactively aligns incoming traffic with AMF instances, thereby reducing blocking requests.

However, it is important to note that while their solution can match incoming traffic with AMF instances, it does not ensure that these instances utilize the maximum capacity of the allocated resources. Furthermore, the authors' manual categorization of the training data to assign AMF instances may not be optimal for maximizing resource utilization of the VNFs, potentially leading to resource wastage and suboptimal scaling.

The study [114] propose an autoscaling solution for a comprehensive 5G network. This solution employs two distinct scaling strategies: horizontal and vertical. These strategies are applied to three key components of the network: the RAN, the CP and UP in the CN. The authors underscore the significance of dynamically adapting resources to meet demand on both the RAN and CNs. Dynamically adapting resources is particularly crucial



(a) Evolution of AMF instances number with control traffic requests: i. threshold-based solution; ii. RNN-based solution.)



(b) Number of blocked control traffic requests compared to the number of AMF instances: i. threshold-based solution; ii. RNN based solution.

Figure 3.20 – Experimental results. (from [78]) (Copyright © 2018 IEEE)

to enhance energy efficiency during peak usage hours, as well as during periods of low usage, such as night time.

In this solution, the selection of the scaling approach and monitoring metric is contingent upon the pressure conditions within each network component. For instance, in RAN, alleviating the pressure to decode incoming traffic at eNodeBs can be achieved through vertical autoscaling. This not only conserves energy but also optimizes resource utilization by maintaining lower unused resources during periods of low usage. This periods can be identified by monitoring the UP bytes transmitted over the Packet Data Convergence Protocol (PDCP). Similarly, both the UP and CP can benefit from horizontal scaling to prevent resource congestion. The scaling decisions for the UP can be determined by the traffic sent to the SWG/PGW over the S1 interface. Conversely, the CP can be scaled based on the traffic sent over the S1-MME interface, as depicted in the accompanying Figure 3.21.

The authors argue that while these metrics offer insights into the network's current

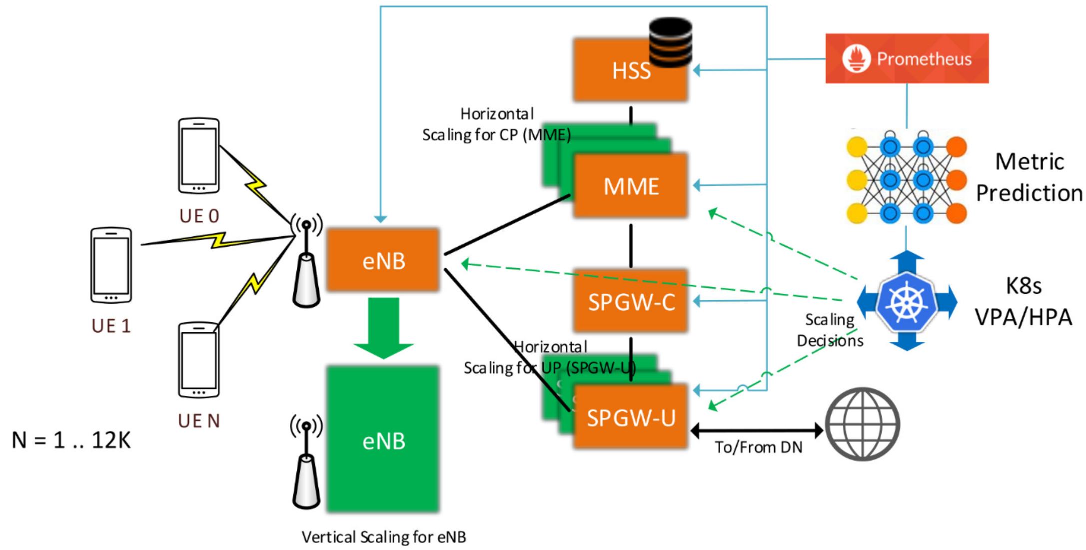


Figure 3.21 – System architecture for the under-study cloud-native network. (from [114]) (Copyright © 2021 IEEE)

state, real-time prediction could be advantageous for network self-organization. To render the autoscaling proactive, the authors employ a LSTM model. This model is trained and tested to predict using a publicly available telecom dataset, with the Mean Absolute Error (MAE) used to gauge the accuracy of the predictions. Subsequently, the authors determine the scaling decision by applying rules on the predicted metric. In this scenario, the eNodeB either increases or decreases resource allocation based on predefined thresholds. For the UP and CP, the number of replicas to be added or removed is determined by predefined thresholds. This solution takes into account the relevant scaling approach and metrics to ascertain the need to scale, customized to each component. This enables the network to optimize its resources more adaptively and proactively.

However, the solution's execution of scaling decisions is based on predefined static thresholds, which may not always yield optimal resource allocations. Moreover, in the case of eNodeB vertical scaling, a predefined amount of resources is added or removed, rather than customizing to the actual need, leading to suboptimal resource allocation. Additionally, in the horizontal scaling of the CN, the authors scale based on the number of connections, without information about the resource consumption of each VNF, which also results in suboptimal scaling decisions.

In the referenced work [115], the authors address the issue of QoS degradation due to reactive scaling which can be reduced by proactive scaling. They propose an autoscal-

Graphic removed to respect copyright

Figure 3.22 – Prediction-based autoscaling system architecture. (from [115])

ing solution that leverages a MAPE closed-loop architecture as illustrated in the Figure 3.22. This solution predicts the incoming requests using Bi-directional Long Short-Term Memory (Bi-LSTM) models.

Unlike conventional LSTM models, which are unidirectional and only consider forward learning in a given sequence, Bi-LSTM models incorporate both forward and backward learning processes. The authors argue that this bi-directional learning process is more suitable for traffic prediction, as Bi-LSTM can learn information about the input sequence from both ends. The authors present a forecasting model that predicts incoming traffic and calculates the required number of replicas based on a constant value, which represents the number of requests that can be processed by a single pod. Experimental results on multiple public datasets demonstrate the greater performance of Bi-LSTM compared to conventional LSTM and AutoRegressive Integrated Moving Average (ARIMA) models. The results indicate that the proposed solution can accurately predict the traffic, proactively scale, and use fewer replicas compared to the Kubernetes HPA.

However, this solution calculates the number of pods based on the maximum workload that can be processed by a single pod. Even when the workload matches the required pod number, it is challenging to determine when scale up is necessary without resource consumption data. There are scenarios where the workload is not high enough to trigger

scaling but sufficient to activate CPU throttling, which can impact the service response time. These scenarios are not visible if the scaling decisions rely solely on incoming traffic data.

Prediction-based autoscaling mechanisms provide a proactive approach to cloud resource management. By leveraging demand forecasting and executing preemptive scaling actions, these methods can minimize the inherent delays associated with scaling operations and reduce oscillations in resource allocation, a common issue in reactive scaling strategies. Once relevant metrics have been predicted, determining the precise timing for scaling actions based on predicted demand becomes critical. Correctly timing scaling operations is essential for optimizing cost-efficiency while ensuring the required QoS is maintained. Moreover, most prediction-based autoscaling techniques depend on large volumes of historical data for offline model training,which can be resource-intensive.

3.7 Summary

This chapter explores the various mechanisms used for cloud resource autoscaling, focusing on their impact on operational efficiency and the trade-off between cost and QoS. Autoscaling methods are designed to minimize resource overprovisioning to reduce costs while avoiding underprovisioning, which could lead to degraded QoS and SLA violations. However, selecting or designing an autoscaling solution that fits specific cloud-native applications is a complex challenge. The performance of an autoscaling mechanism is influenced by several factors, including the application's architecture, whether it is stateful or stateless) its ability to adapt to fluctuating workloads, the types of metrics used to make scaling decisions, the handling of decision oscillation, whether the decision-making is reactive or proactive, and the method of scaling (vertical or horizontal). These factors significantly influence how well an autoscaler manages the cost-QoS balance in cloud environments.

In examining the literature, the chapter first considers rule-based autoscalers, which are simple to implement and widely used. They perform adequately for most cloud applications, but their reactive nature makes them less effective in handling fluctuating workloads. Queuing theory-based autoscaling, on the other hand, dynamically adjusts to varying workloads, offering better adaptability than rule-based approaches. Despite this, it remains reactive and faces similar challenges, with the added complexity of being harder to implement in large-scale architectures and potentially computationally intensive.

Control theory-based autoscalers, such as those using PID control and fuzzy logic, are more adaptable to workload changes and can allocate resources more precisely, thus improving cost efficiency. However, these methods are less suited to modern multi-tier application architectures and are still reactive in nature. RL-based autoscalers represent a more advanced approach, utilizing self-learning and decision-making processes to understand complex workload and resource usage patterns. These methods can handle intricate application architectures, but they come with drawbacks such as long learning times and state-action explosion due to the large number of parameters that need to be controlled. Additionally, RL-based solutions suffer from limited explainability, making it difficult to interpret and troubleshoot their autoscaling decisions.

Prediction-based autoscaling takes a proactive approach, predicting workload and resource consumption behaviors and allocating resources accordingly. This method can be customized for specific application components or tiers, providing a high level of adaptability to balance cost and QoS. However, it requires a significant amount of historical data for training, and the training process can be computationally demanding.

Overall, the literature reveals that there is no universal autoscaling solution that fits all requirements. Each method has its own advantages and limitations, and the choice of autoscaler must be tailored to the specific needs of the application to achieve an optimal cost-QoS balance. The next chapter delves into the autoscaling requirements for performance-sensitive applications, such as 5G networks, and proposes a new approach to balancing the cost-QoS trade-off in cloud environments.

CHAPTER 4

RESOURCE USAGE FORECASTING FOR CNFS IN KUBERNETES ENVIRONMENT

4.1 Introduction

Adopting a cloud-native paradigm within 5G networks introduces novel challenges in optimizing operational costs for CSPs. The increasing number of users, devices, and diverse use cases necessitating tailored network configurations significantly amplifies the complexity of this issue, surpassing that encountered in traditional cloud-based IT services. As highlighted in the Introduction chapter, mitigating cloud resource wastage—a primary driver of elevated cloud expenditures—can be effectively addressed through dynamic resource allocation and automation. By understanding resource requirements and precisely allocating resources in a timely manner, it is feasible to circumvent both over-provisioning and under-provisioning, thereby achieving an better cost-QoS balance.

This chapter propose a novel proactive autoscaling solution for CNFs deployed within Kubernetes environments. The core objective of this solution is to forecast the anticipated resource utilization of CNFs and dynamically scale CNF resources to sustain an optimal cost-QoS balance. The initial phase of the proposed solution concentrates on resource usage forecasting. This involves an examination of the limitations inherent in existing Kubernetes autoscaling solutions and an analysis of the resource usage profiles of 5G CNFs to discern their consumption patterns. Subsequently, the correlation between incoming workloads and resource usage is investigated. By leveraging deep learning-based forecasting methodologies, the forecasting accuracy of CNF resource usage is enhanced, providing intelligent analytics for the autoscaling solution. Furthermore, an exhaustive analysis of the empirical results from the initial forecasting phase of the proposed solution is presented. These findings provide valuable insights into the performance and efficacy of the approach, laying the groundwork for subsequent stages of autoscaling solution.

4.2 Research challenge

In the domain of cloud computing, Kubernetes serves as a container management and orchestration platform, providing native support for horizontal autoscaling of container deployments. This autoscaling mechanism, detailed in Section 2.4.3, is threshold-based and inherently reactive. Upon reaching predefined thresholds of selected performance metrics, Kubernetes HPA dynamically scale the number of replicas within the cluster to maintain QoS levels and minimize resource wastage.

However, in the context of 5G networks, characterized by high traffic volume and diverse traffic patterns, such a reactive scaling mechanism may not optimally balance the cost-QoS trade-off for several reasons. Firstly, there is a temporal delay between the initiation of the scaling-up process and the operational readiness of new replicas [116]. This delay is attributable to several factors, including decision-making overhead in the Kubernetes control plane, CNF image availability, and software dependencies. Consequently, this delay is also contingent on the CNFs within the pod. During this period, existing replicas are subjected to heavy workloads under resource constraints, potentially degrading QoS. This issue is particularly pronounced when the scaling-up threshold is set high to maximize pod resource utilization, which helps reduce resource wastage. To circumvent this issue, scaling-up thresholds can be set significantly lower, but this may lead to premature scaling without exceeding the resource capacity of existing replicas, thereby increasing resource wastage. Manually selecting an appropriate threshold to balance the cost-QoS trade-off is thus a challenging task.

Secondly, in scenarios with volatile traffic behavior, threshold-based scaling can result in oscillations in scaling decisions. Frequent fluctuations in incoming workload prompt the threshold-based mechanism to continuously adjust the number of replicas, leading to frequent scaling-up and down of replicas. Given the aforementioned delay in Kubernetes HPA, this can result in QoS degradation of the CNFs. Although Kubernetes offers a cool-down period to mitigate the impact of such oscillations, restricting the HPA from making rapid replica changes for a specified duration, this limitation hinders the autoscaler's ability to make optimal scaling decisions, affecting the cost-QoS trade-off.

To optimize dynamic scaling decisions in an autoscaler, thereby minimizing QoS degradation, SLA violations, and operational costs, it is essential to understand the startup delay of CNFs and anticipate future resource usage behavior. Proactively analyzing resource usage and making precise scaling decisions thus provides a more effective solution

Table 4.1 – Cluster resource specifications. (from [4]) (Copyright © 2022 IEEE)

	CPU	Memory	Storage
Master Node	12 cores	8GB	240GB
Worker Node 1	8 cores	16GB	225GB
Worker Node 2	8 cores	16GB	240GB

in such scenarios. Accurately predicting the resource usage of CNFs requires consideration of several factors. Firstly, it is essential to accurately identify the pivotal resource for dynamic scaling decisions. Making scaling decisions based on incorrect resource usage metrics can significantly disrupt the cost-QoS trade-off. Secondly, understanding how the selected resource usage metric responds to incoming workloads is crucial. Given that different CNFs perform varied tasks, the relationship between workload and resource usage can differ significantly depending on the specific CNF. Therefore, investigating these factors is imperative for accurately forecasting future resource usage to enable proactive autoscaling.

4.2.1 Resource usage profiles of 5GC network functions

To devise a custom proactive autoscaling solution for cloud-native 5G network functions, a thorough investigation of their resource usage profiles is essential for designing a resource usage forecasting model. This process involves identifying critical resources for application performance and scaling needs, as the impact of resource scarcity on the network function's QoS profile varies depending on the type of resource. For example, insufficient CPU availability can result in CPU throttling, which adversely affects service response times until the autoscaler increases resources. Conversely, a lack of memory may trigger an Out-Of-Memory (OOM) condition, leading to the termination of the network function and resulting in service unavailability.

For this purpose, an experiment was conducted by deploying 5G CN CP CNFs in a Kubernetes cluster. The objective of the experiment is to inject UE traffic into the 5G CN CNFs, thereby inducing resource consumption due to the processing of incoming traffic. This configuration enables the analysis of the individual resource consumption profiles of each CNF. The Kubernetes testbed was deployed in a bare-metal configuration consisting of three physical nodes: one master node and two worker nodes. Each node operated on Ubuntu 20.04.4 and utilized Docker as the container runtime. Detailed hardware specifications for each node are provided in the accompanying Table 4.1.

For the 5G CN, the open-source SD-Core version 1.3 [117], a project from the Open Network Foundation (ONF) [118], was utilized. It is noteworthy that each network function was deployed without any resource limitations, thereby allowing unrestricted resource usage. The resource pool within the cluster was sufficiently large, eliminating the need for network functions to compete for resources. The 5G CN CP NFs focused on this experiment:

— Access and Mobility Management Function (AMF)
— Authentication Server Function (AUSF)
— Unified Data Repository (UDR)
— Network Repository Function (NRF)
— Unified Data Management (UDM)
— Policy Control Function (PCF)
— Session Management Function (SMF)

In addition to the 3GPP standard NFs, a SCTP load balancer was also deployed. This feature, introduced in SD-Core version 1.3, facilitates the handling of multiple AMF instances. To monitor the resource consumption of the 5G CN NFs, Prometheus [119] monitoring service was deployed. Additionally, Grafana [120] data visualization software tool was also deployed.

For UE traffic generation, an open-source traffic simulator named UERANSIM version 3.2.6 [121] was employed, which includes both UE and gNodeB functionalities. The traffic simulator was located outside the Kubernetes cluster, and the gNodeB connected with the AMF via the SCTP protocol. The complete testbed is depicted in the Figure 4.1. As outlined in Section 2.1.2, 5G CN NFs perform various tasks; however, for simplicity, this experiment focused solely on the UE registration and de-registration processes. The 3GPP standard call flow for UE registration and de-registration is detailed in the corresponding specification [122].

The UE registration and de-registration requests were generated based on the following three scenarios as a continuous stream without any sleep time between scenarios.

1. 200 UEs registrations at 1 UE per second → Sleep for 100 seconds → De-register all the UEs at 1 UE per second.
2. 400 UEs registrations at 2 UE per second → Sleep for 100 seconds → De-register all the UEs at 2 UE per second.
3. 800 UEs registrations at 4 UE per second → Sleep for 100 seconds → De-register all the UEs at 4 UE per second.

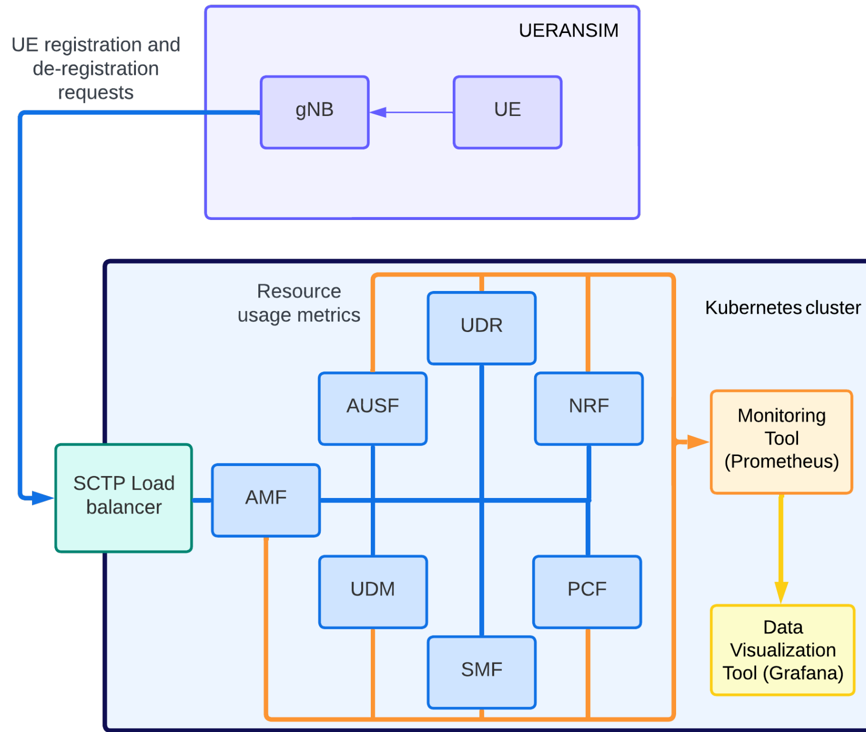
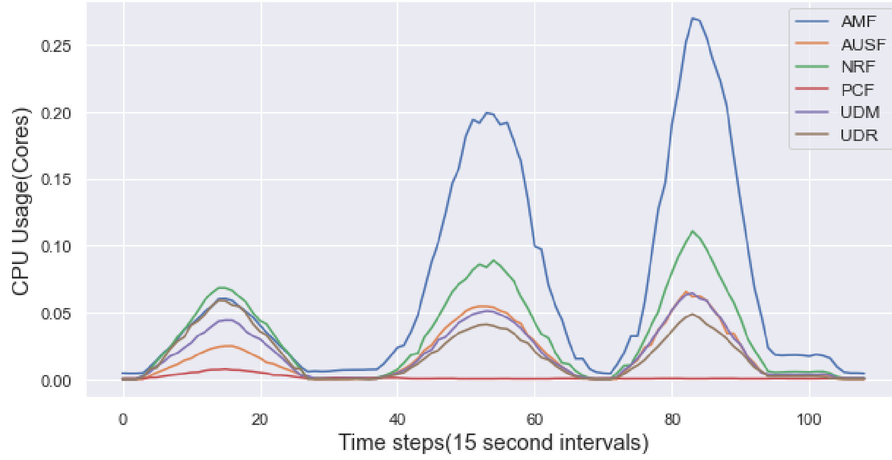


Figure 4.1 – Kubernetes testbed with 5G CN.

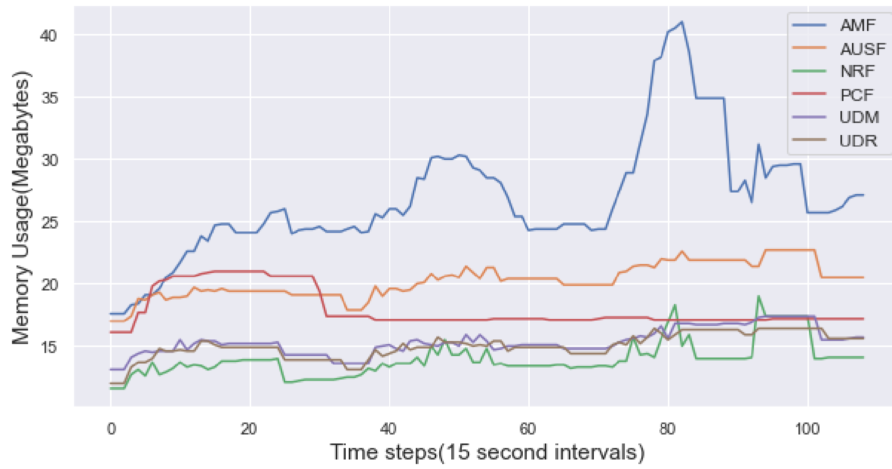
Upon completion of the traffic generation process, the CPU and memory usage profiles of all 5GC NFs were collected from the Prometheus database. These resource usage profiles were gathered and stored as time series data, with data points collected at 15-second intervals by Prometheus. The captured data is illustrated in the Figure 4.2. It is noteworthy that during the experiment, the SMF exhibited no significant resource utilization. As a result, its resource usage profiles were excluded from the analysis.

The Figure 4.2a illustrates the CPU usage profiles of all NFs during the experiment. Initially, before injecting traffic, each NF exhibited different CPU consumption levels, but all were negligible, indicating the baseline resource usage in their idle states, as detailed in Section 2.4.2.

A noteworthy observation is the CPU usage profile of the PCF NF. In the first scenario, there is a slight increase in CPU consumption, but it subsequently returns to idle values for the remainder of the experiment, irrespective of the number of UE connections. In contrast, the CPU consumption of other NFs varies with the incoming workload. For instance, while there are some differences in CPU consumption among all other NFs in



(a) CPU usage profiles of 5G CN.



(b) Memory usage profiles of 5G CN.

Figure 4.2 – 5G CN resource usage profiles.

scenario 1, the CPU usage for AUSF, UDM, and UDR in scenarios 2 and 3 remains nearly identical, despite the doubling of UE connections and traffic rate between scenarios.

AMF and NRF exhibit significant changes in CPU usage corresponding to the incoming workload. However, even though the number of connections and the connection rate doubles with each scenario, the CPU usage does not increase proportionally. The AMF, being the gateway to the 5G CN and responsible for UE registrations and de-registrations, shows the most substantial changes in CPU usage across the scenarios. After the traffic injection ceases, all NF CPU usages return to their idle states, reducing consumption.

The Figure 4.2b illustrates the memory usage profiles of all NFs during the experiment. In contrast to CPU usage, initial memory usage is substantial and varies for each NF. The

AMF exhibits notable changes in memory usage corresponding to variations in workload. However, these workload increments and decrements are not consistently reflected across all scenarios, making the detection of workload changes from the AMF memory usage profile ambiguous. Despite variations in the number of connections and traffic rates across scenarios, the memory usage profiles of the other NFs do not exhibit significant changes in response to varying workloads compared to initial resource consumption.

A significant observation is that, following the experiment, the memory usage of all NFs does not revert to their initial values. In this version of the SD Core, all NFs operate in a stateless manner. This design implies that for any given request, NFs—including but not limited to AMF and SMF, which were strictly stateful in previous versions—retrieve the necessary data from a repository solely for request processing. After processing, the data is stored back in the repository, thus not remaining in memory. Therefore, the memory usage profiles of the NFs are expected to revert to idle usage values after the experiment. The failure of memory usage profiles to return to these values could be attributed to factors such as the efficiency of the garbage collection process, the programming language used, and compiler optimizations.

Cloud-based applications are typically classified as CPU-intensive, memory-intensive, or a combination of both [79]. In this study involving the 5G CN, NFs such as AUSF, UDR, NRF, and UDM displayed significant variations in CPU usage relative to memory usage profiles under incoming workloads, thereby categorizing them as CPU-intensive. In contrast, the AMF demonstrated consistent behavior in both CPU and memory usage under varying workloads, indicating characteristics of both CPU and memory-intensive operations.

Based on the analysis of resource usage profiles across all NFs under varying workload patterns, it is concluded that CPU should be selected as the critical resource for autoscaling the 5G CN in this research. When the autoscaler horizontally scales pods based on CPU usage, it also increases memory resources due to pod replication, thereby fulfilling memory requirements, provided that memory requests and limits are appropriately configured. This approach addresses the issue of selecting CPU as the crucial resource for autoscaling, even for NFs such as AMF, which are both CPU and memory-intensive. The findings of this experiment, indicating that NFs are primarily CPU-intensive rather than memory-intensive, are also confirmed by the study [123] that utilized Free5GC as their 5G CN within a Kubernetes environment.

However, certain NFs, such as the NWDAF, which is responsible for data collection

and statistical model learning processes within the 5G CN, require substantial memory resources, making them more likely to be memory-intensive. Nevertheless, this NF is not available in the SD Core version used in the experiment and is thus out of scope for this research study.

A limitation of the experiments is the exclusive use of UE registrations and de-registrations with only a few NFs. For more robust and conclusive results, future experiments should incorporate a broader range of 5G CN services, including other 3GPP-specified NFs, under real-world traffic conditions.

4.2.2 High level vs low level metrics

A pivotal consideration in the realm of autoscaling is the continuous monitoring of cloud-native applications. This involves the systematic collection of deployment status data, performance metrics, resource consumption information, etc.. Such data aggregation enables sophisticated analytics, thereby informing and optimizing scaling decisions. However, the operation of these monitoring services can incur significant costs [124]. A monitoring service may either be an additional subscription in a public cloud or an implementation within the cluster which necessitates allocation of additional resources for deployment, thereby both scenarios contributing to operational costs.

Moreover, an increase in the number of monitoring metrics can potentially overload the monitoring service, leading to increased resource consumption. Therefore, metric selection is a critical component of an autoscaling solution, presenting a trade-off between the cost and visibility offered by monitoring services. In essence, the cost of enhancing visibility through monitoring services should not outweigh the benefits derived from their use in autoscaling. Therefore, identifying the critical metrics to monitor, determining the appropriate frequency of data collection, and configuring other parameters that facilitate the autoscaling solution are essential to balancing the cost-QoS trade-off in cloud applications.

Modern telecom networks exhibit a diverse range of traffic patterns. Consequently, system-level metrics or low-level metrics [98], such as the resource consumption of CNFs, are insufficient to provide the necessary visibility for proactive resource allocation [78]. While some studies employ low-level metrics in their autoscaling solutions, others prefer application level metrics or high-level metrics, such as request rate, latency etc. obtained from third-party monitoring services [125] [126]. These studies argue that high-level metrics offer superior prediction visibility. However, relying solely on high-level metrics for

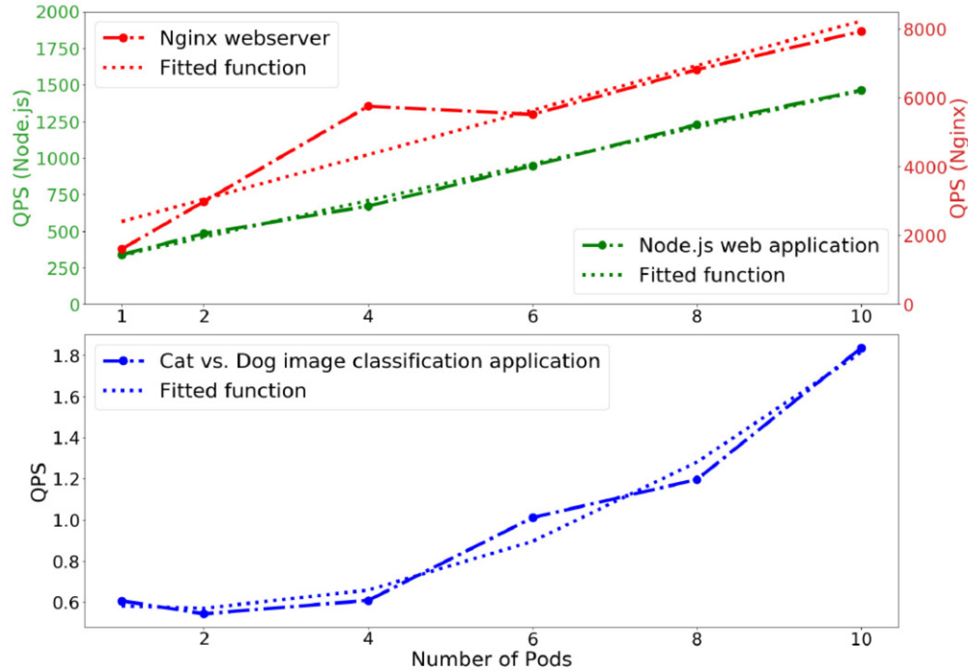


Figure 4.3 – Resource usage profile of different applications. (from [125]) (Copyright © 2021 IEEE)

prediction can present several challenges.

In the context of CNFs, the correlations between incoming workload and resource usage of CNFs vary [123]. Consequently, identical traffic patterns can result in different resource consumption patterns. Therefore, relying solely on high-level metrics, such as the rate of incoming requests, does not accurately reflect the actual resource consumption. This inadequacy hinders informed assessments of resource utilization and impedes effective scaling decisions.

To understand the varying correlations between incoming traffic and resource consumption for different CNFs, a study [125] examines resource utilization profiles across diverse applications by comparing resource usage for three distinct applications within a Kubernetes environment, correlated with increasing rates of incoming requests.

For this purpose, an NGINX web server, a Node.js web application, and an image classification application were deployed along with monitoring services. The experiment involves injecting increasing request rate and monitoring the number of Queries served Per Second (QPS) relative to the number of deployed pods. The collected data is illustrated in Figure 4.3. It is important to note that the dotted lines in each application's resource profile, labeled as "Fitted function," represent the authors' mathematical approximations

of the resource usage profiles and are not pertinent to this study. In the plot, the Node.js application demonstrates a linear relationship between the incoming request rate and pod usage. Similarly, the NGINX application exhibits an almost linear correlation. In contrast, the image classification application, which is markedly resource-intensive, presents a quadratic relationship between the request rate and pod count. Additionally, NGINX served the highest number of requests, followed by Node.js, with the image classification application serving the lowest number of requests despite utilizing a similar number of replicas. This experiment indicates a more complex interaction between the incoming request rate and resource usage, underscoring the importance of understanding this correlation for effective autoscaling decision-making.

However, this experiment does not elucidate the interaction between resource usage and incoming request rates in complex workload patterns among CNFs. To further investigate this impact on autoscaling decisions, an experiment was conducted using a web application deployed on the Kubernetes testbed described in Section 4.2.1. This web application consists of a single service that processes a complex trigonometric math function upon receiving a parameter via an HTTP request. By altering this parameter, the complexity of the function can be varied, thereby modifying the CPU usage and processing time (t_{base}) for each request. In this experiment, two t_{base} were selected to analyze two different CPU usage profiles for the same incoming workload pattern. The web application was containerized and deployed on a pod with a CPU request of 350 milicores and no CPU limit, enabling unrestricted CPU consumption. Once deployed, a predefined HTTP traffic pattern was injected into the web application for the two distinct t_{base} values in separate trials.

The first t_{base} value, denoted as $t_{base}(P1)$ was used to create CPU usage profile 1, while the second t_{base} value, denoted as $t_{base}(P2)$ was used to create CPU usage profile 2. Since $t_{base}(P1)$ is greater than $t_{base}(P2)$, CPU usage profile 1 is expected to exhibit higher CPU usage compared to CPU usage profile 2. Data collected from both trials, obtained through monitoring services, is presented in Figure 4.4.

In the collected data, the CPU usage profile 1, characterized by a higher t_{base} value, exhibits elevated CPU usage which reached a max of 50% in comparison to profile 2 which only maxed around 39% under identical traffic conditions.

A key distinction between the two CPU usage profiles is observed during the high CPU usage period, specifically between the 250 and 800 time points. During this interval, CPU usage profile 2 exhibits significant fluctuations in comparison to CPU usage profile 1.

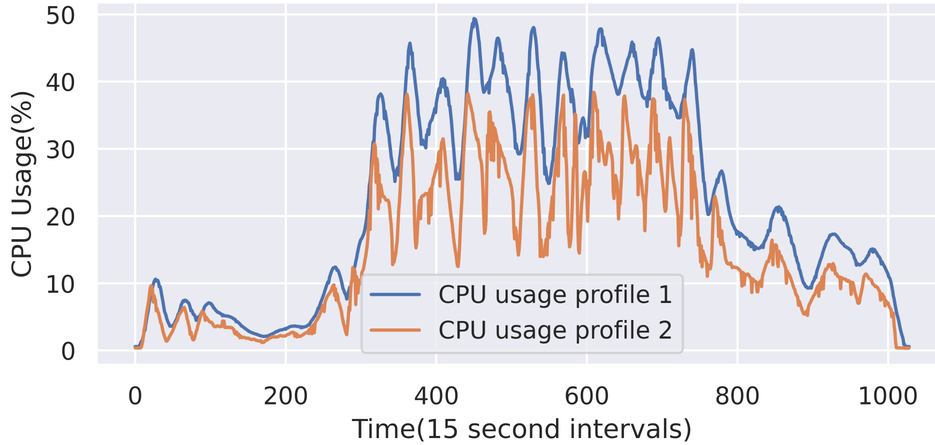


Figure 4.4 – CPU usage profiles of the web application under different t_{base} values. (from [4]) (Copyright © 2022 IEEE)

This discrepancy can be attributed to the higher t_{base} value of CPU usage profile 1, which results in longer processing times for incoming requests. Consequently, CPU usage profile 1 demonstrates reduced sensitivity to variations in the incoming workload. This indicates that, under identical workload patterns, CPU usage profiles influenced by differing request processing times do not exhibit proportional behavior.

Returning to the original assertion, the reliance on high-level metrics, such as incoming traffic data, fails to accurately represent the actual resource consumption of CNFs due to the complex correlations between workload and resource consumption, as demonstrated by these experiments. As explained in the Section 3.6, autoscaling methodologies that utilize predictive high-level metrics, such as request rate, to estimate required resources often overlook the dynamic interplay between workload and resource consumption.

For instance, Study [126] proposed a proactive autoscaling method utilizing a forecasting model to predict future traffic loads. This method then employs a static value representing the maximum traffic load manageable by a CNF to convert the predicted traffic load into the expected number of replicas required for scaling. From the experiments, it was evident that the correlations between incoming request rate and resource usage varied depending on the CNF type and is complex. Furthermore, this complexity increases with more intricate workload patterns. Therefore, using static values to determine the number of replicas from forecasted request rates overlooks these correlations, potentially leading to inefficient autoscaling decisions.

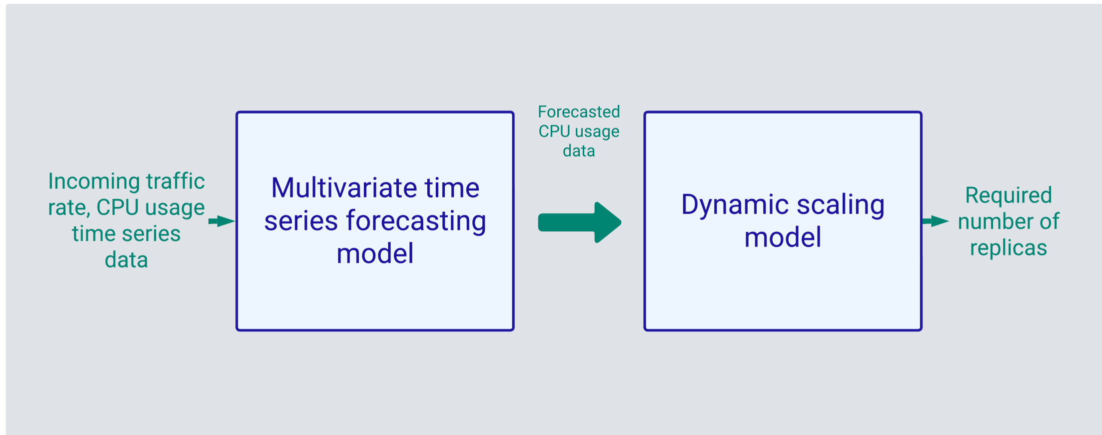


Figure 4.5 – High level architecture of the proposed solution.

4.3 Proposed system model

As an alternative solution for Kubernetes HPA and existing state-of-the-art autoscaling solutions, we propose an AI-assisted proactive scaling solution that balances the trade-off between operational costs and QoS for CNF deployments in a cloud-native environment. The cornerstone of this solution is its proactive nature, necessitating accurate scaling decisions based on anticipated pod resource usage behavior, precise replica calculation, and timely scaling actions. The solution unfolds in two stages: initially, a multivariate time series forecasting model predicts future resource consumption of pods within a single deployment; subsequently, a dynamic scaling algorithm makes scaling decisions informed by the forecasting model's predictions. The proposed high-level system architecture is illustrated in Figure 4.5.

To ensure the seamless automation of our scaling approach, the solution aligns with the closed-loop stages of the ETSI ZSM framework. The process begins with data collection, such as incoming traffic and CPU usage of replicas, using monitoring services within the cluster. This data is then processed to enable the forecasting model to predict the total future CPU usage of the replicas per service. The analytics derived from this process inform the dynamic scaling model's decisions, which adapt to the cluster's status. Finally, the Kubernetes control plane executes these decisions, managing the creation or termination of additional replicas. The interconnected stages of the proposed solution are depicted in Figure 4.6, demonstrating its integration with the closed-loop system.

In the context of developing an autoscaling solution for the 5G CN, high availability

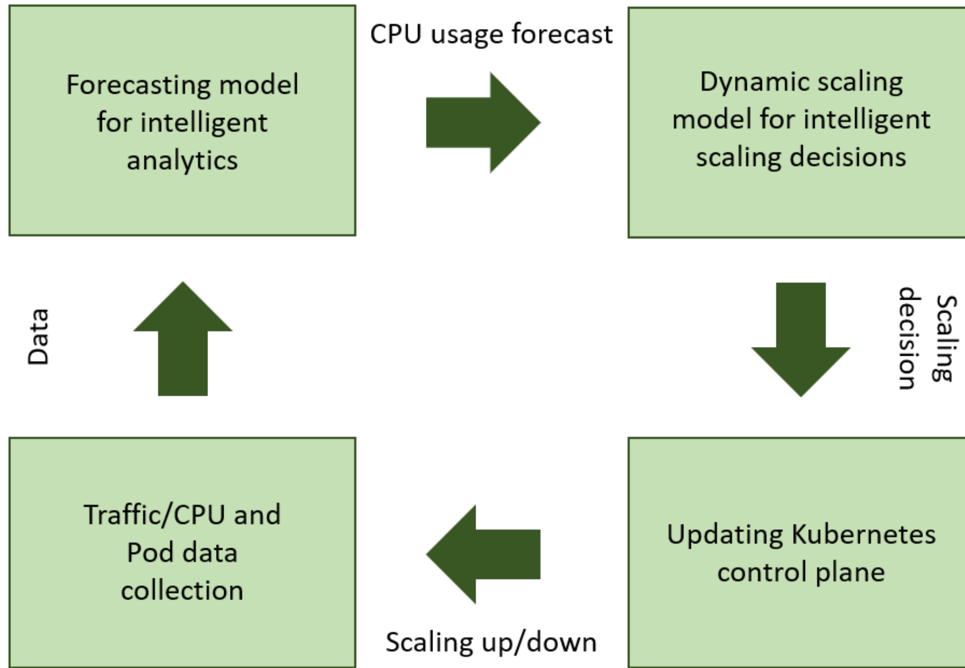


Figure 4.6 – Proposed solution with closed loop architecture. (from [5]) (Copyright © 2023 IEEE)

is a critical priority. Although both horizontal and vertical scaling have their respective advantages and disadvantages, horizontal scaling is preferred due to its capacity to add redundancy through replication, thereby mitigating the single point of failure inherent in vertical scaling. Consequently, our solution design emphasizes a horizontal scaling approach to enhance system resilience and ensure continuous availability. It is important to note that the design choices of the proposed solution are exclusively intended for stateless applications and are not directly applicable to stateful CNFs.

4.3.1 Resource usage forecasting

The primary goal of the initial phase is to accurately forecast the total resource usage for a specific Kubernetes service within the cluster. Leveraging insights from prior experimental analytics detailed in Section 4.2.2, the forecasting model is tailored to predict CPU usage in CNFs, where CPU is the critical resource for autoscaling decisions. It is essential to assimilate the complex correlation between CPU usage and the rate of incoming requests to enhance the precision of the forecasts. When designing a predictive model, multiple factors must be meticulously evaluated. These include the number of

features used in the forecasting, the computational complexity—especially if the model is to be employed in real-time prediction scenarios—and the model's proficiency in recognizing temporal patterns such as trends and seasonality. The current state-of-the-art encompasses a wide range of methodologies for time series forecasting. Therefore, it is crucial to judiciously select a model that aligns with the objectives.

4.3.2 Model selection

Since the proposed approach aims to incorporate both CPU usage data and the incoming request rate to enhance forecasting accuracy, both time series data must be utilized as features to forecast future CPU usage, necessitating the use of a multivariate model. Consequently, the chosen model must be proficient in handling the temporal relationships of multiple variables and capturing the correlations between features during forecasting.

A crucial aspect to consider is the computational complexity of the forecasting model. The objective is to implement a real-time forecasting model that delivers instantaneous analytics for an autoscaling solution. Models with elevated computational complexity may necessitate extended processing times, potentially compromising decision accuracy due to delays in analytics. Furthermore, high computational complexity results in increased resource utilization,which contradicts the primary objective of the autoscaling solution: minimizing operational costs.

Moreover, the selection of an appropriate time series forecasting model must be congruent with the inherent characteristics of the dataset. Datasets may exhibit singular or multiple underlying trends and seasonal components, necessitating the use of models specifically designed to address these features. Telecom datasets, in particular, frequently present complex patterns, including multiple trends and seasonalities across various time scales such as yearly, monthly, weekly, daily, or hourly intervals. Not all forecasting models are equipped to accommodate these diverse patterns effectively.

In the literature, the most commonly used statistical models for time series forecasting, such as AutoRegressive (AR), Moving Average (MA), and AutoRegressive Integrated Moving Average (ARIMA), are less complex [127] and relatively easy to implement. However, these models are not designed to forecast data with multiple trends and seasonalities. Extended versions of these models, such as Seasonal Auto Regressive Integrated Moving Average (SARIMA), Seasonal AutoRegressive Integrated Moving Average with Exogenous Regressors (SARIMAX), and HW models, are more complex compared to ARIMA and can handle multiple trends and seasons [128], but they are still designed to forecast

single-variable data. To address the issue of multivariate data, models like Vector AutoRegressive (VAR) [129] and Multivariate AutoRegressive Integrated Moving Average (MARIMA) [130] are specifically designed.

Deep learning models, such as Long Short-Term Memory (LSTM), Gated Recurrent Units (GRU), and Convolutional Neural Networks (CNN), are known for their capabilities in complex pattern recognition. These models are more sophisticated and resource-intensive during training, yet they can handle multiple trends and seasonal features [131] and can be multivariate. Additionally, forecasting with deep learning models shows significantly higher accuracy compared to conventional statistical models in some studies [132].

Compared to VAR and MARIMA models, deep learning models have the advantage of storing information about the dataset within their cell structures [129], which can beneficial for accurate forecasting with a short historical data window.

Given the advantages of deep learning models in time series forecasting and their high compatibility with the requirements, LSTM, GRU, and CNN models have been selected for initial phase of the autoscaling solution. It is essential to acknowledge that these models exhibit differences in their internal mechanisms, which could affect differently in their forecasting accuracy. Further investigation is required to elucidate these accuracy variations and to identify the best performing model for the autoscaling solution.

4.3.3 Data collection

Deep learning models are data-driven algorithms that necessitate historical data to learn hidden patterns, which are subsequently employed for forecasting, classification, and various other tasks. The accuracy of these models' outputs is highly dependent on the quality and quantity of the data utilized during the training phase.

A suitable dataset that meets these requirements and aligns with several objectives of this research is the open-source real-world dataset from Telecom Italia [133]. This dataset encompasses data on incoming calls, SMS, and internet traffic over a three-month period, providing detailed traffic patterns on a monthly, weekly, daily, and hourly basis within the Telecom Italia cellular network in the city of Milano.

Given that the primary objective of this study is to incorporate both CPU usage data and incoming traffic data (request rate) to enhance the forecasting of future CPU usage for CNFs, the dataset from Telecom Italia is unsuitable for the experiments for the following reasons. First, the dataset exclusively contains traffic load information (number

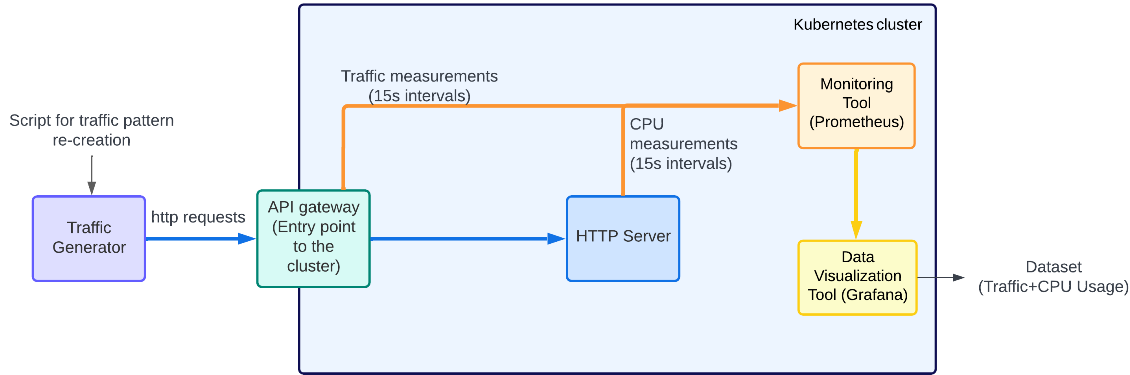


Figure 4.7 – Data collection testbed. (from [4] - modified) (Copyright © 2022 IEEE)

of requests) and lacks resource usage data for the associated network functions. Secondly, the data extraction rate, with a temporal granularity of 10 minutes, is too coarse for an autoscaling algorithm, where autoscaling solution must detect changes and the the replicas must become operational within seconds. Therefore, inspired by the hourly traffic patterns in the Telecom Italia dataset, a new dataset was created to align with the objectives of our proposed autoscaling solution.

To create the new dataset, a containerized web application was deployed in a pod within the Kubernetes testbed established in Section 4.2.1, as illustrated in Figure 4.7. The web application emulates a single-service, CPU-intensive CNF that uses REST APIs for communication. For monitoring and data collection from the cluster, the Prometheus monitoring tool was deployed. Prometheus scrapes data metrics from the cluster at 15-second intervals, storing them as time-series data. Additionally, to collect the HTTP request rate, a free version of the Kong API gateway [134] was deployed. The API gateway acts as an access point for external requests to the cluster and is integrated with Prometheus to scrape traffic data metrics. For data visualization, Grafana was deployed.

The methodology involves injecting the web application with HTTP requests according to a predefined traffic pattern, thereby causing the web application to process each incoming request and induce CPU consumption within the pod. To generate HTTP traffic loads, the free version of the K6 load testing tool was employed.

For the predefined traffic pattern, the objective was to replicate the hourly traffic pattern of a weekday from the Telecom Italia dataset, maintaining both trend and seasonality characteristics. However, discrepancies in data extraction and limitations in the parameters of the K6 traffic-generating tool impeded the successful recreation of the traf-

fic pattern. Despite these constraints, a synthetic traffic pattern was developed, exhibiting multi-trend and volatility attributes. This synthetic pattern is designed to challenge forecasting models and autoscaling decision-making processes in subsequent phases of the research.

Additionally, the pod hosting the web application was deployed with no resource limitations and isolated from other deployments within the cluster to prevent CPU throttling and minimize potential interference with CPU usage data. This approach ensures that the CPU usage metrics accurately reflect the resource demands of the specified traffic pattern. Any interference during data collection could distort these metrics and impact scaling decisions in subsequent stages of the analysis.

During the data collection process, Prometheus concurrently extracts both incoming request rate and corresponding CPU usage metrics at same timestamps. The resulting dataset comprises 1,000 data points for each metric, spanning a duration of 4 hours and 10 minutes. The incoming traffic rate varies from 0 to 255 requests per second, while CPU usage ranges from 0 to 520 millicores.

4.3.4 Data pre-processing, model design and training

To train the selected deep learning models, raw data must be pre-processed to ensure compatibility with the model. This pre-processing involves handling missing values, encoding categorical data, and other necessary transformations. A crucial step in this process is data normalization, which addresses the vanishing gradient problem [135] by ensuring that all data features are within the same range. This normalization step also improves model convergence speeds [136]. In this study, min-max normalization was applied to both CPU usage data and incoming request rate data, as shown in equation 4.1.

$$x^{'} = \frac{x - min\,(x)}{max\,(x) - min\,(x)} \tag{4.1}$$

Where x represents the original value and $x^{'}$ denotes the normalized value.

Given that the selected deep learning models fall under the category of supervised learning models, the training processes require target label data in addition to the input features. Consequently, the normalized data must be restructured to align with the deep learning model architecture. To achieve this, a sliding window approach is employed. In this approach, the model utilizes a window of past values to predict a future window, with the window sliding one step at a time. For this experiment, the past value window is fixed

at 10 steps, while the future value window ranges from 1 to 12 steps. This configuration facilitates the assessment of the model's forecasting accuracy across varying time horizons. For a given time t_i, incoming request rate $R[t_i]$ and total CPU usage $C[t_i]$ input time series for the forecasting model can be represented as $\{(R[t_{i-k}], C[t_{i-k}])\}_{k=0,\ldots,9}$ where k represents the step numbers. Similarly, the output time series $\hat{\Gamma}\left[t_i\right]$ given by equation 4.2 where $\hat{C}\left[t_{i+k}\right]$ is the forecasted CPU usage value.

$$\hat{\Gamma}\left[t_i\right] = \left(\hat{C}\left[t_{i+k}\right]\right)_{k=1,\ldots,12} \tag{4.2}$$

Existing literature demonstrates that the input window size has a substantial effect on model accuracy [137]. Specifically, larger window sizes enhance informational content but concurrently increase computational complexity. Conversely, smaller window sizes lower computational demands but reduce the amount of information available. An in-depth analysis of these trade-offs was beyond the scope of the present study.

In alignment with the defined model architecture, the normalized data were adjusted to conform to the prescribed input and output window sizes. For the purpose of model training, the dataset was partitioned into two segments: 60% was designated for training, and the remaining 40% was allocated to testing. Given the sequential nature of the data, it was imperative to preserve the continuity of the data stream throughout the preprocessing stage.

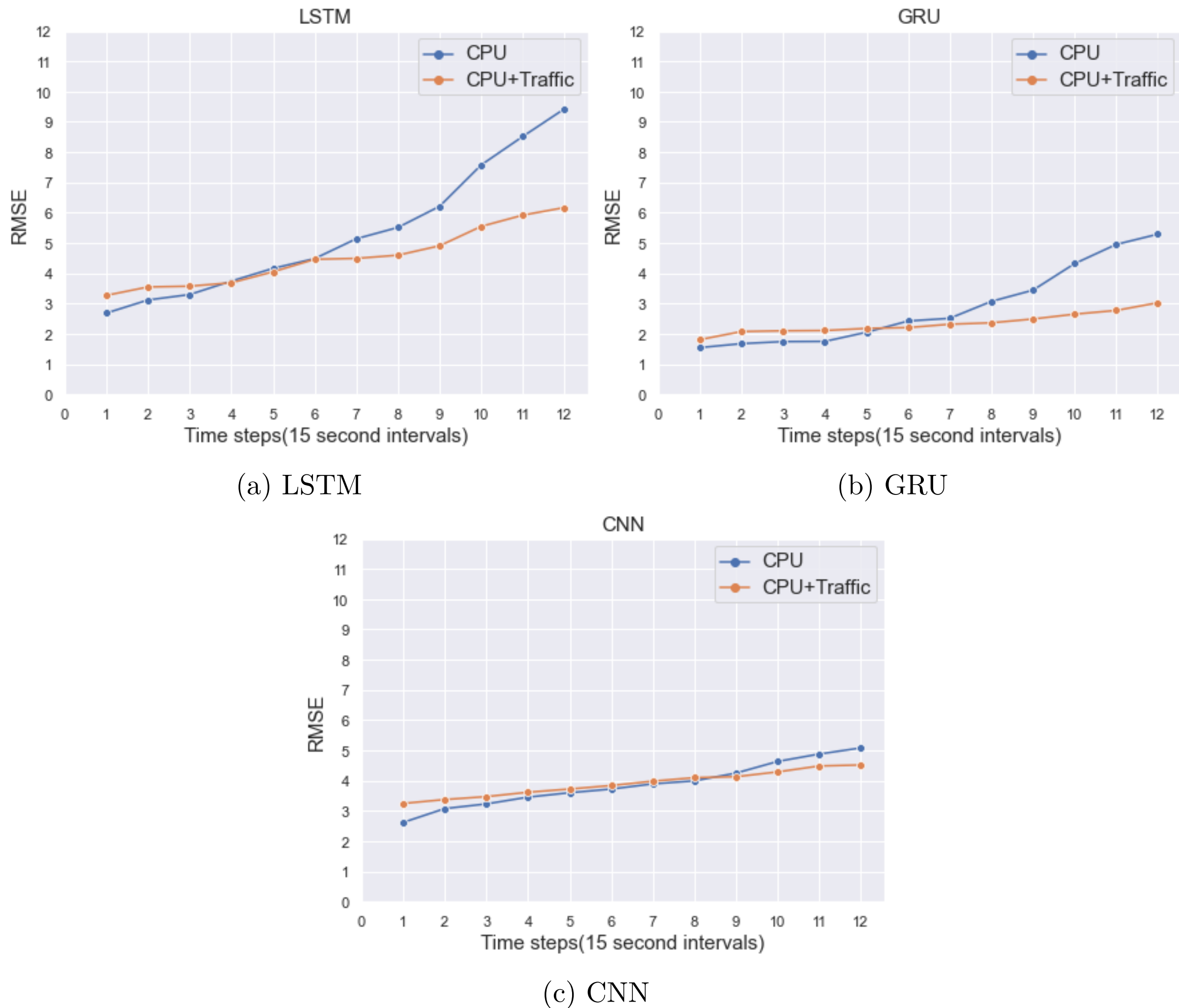
4.4 Results

4.4.1 Evaluation metrics

Several methodologies are available for assessing the accuracy of time series predictions. To evaluate the accuracy of the forecast, the error between the target labels and the model outputs is computed. Two of the most widely utilized metrics for error measurement in time series forecasting are Root Mean Square Error (RMSE) and Mean Absolute Error (MAE). RMSE values can be calculated using equation 4.3 where the label data (target time steps) represents x_i and predicted data represents $\hat{x_i}$ and n represents the number of data points.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x_i})^2} \tag{4.3}$$

MAE can be calculated using the equation 4.4 where x_i represents the label data



(a) LSTM

(b) GRU

(c) CNN

Figure 4.8 – RMSE values for forecasting horizons from 1 to 12. (from [4]) (Copyright © 2022 IEEE)

(target time steps) and $\hat{x}_i$ represents the predicted value and n represents the number of data points. And MAE gives the absolute average error value for the prediction.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|x_i - \hat{x}_i| \tag{4.4}$$

4.4.2 Prediction accuracy

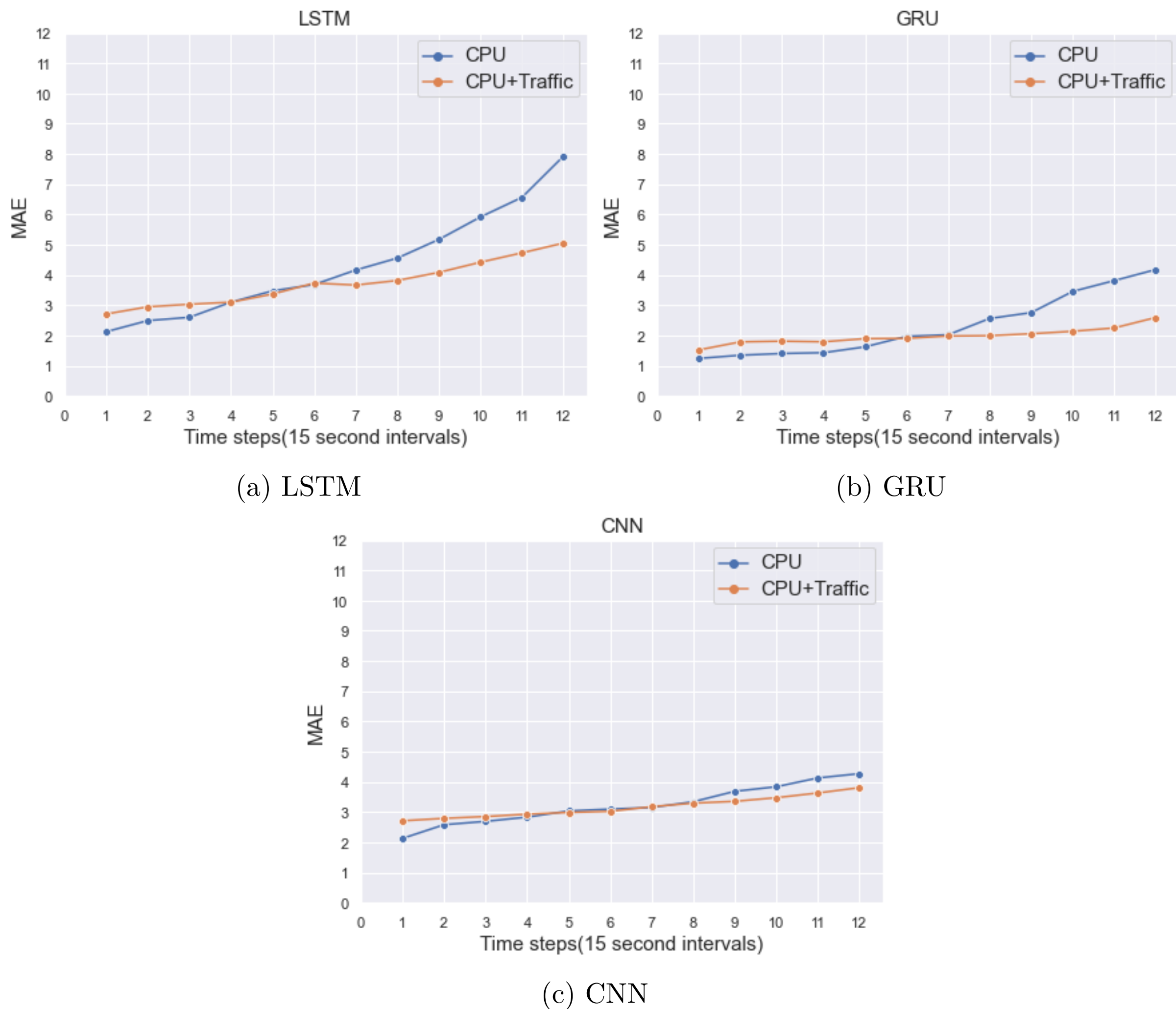
The trained models were assessed using the test dataset by generating forecasts for horizons spanning from 1 to 12 time steps and subsequently computing the RMSE and MAE metrics. Forecasting future CPU usage using the proposed approach, which inte-

grates both incoming request rate and CPU usage, is expected to yield relatively high accuracy. This is due to the model's capability to account for the temporal patterns of both features and their interactions, in contrast to univariate CPU forecasting models that rely solely on temporal patterns of CPU usage. To evaluate the advantage of this approach in forecasting, a comparative analysis was conducted against univariate deep learning models that were exclusively trained on historical CPU usage data. For this comparison, the same deep learning models and datasets were employed, excluding the incoming request data, and identical preprocessing procedures were applied to the univariate model. For ease of reference, the proposed multivariate models will be referred to as M_{C+R} mode, and the univariate models will be referred to as M_C mode. Each model under each mode was tested 10 times to ensure robustness, and the average RMSE and MAE values were calculated. and shown in the Figure 4.9.

Since RMSE and MAE measure the error metrics of the forecast, predictions with relatively lower RMSE and MAE values indicate higher forecast accuracy. The RMSE values calculated in the LSTM and GRU forecasting models under M_{C+R} mode are initially higher than those under M_C mode, but the difference between the two modes decreases as the step size increases. For steps beyond 6, the RMSE values of M_{C+R} mode become lower than those of M_C mode, and the difference between RMSE values widens with the step number. However, although the CNN model under M_{C+R} mode shows a slightly higher RMSE value compared to M_C mode, the RMSE difference between the two modes remains insignificant until step 9. Beyond step 9, the difference between the two modes starts to widen slightly. Globally, all models under all modes show increasing RMSE values with step size. Additionally, this increase is gradual in M_{C+R} mode, whereas M_C mode experiences a more aggressive increase. The behavior of the calculated MAE values for all models under all modes is similar to the RMSE plots.

Due to the squaring of errors prior to averaging, the RMSE assigns greater weight to larger errors. Consequently, RMSE values are highly sensitive to substantial deviations; sharp increases in RMSE at higher step numbers suggest a significant deterioration in forecasting accuracy with increasing forecast horizons. Conversely, the MAE exhibits a similar trend, indicating fewer extreme outliers and reduced bias in error direction.

Based on the behavior of the RMSE and MAE values across forecast horizons, M_C mode is beneficial for shorter forecasting horizons, while M_{C+R} mode is more advantageous for longer forecasts. In M_C mode, the time series forecasting model needs to learn past time series properties, whereas in M_{C+R} mode, the model must learn both the time



(a) LSTM (b) GRU

(c) CNN

Figure 4.9 – MAE values for forecasting horizons from 1 to 12. (from [4]) (Copyright © 2022 IEEE)

series properties of each feature and the interrelationships between features, increasing the complexity of the learning phase. The results indicate that this complex correlation in M_{C+R} mode negatively impacts short-term forecasting but significantly benefits long-term forecasting, particularly in LSTM and GRU models compared to M_C mode. Additionally, the CNN model shows a smaller difference between M_C and M_{C+R} modes in RMSE and MAE values, indicating that the additional correlations between incoming request rate and CPU usage have less impact on model accuracy.

Among all models and modes evaluated, the GRU model exhibits the lowest RMSE and MAE values, indicating it is the best-performing model, especially for longer fore-

casting horizons. The results demonstrate that the proposed solution, which integrates both incoming request rate and CPU usage, offers distinct advantages over alternative methods, particularly from the perspective of proactive autoscaling. Providing long-term forecast analytics with relatively high accuracy is beneficial for understanding future resource usage patterns in Kubernetes pods and facilitating scaling decisions to optimize the trade-off between cost and QoS.

4.5 Summary

This chapter explores the trade-off between cost and QoS in cloud-native applications, with a particular focus on cloud-native 5G environments through dynamic autoscaling. As 5G networks expand, managing this balance becomes increasingly challenging due to their large user capacity and diverse use cases. To address this, the chapter introduces a novel proactive autoscaling solution tailored to the resource demands of 5G applications.

The chapter begins by identifying key resource metrics that influence autoscaling decisions, emphasizing the importance of understanding the correlation between incoming traffic and resource utilization for accurate prediction-based autoscaling. It then presents a high-level model of a two-stage autoscaling solution, focusing on the design of the first stage. In this stage, a new resource usage forecasting approach is proposed, leveraging deep learning models that integrate both system-level and application-level metrics. This approach aims to accurately predict future resource needs, providing real-time analytics essential for the next stage of the autoscaling solution. The results show that this forecasting method is more accurate than traditional approaches that rely on either system-level or application-level data alone.

However, a notable limitation of this research is its focus on a limited subset of 5G CN NFs and types of services for analyzing resource usage profiles and correlations within 5G networks. To enhance the generalizability of the proposed solution, future work should extend the experimental scope to include all network functions and a comprehensive range of services. Furthermore, owing to limitations inherent in open-source 5G traffic generators at that time, specifically their incapacity to generate complex traffic patterns, a synthetic dataset derived from a web application was used for training and testing the forecasting model. The research would benefit from employing real 5G traffic data across various network functions to improve the accuracy and applicability of the forecasting approach. The chapter concludes by setting the stage for the next phase of the autoscaling

solution, where the analytics from the forecasting model will be used to make precise scaling decisions, ultimately aiming to balance cost and QoS more effectively.

Chapter 5

AI-assisted proactive autoscaling solution for CNFs

5.1 Introduction

The second phase of the proposed autoscaling solution focuses on interpreting the CPU usage forecast generated in the previous phase and determining the appropriate scaling actions, including the required number of replicas and the timing of these decisions.

This decision-making process is inherently complex. Even with precise CPU usage forecasts, autoscaling decisions can easily disrupt the cost-QoS trade-off. Despite utilizing forecasted values and proactive scaling decisions, similar issues encountered in reactive threshold-based scaling approaches may still arise. Incorrect identification of scaling actions or miscalculation of the necessary replicas can lead to resource overprovisioning, increasing operational costs, or under-provisioning, resulting in QoS degradation due to resource shortages. Furthermore, the cost-QoS trade-off may be adversely affected by improper timing of scaling decisions. It is crucial to recognize that scaling decisions are based on future CPU usage predictions, and there is a latency between the creation and operational readiness of new replicas. Consequently, the operational cost and QoS are highly sensitive to the timing of scaling decisions, whether they are made too early or too late. This chapter conducts an in-depth analysis of how scaling decisions impact the cost-QoS trade-off. A novel dynamic scaling model is introduced, leveraging real-time forecasted CPU usage data to determine the optimal scaling actions and required number of replicas by employing both static and dynamic thresholds. Subsequently, the timing of scaling decisions is experimentally assessed to achieve a balanced cost-QoS trade-off. This new dynamic scaling model is then integrated into the comprehensive autoscaling solution and evaluated under real-time traffic conditions within a Kubernetes environment. The performance of this solution is compared against Kubernetes HPA under various thresholds and a state-of-the-art proactive autoscaling solution. The results of each comparison

experiment are presented to identify the model that best balances the cost-QoS trade-off.

5.2 Research challenge

One of the primary challenges in implementing an autoscaling solution is the precise calculation of required resources and the determination of optimal timing for executing scaling decisions. In the context of horizontal scaling, this involves accurately computing the necessary number of replicas and deploying them at appropriate moments during runtime. The proposed autoscaling solution adopts a proactive approach, utilizing real-time CPU usage forecasting to inform scaling decisions and timing. To ensure the correct number of pods is allocated and to optimize scaling timing, several factors need to be considered.

Reducing operational costs through autoscaling can be achieved by minimizing the deployment's operational time, particularly under a pay-as-you-go pricing model. In the case of horizontal scaling, the total operational cost for a given period is calculated by summing the operational time of all replicas within that period, as detailed in Section 2.3.4. Consequently, reducing the number of replicas directly lowers operational costs. One effective strategy to minimize the total operational time and reduce the number of replicas is to maximize the resource utilization of existing replicas before triggering further scaling. Conversely, replicas should be scaled down promptly when there is a significant decrease in resource utilization. This approach minimizes underutilized replicas, thereby reducing the required number of pods and associated costs.

However, aggressive cost-reduction strategies can adversely impact the QoS of the deployment. Delaying the scaling-up of the deployment until existing replicas reach their maximum resource capacity can result in an inability to provision additional replicas in a timely manner. This delay may cause existing replicas to operate under resource starvation, leading to QoS degradation. For instance, replicas experiencing CPU starvation will struggle to process incoming requests promptly, directly affecting service response times. Conversely, prematurely scaling-down replicas can redirect the workload from terminated replicas to those still operational, increasing their resource usage and negatively impacting QoS. This sudden surge in resource demand may push existing replicas beyond their capacity, potentially triggering oscillations in scaling decisions. Such oscillatory behavior can degrade the performance of CNFs, further compromising QoS.

While these issues are more prominent in reactive autoscaling strategies, they can

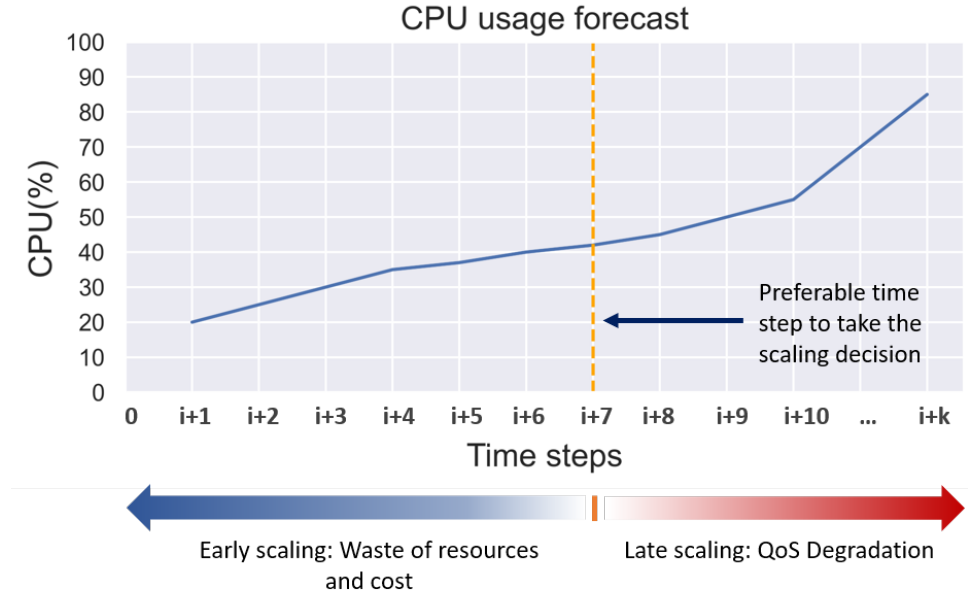


Figure 5.1 – Selecting the time step to scale in the CPU usage forecast. (from [5]) (Copyright © 2023 IEEE)

still arise in proactive scaling strategies if not properly addressed. The proposed autoscaling solution forecasts future CPU usage of the deployment in real-time, enabling proactive scaling decisions to manage the cost-QoS trade-off effectively. Additionally, the forecasting stage provides long-term forecasts with relatively high accuracy, facilitating an understanding of CPU usage behavior within the forecast horizon.

The primary challenge is to maximize CPU usage of the replicas while accounting for the scaling-up delay. When forecasted CPU usage indicates that expected usage will exceed the existing capacity of the replicas, the autoscaling solution must identify this need in advance and deploy additional replicas before the existing replicas reach maximum capacity, considering the scaling-up delay. Similarly, when forecasted CPU usage indicates a decrease, the autoscaling solution must scale down promptly while estimating workload distribution among existing replicas to prevent sudden CPU usage spikes and minimize the impact of oscillation on QoS.

Consider a CPU usage forecast provided by the deep learning model at time t_i. An example output of the forecasting model is shown in Figure 5.1. The forecasting model predicts $i + k$ steps into the future, indicating an increase in CPU usage in the upcoming time steps. If the autoscaling solution executes the scaling decision at t_i by analyzing the forecast from t_i to t_{i+k}, several scenarios are possible. If the scaling-up decision is exe-

cuted earlier than the optimal timing, operational costs may increase due to unnecessary early resource allocations. Conversely, if the scaling-up decision is executed later than the optimal timing, there will be insufficient time for a new replica to become ready and operational, thus increasing the service response time.

Similarly, if the expected CPU usage is projected to decrease in the upcoming time steps, executing the scaling-down decision earlier than the optimal timing will redirect the incoming load to other operational pods, potentially exceeding their capacity and triggering a subsequent scaling-up action. This scenario can create an oscillation or ping-pong effect between scaling-up and down, negatively impacting service response time. If the scaling-down decision is executed too late, operational costs will increase as resources are allocated for an extended period without being fully utilized. Therefore, to balance QoS and operational cost, scaling-up and down decisions must be carefully dimensioned. The solution must leverage accurate CPU usage forecasts to make timely scaling decisions that preemptively address potential resource constraints and avoid unnecessary operational costs.

5.3 Dynamic scaling model

In the subsequent phase of the proposed autoscaling framework, an advanced decision-making algorithm is implemented to address the previously identified challenges associated with the cost-QoS trade-off. The primary functions of this model are to process the output of the CPU usage forecasting model, determine the appropriate scaling direction (up or down), compute the required number of replicas, and optimize the timing of scaling operations. To accurately interpret the forecasted CPU usage and determine the scaling direction, two thresholds are established. These thresholds filter the forecasting model's output, thereby dividing the decision-making process into three distinct zones, as illustrated in Figure 5.2.

5.3.1 Scaling-up

To filter scaling-up events in the forecasted output sequence generated by the deep learning model, a scaling-up threshold is introduced. This threshold is used to identify scenarios where the autoscaler must add new replicas, thereby increasing resources to handle anticipated resource usage while maintaining expected QoS levels. The objective

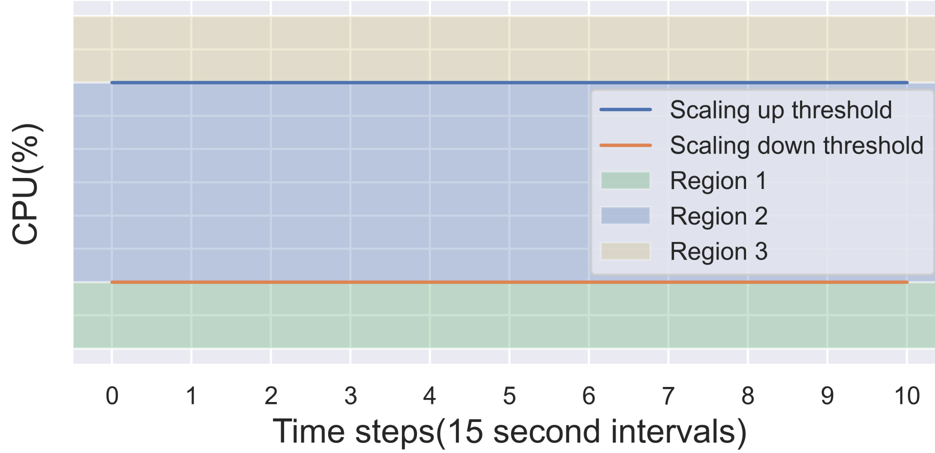


Figure 5.2 – Dynamic scaling model decision space. (from [5]) (Copyright © 2023 IEEE)

is to allow existing replicas to fully utilize the allocated resource capacity and trigger scaling-up before resource usage surpasses this capacity, which could negatively impact the service response time of the CNF. In this context, the maximum resource capacity refers to the CPU request specified in the Kubernetes pod. Consequently, the scale up threshold can be aligned with the CPU request (or 100% per-pod utilization) to detect scaling-up events in the forecast.

Considering these factors, a new formula has been developed to calculate the required number of replicas during scaling-up, as shown in equation 5.1. This formula determines the number of replicas $\hat{P}[i]$ at each forecasted step by dividing the forecasted total CPU usage per service $\hat{C}[i+k]$ for k number of steps by the CPU request. The maximum value obtained from these calculations represents the minimum number of pods required at a given instance i to maintain the necessary QoS levels. Choosing a higher number of replicas than this value results in overprovisioning, leading to increased costs, while selecting a lower number of replicas leads to QoS degradation due to insufficient resources. Since the formula may produce fractional values, and the number of replicas must be an integer, these values are rounded up to the next nearest integer.

$$\hat{P}[i] = \lceil max\{\frac{\hat{C}[i+k]}{CPUrequest}\}_{k=1...12}\rceil \tag{5.1}$$

5.3.2 Scaling-down

To achieve cost-effective operations, it is essential to reduce the number of replicas as soon as resource utilization decreases. However, determining the appropriate threshold for resource decline that permits safe scaling-down is critical. To address this challenge, a dynamic scaling-down threshold is introduced that adapts based on the expected redistribution of processing loads after scaling-down. In Kubernetes, when a pod is terminated, the total traffic for the service is redistributed among the remaining replicas. Kubernetes, by default uses a round-robin load-balancing algorithm for distributing incoming requests across replicas [138] resulting CPU usage across all replicas is nearly uniform. As a result, an increase in workload on the remaining replicas is expected, leading to higher average CPU usage. It is critical to ensure that this increase does not exceed the full capacity of each remaining replicas during the scaling-down events. Failure to manage this appropriately can lead to an oscillating (ping-pong) effect between scaling-up and down, thereby affecting QoS.

Given these considerations, to minimize the impact on average CPU usage in the remaining replicas after scaling-down, a sequential termination of pods is preferred over simultaneous termination. Consequently, when determining the scaling-down threshold, the average CPU usage of the remaining replicas after scaling-down must remain below the CPU request, or below 100% in percentage terms, as demonstrated in the example case illustrated in Figure 5.3.

According to the aforementioned principle, consider a scenario where the current number of replicas $P[t_i]$ is operating with an average CPU usage equal to the scaling-down threshold. For a given moment t_i the scaling-down threshold $\phi[t_i]$ can be calculated as,

$$\phi[t_i] = \frac{CPU\ request \times P[t_{i+1}]}{P[t_i]} \tag{5.2}$$

where $P[t_{i+1}]$ is the number of pods after scale down. Since the termination of pods is sequential,

$$P[t_{i+1}] = P[t_i] - 1 \tag{5.3}$$

From 5.2 and 5.3,

$$\phi[t_i] = \frac{CPU\ request \times (P[t_i] - 1)}{P[t_i]} \tag{5.4}$$

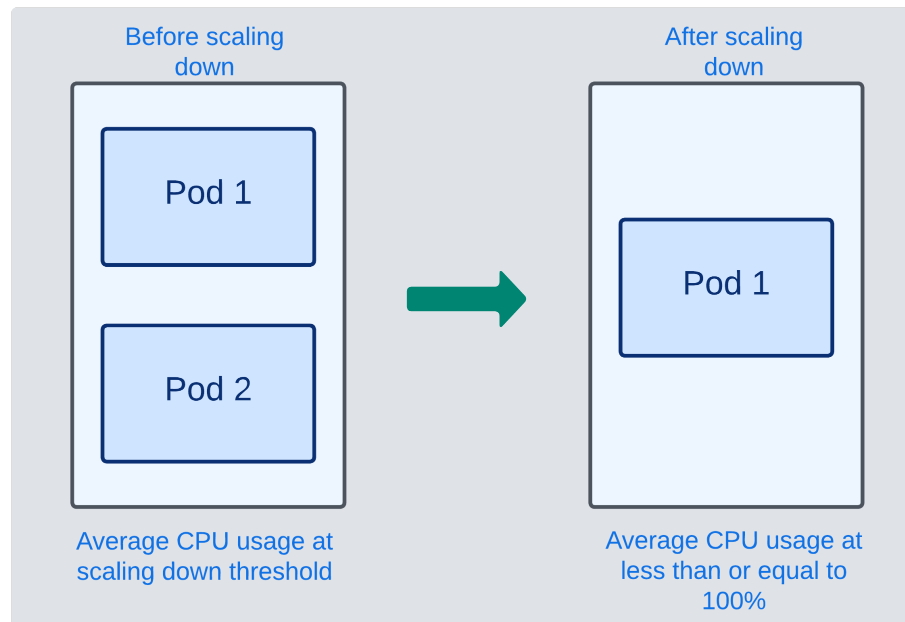


Figure 5.3 – Average CPU usage post scale down must not surpass the remaining replica resource capacity.

When scaling-down one pod at a time, it is more cost-effective to determine the scaling-down threshold by considering the load redistribution for the remaining replicas that only exceed their CPU usage up to their maximum capacity. However, in practical environments, the CPU usage of individual replicas can deviate slightly from the average. If the CPU usage of any replica exceeds its full capacity, even marginally, it can lead to QoS degradation. Therefore, the solution includes a 20% buffer of the full capacity when calculating the scale down threshold.

Furthermore, the proposed pod scaling-down mechanism enforces a constraint that limits the scaling-down to a minimum of one replica at all times, irrespective of whether resource consumption exceeds the specified scale down threshold. This design choice is put in place to guarantee uninterrupted service availability.

Additionally, the scaling-down process is initiated only when all forecasted values consistently remain below the dynamic scale down threshold. Utilizing extended forecast sequences for CPU usage enables the autoscaling system to mitigate fluctuations and reduce the risk of QoS degradation caused by volatile traffic patterns.

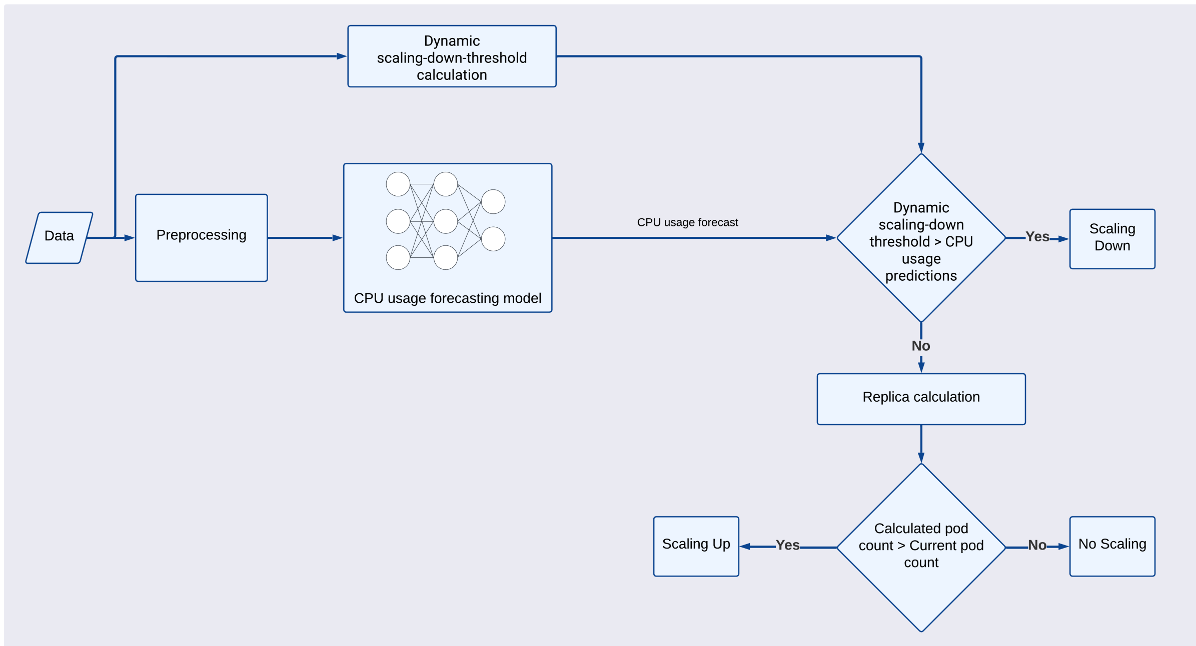


Figure 5.4 – Autoscaling decision-making process.

5.3.3 No scaling

In a given situation, if the forecasted CPU usage values lie between the scaling-up and scaling-down thresholds, it indicates that the anticipated CPU usage can be managed by the current number of operational replicas. Consequently, no scaling action is necessary. The complete autoscaling decision-making process is illustrated in the Figure 5.4.

5.3.4 Decision timing

The proposed autoscaling solution bases scaling decisions on forecasted CPU usage values, making the timing of these decisions critical for balancing cost and QoS trade-offs. Specifically, in the context of scaling-up events, an anticipated increase in CPU usage first manifests at the end of the forecasted sequence (at the horizon). The proposed solution initiates scaling actions at the current time step based on predicted future CPU usage values. Consequently, longer forecast sequences prompt earlier scaling actions, which may result in additional replicas being provisioned before the CPU usage reaches its peak. This preemptive scaling can be suboptimal, as it may prevent the CPU usage from reaching full capacity, thereby impacting long-term operational costs. The primary question is: What is the optimal length of the forecasted CPU usage sequence that should be monitored to

initiate the scaling-up decision, thereby maximizing CPU utilization while maintaining acceptable QoS levels.

An optimal strategy involves aligning the forecast horizon with the scaling-up delay in Kubernetes. This approach ensures that additional replicas are fully operational by the time CPU usage of the existing replicas is expected to peak.

However, this strategy may not be effective in practice for several reasons. First, even if additional replicas are deployed and operational when the current replicas reaches its maximum CPU usage, the round-robin load balancing algorithm may still distribute incoming requests equally among all replicas. As a result, the replicas that has already reached its maximum CPU capacity will continue to receive new requests, leading to potential QoS degradation. Furthermore, this degradation is influenced by the incoming traffic rate . Additionally, although the most accurate forecasting model was selected, the predictions are not entirely precise. The accuracy of the forecast decreases as the number of steps forecasting increases, as discussed in Section 4.4.2. Consequently, longer forecast horizons with decreasing accuracy at the end of the sequence directly impacting scaling-up decisions and their timing.

Conversely, the proposed algorithm has already addressed the timing for scaling-down. When the average CPU usage falls below the scaling-down threshold for the duration of the entire sequence, the additional replicas will be terminated immediately.

Given these challenges, determining the optimal timing for scaling-up decisions is a complex problem that requires balancing the cost-QoS trade-off. Instead of relying solely on a theoretical approach, an experimental, iterative method was employed to identify the appropriate forecast length for executing scaling-up decisions. The complete scaling solution was deployed alongside the target application for autoscaling, and a traffic pattern (see Section 4.3.3) was injected (detailed deployment procedures will be explained in the next section).

Experiments were conducted by setting the forecast length to a fixed value and allowing the autoscaler to make scaling decisions. Upon completion of each experiment, Key Performance Indicators (KPIs) detailed in the next section were monitored to evaluate performance in balancing the cost-QoS trade-off. By systematically varying the forecast length and assessing the KPIs, this iterative approach identified a forecasting sequence length of 7 steps as providing the best balance between cost and QoS trade-offs.

The challenge of determining the optimal timing for decision-making in AI/ML domain is well known [139]. Although addressing this issue through a theoretical approach is

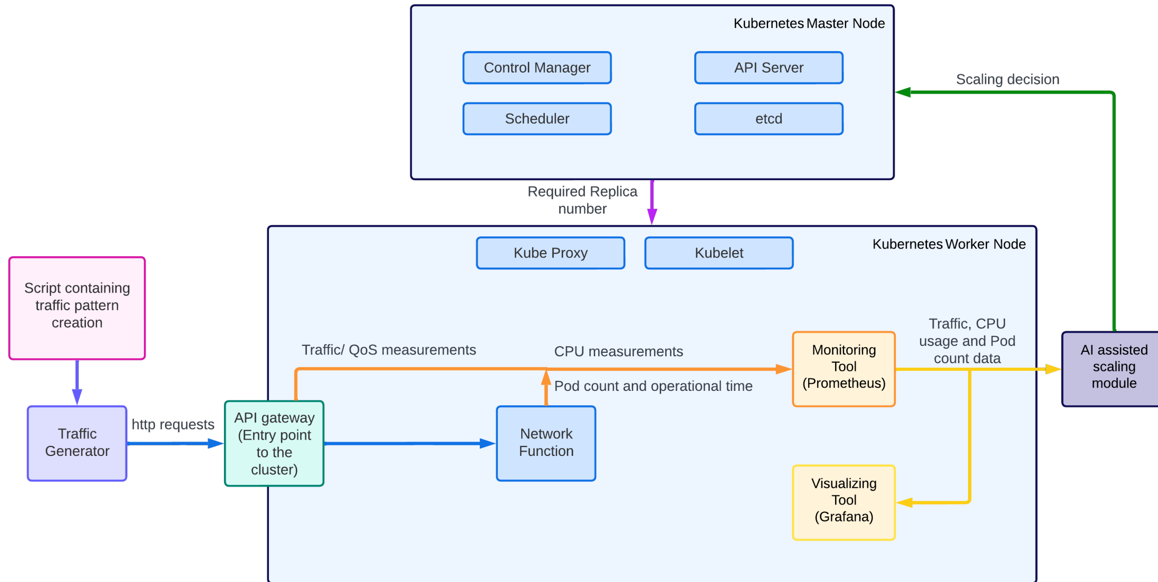


Figure 5.5 – Testbed for AI-assisted scaling solution. (from [5]) (Copyright © 2023 IEEE)

beyond the scope of this study, we draw readers' attention to studies such as [140] and [141], which address the cost-accuracy trade-off in the decision-making process for time series classification problems using a theoretical approach.

5.3.5 Testbed

To deploy the proposed autoscaling solution, the testbed described in Section 4.2.1 was utilized, incorporating several modifications to the system architecture. The complete configuration of the testbed is presented in Figure 5.5. The existing testbed setup uses Prometheus to collect data on incoming requests and CPU usage from the cluster. In addition to these metrics, the current replica count data is also retrieved for the dynamic scale down threshold calculation. Prometheus provides access to the collected real-time time series data via its APIs, enabling the solution to be deployed externally to the Kubernetes cluster. The same web application described in Section 4.2.2 , with identical resource configurations, was used as the target application to evaluate the proposed autoscaling solution. Following deployment, continuous traffic was generated using the script prepared during the data collection phase.

Similar to the Kubernetes HPA, the proposed solution continuously collects cluster information at 15-second intervals. It queries the Prometheus database for metrics in-

cluding incoming request rate, CPU usage, and current replica count data per service. The solution forecasts future CPU usage for 7 steps ahead at each interval and makes real-time scaling decisions based on these forecasts. These scaling decisions are communicated to the Kubernetes API server on the master node. The Kubernetes control manager and scheduler then execute these decisions by creating or terminating pods on the worker nodes.

5.4 Results

The proposed solution aims to minimize QoS fluctuations during scaling operations while simultaneously reducing operational costs. Consequently, the chosen KPIs for the evaluation process must reflect these objectives. For the evaluation, three KPIs have been selected: service response time, total replica count, and replica operational time.

5.4.1 KPIs for the evaluation

Service response time

As outlined in Section 5.2, delays in initiating the addition of new replicas or premature reduction in the number of existing replicas by the scaling mechanism can lead to QoS degradation. This degradation is primarily caused by CPU overload on the remaining replicas. To quantify QoS variations during the scaling process, the 95th percentile of the service response time—defined as the round-trip time between the API Gateway and the HTTP server replicas—was calculated.

Pod count and pod operational time

Inadequate scaling of replicas by the scaling mechanism can directly impact operational costs. Minimizing the use of additional replicas and their operational time can reduce costs in a consumption-based pricing model. Therefore, the scaling mechanism must create and terminate the desired number of replicas at the right time. To evaluate operational costs, data on the total replica count (number of active replicas) and their operational time per service were collected.

5.4.2 Benchmarking the autoscaling solution

To benchmark the performance of the proposed scaling solution against Kubernetes HPA and the leading state-of-the-art solution, experiments were conducted under identical testbed configurations and traffic patterns for each scenario.

HPA with different thresholds

The Kubernetes HPA was configured to scale based on pod CPU utilization using three predefined thresholds. These thresholds were selected to illustrate distinct operational scenarios. In the first scenario, the HPA triggers replica scaling when the average pod CPU utilization reaches 100%, prioritizing the reduction of pod count and operational duration over maintaining QoS levels during scaling. This approach allows pods to utilize their maximum CPU capacity before scaling occurs. In the second scenario, a 50% threshold was set, enabling the HPA to initiate scaling early, emphasizing the maintenance of higher QoS levels during scaling, albeit potentially increasing operational costs. Lastly, the third scenario employs an 80% threshold to strike a balance between maintaining elevated QoS levels during scaling and minimizing operational expenditures.

Autoscaling with traffic load prediction

To benchmark the proposed solution against a state-of-the-art solution, a proactive scaling approach based on traffic load prediction was adopted, inspired by a previous study [126]. This approach utilizes a deep learning-based time series forecasting model, specifically an LSTM model, to predict future traffic load and adjust the number of replicas in the cluster. The solution was implemented with modifications to suit this experiment testbed.

In the forecasting model, as described in the study, only incoming request rate data was used to predict future incoming request rates. Therefore, the LSTM model was trained using 60% of traffic data in the dataset created in Section 4.3.3 and validated using the remaining 40%. The model's accuracy was evaluated using the RMSE, achieving a value of 0.0849 when predicting traffic load for a one-time step interval of 15 seconds. After predicting the traffic load, the required number of replicas $\hat{P}[i]$ was determined based on equation 5.5:

For a given time t_i,

$$\hat{P}[i] = \min\left(vnf_{max}, \left\lceil \frac{\hat{R}[t_{i+1}]}{\gamma} \right\rceil\right) \tag{5.5}$$

In this context, $\hat{R}[t_{i+1}]$ denotes the predicted requests per second for the t_{i+1} time step, while vnf_{max} represents the maximum number of pods supported by the cluster, which can vary depending on the CPU allocation and cluster resource capacity. The parameter γ signifies the maximum number of requests a pod can manage, adjusted according to the CPU allocated per pod. For the experiments, each pod was provisioned with 400 millicores, and after empirical testing, it was determined γ to be 160 requests per second for this CPU configuration.

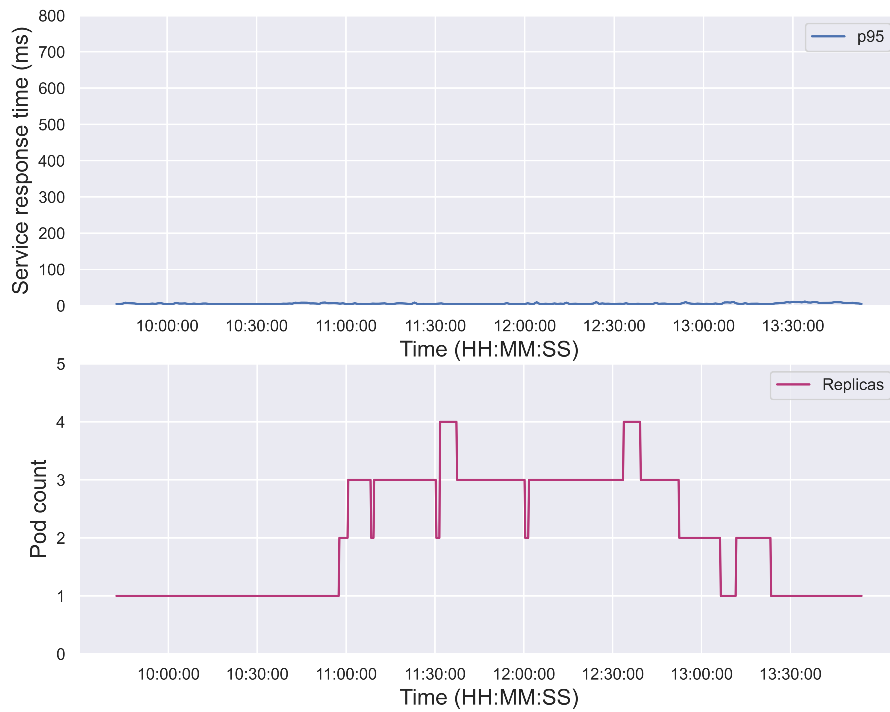
5.4.3 Autoscaler comparison

For each scenario, the experiment was conducted five times to assess the robustness of the results. Figure 5.6 depicts the service response time (95th percentile) and the total pod count over time for each experiment. Figure 5.6a shows the performance of the HPA with a scale up threshold of 100%. With this threshold, the HPA added a second replica when CPU usage spiked due to heavy traffic and terminated replicas even with brief drops in CPU usage. This led to numerous scaling-up and scaling-down events during the experiment. Each scaling-up event significantly increased service response time, reaching a maximum peak of 720 milliseconds (ms). However, the service response time swiftly returned to an average of 5ms once the additional replica was active. Although there were spikes in service response time during each scaling-up event, the peak values differed due to varying incoming traffic rates at those times. The HPA's 100% threshold led to delayed scaling, causing spikes in service response time due to insufficient time for newly created replicas to become operationally ready.

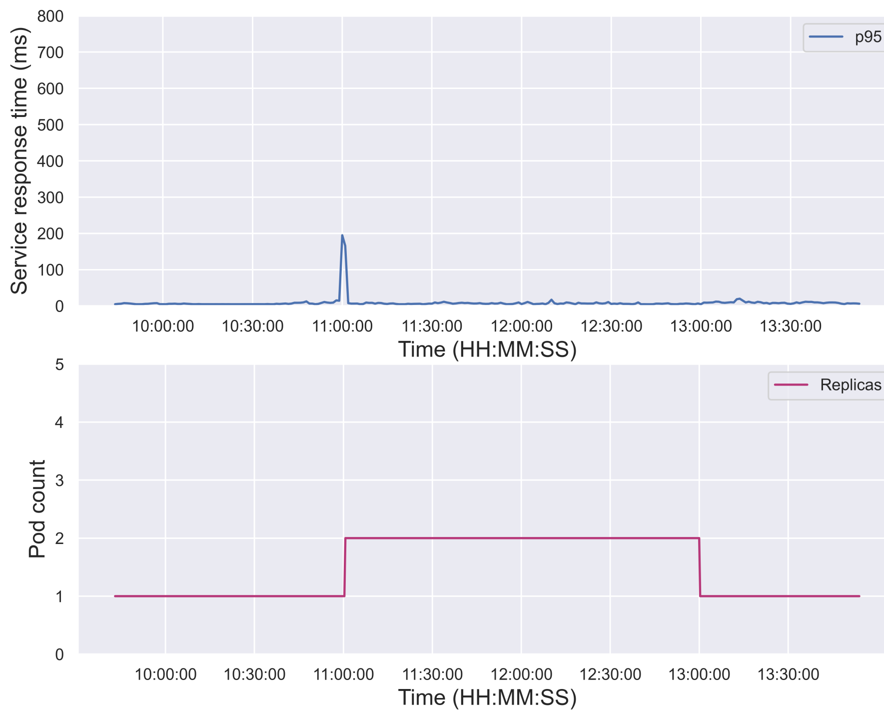
Figure 5.6b illustrates the service response time and pod count for the HPA with a 50% scale up threshold. In this setup, the HPA added replicas when the average CPU usage of existing replicas reached half their capacity, leading to a maximum of four pods during the experiment. Even though the HPA scaled down replicas during periods of lower CPU usage, the pod count remained higher compared to the HPA 100% threshold setup. In this case, the HPA maintained the service response time at 5ms without any spikes during scaling events. This demonstrates that an autoscaler prioritizing QoS levels can maintain stable service response times, albeit with increased operational costs.

For the HPA with an 80% scale up threshold, as depicted in Figure 5.6c, the HPA

(a) HPA 100%



(b) HPA 50%



(c) HPA 80%

(d) SOA Solution.

(e) Proposed AI assisted scaling

Figure 5.6 – Service response time during scaling. (from [5]) (Copyright © 2023 IEEE)

generated additional replicas during periods of high CPU usage. The HPA kept the second replica active throughout the peak period, not scaling-down during brief CPU usage drops. Importantly, this configuration led to only one scale up event, resulting in a single peak (maximum 195ms) during scaling, unlike the HPA with a 100% threshold. These observations indicate that the HPA with an 80% threshold provides a more balanced cost-QoS trade-off compared to the HPA with 100% and 50% thresholds.

Figure 5.6d shows the results of the autoscaler inspired by state-of-the-art solution [126], which scales based on predicted traffic loads. This solution scaled up for each CPU usage spike and scaled down during brief lower CPU usage periods. Each scaling-up event caused a spike in service response time (maximum 700ms) due to the delay in creating new replicas. Furthermore, this solution experienced more frequent scaling-up and scaling-down events compared to any HPA threshold setting. Despite predicting future traffic loads to determine the necessary number of replicas, the solution failed to proactively scale in a way that reduced service response time during scaling. This failure is likely because predicting one step ahead is insufficient for the autoscaling solution to detect and initiate scaling. Additionally, the number of requests that a single pod can handle,

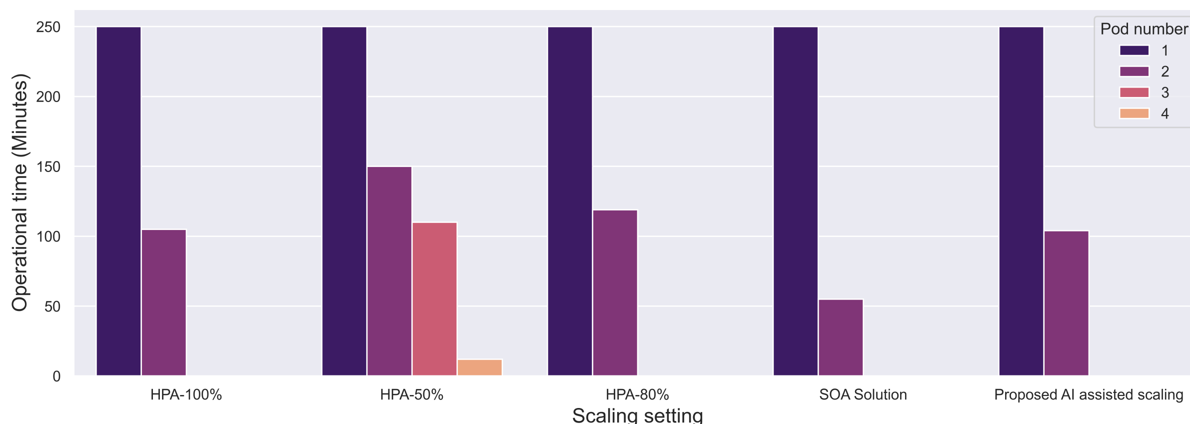


Figure 5.7 – Pod count and pod operational time. (from [5]) (Copyright © 2023 IEEE)

denoted as γ, can vary during request processing, potentially leading to inefficient scaling decisions.

Figure 5.6e presents the results of the proposed solution. The data indicate that service response time exhibited spikes during each scaling-up event, with a maximum peak of 50ms. The solution successfully scaled up the second replica in response to high CPU usage and scaled down during periods of low CPU demand. Notably, the solution effectively minimized service response time during scaling-up events, despite scaling-down the second replica briefly when CPU usage dropped. This demonstrates that accurate longer-term forecasts can enable proactive decision-making to minimize QoS degradation.

Figure 5.7 displays the operational time of replicas across all experiments. All approaches maintained the initial pod operational throughout the experiment due to the minimum replica count being set to one. However, the operational times for additional replicas varied depending on the approach. Specifically, the HPA with a 100% threshold kept the second replica operational for 105 minutes, HPA with an 80% threshold for 119 minutes, and HPA with a 50% threshold for 150 minutes—the longest duration among all approaches. The latter it also maintained two other replicas operational for 110 and 12 minutes, respectively. The state-of-the-art solution operated the second replica for only 55 minutes, the shortest duration observed. The proposed scaling solution utilized the second replica for 104 minutes, which is less than all HPA configurations but more than the state-of-the-art solution. Despite the state-of-the-art solution's lower operational time for the second replica, it exhibited the poorest service response time profile among all experiments. Overall, the proposed solution demonstrates a superior balance between minimizing operational costs and service response time during scaling events, outperform-

ing all other approaches evaluated in terms of cost-QoS trade-off. A significant observation from this experiment is that CPU throttling occurs even when CPU usage remains below the defined CPU limit, regardless of the scaling approach. Consequently, CPU throttling contributes to increases in service response time. Therefore, it is crucial to understand the behavior of CPU throttling and consider its impact on the autoscaling decision-making process.

5.5 Summary

In this chapter, the focus is on addressing the challenge of making precise scaling decisions based on resource usage forecasts provided in an earlier stage. The primary goal is to reduce operational costs by maximizing resource utilization, thereby minimizing unnecessary replication and their associated operational time. However, the inherent delay in scaling-up within Kubernetes, which can influence service response times during scaling events, is carefully considered. To tackle these challenges, the chapter introduces a new scaling model that utilizes both static and dynamic thresholds to determine whether to scale up, scale down, or maintain the current resource allocation. The timing of these scaling decisions, which significantly affects the balance between cost and QoS, is also explored and experimentally derived. The proposed autoscaling solution is then evaluated in a real-world test environment, where it is benchmarked against several other solutions. The results demonstrate that this proactive autoscaling approach offers a better cost-QoS trade-off compared to alternatives. However, the chapter also acknowledges limitations in the solution. Notably, the model does not account for errors in the forecasting model during threshold and decision timing calculations. To mitigate this, static buffer zones are used, though this is not the optimal approach. Additionally, the chapter suggests that the solution would benefit from further evaluation under conditions of high traffic volume and complex patterns to better assess its effectiveness and applicability. The chapter concludes by identifying CPU throttling as another significant factor that could disrupt the cost-QoS balance in the autoscaling solution, setting the stage for further investigation in the next chapter on how to mitigate this issue.

Chapter 6

CPU Throttling aware autoscaling

6.1 Introduction

In the preceding chapter, a novel proactive autoscaling solution was introduced to optimize the cost-QoS trade-off. Despite outperforming both the default Kubernetes HPA and a state-of-the-art alternative, the solution encountered increased service response times during the scaling-up process. This issue primarily arose from suboptimal decision timing within the proposed autoscaling mechanism. Although scaling-up decisions were made proactively, the timing was delayed, leading to QoS degradation. Analysis indicated that even though replicas were created seven steps before the pod reached its CPU capacity, QoS had already been adversely affected.

The root cause of this QoS degradation is linked to the CPU throttling mechanism inherent in the underlying system processes. While designed to enhance resource management in Linux environments by controlling and limiting pod resource overuse, this mechanism negatively impacts the timing computations necessary for effective scaling decisions, resulting in a suboptimal cost-QoS balance.

Kubernetes enforces CPU usage limits for CPU-intensive CNFs through a CPU throttling process managed by the Linux kernel's Complete Fair Scheduler (CFS) [142]. When a pod exceeds its designated CPU limit, CFS intervenes, throttling the pod and restricting its CPU usage beyond the defined limit. This throttling introduces computational delays, directly impacting the CNF's service response time. To maintain high application performance during dynamic scaling with CPU limits, understanding the CPU throttling mechanism is crucial to preventing service response time degradation.

This chapter investigates CPU throttling behavior and its impact on CNF service response times within the Kubernetes environment. Building upon the previously introduced autoscaling solution, an enhanced triggering mechanism is presented that dynamically adjusts the timing of scaling decisions by accounting for CPU throttling effects. This mechanism forecasts potential future CPU throttling events, providing insights to

avoid service response time increases due to throttling. The new solution was tested in a real-world Kubernetes environment, with results compared to previous experiments to evaluate its effectiveness in balancing the cost-QoS trade-off.

6.2 Research challenge

To mitigate the impact of CPU throttling on service response times during scaling operations, a comprehensive understanding of the CPU throttling mechanism within the Linux kernel is essential. In a Kubernetes deployment, assigning a CPU limit translates this parameter into a CPU quota within the Linux kernel, which is managed by the CFS in the cgroup subsystem.

A cgroup, or control group [143], is a Linux kernel feature designed to manage resources for a collection of processes. The CFS ensures equitable distribution of CPU time by periodically enforcing the CPU quota for each cgroup. This enforcement interval, known as the CPU period, has a default duration of 100ms. Therefore, when a CPU limit is set, it is converted into a corresponding CPU time quota within this CPU period.

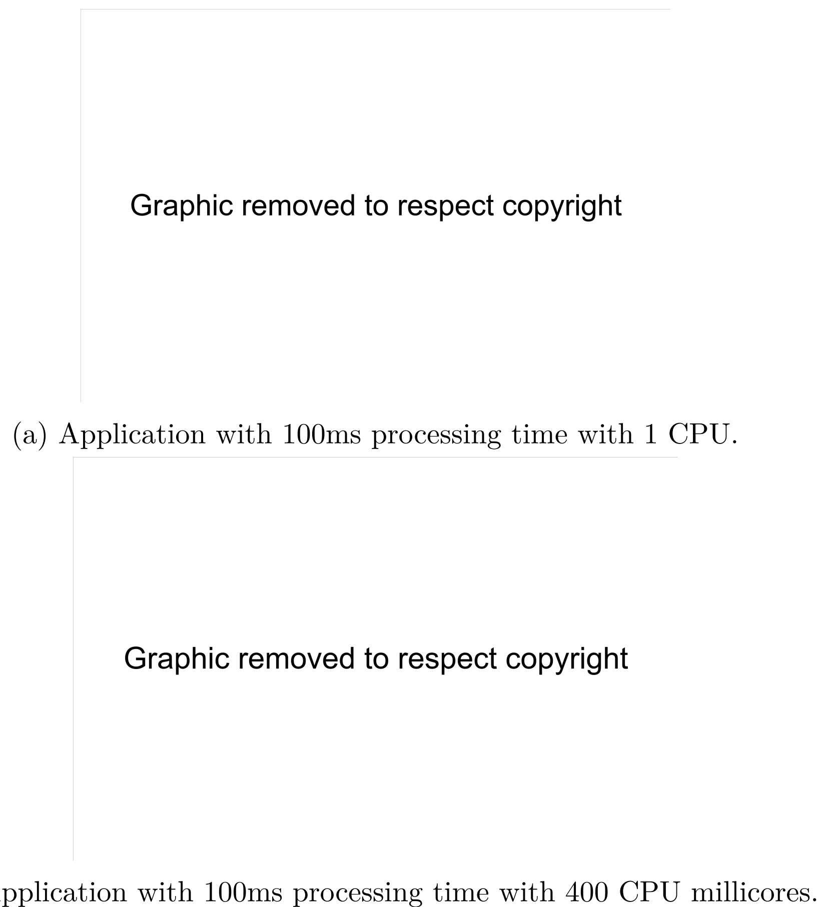
For instance, if a cgroup is allocated 1 CPU core (equivalent to 1000 millicores) limit, it is permitted to operate for 100ms within each 100ms CPU period. This allocation allows the cgroup to utilize 100% of the CPU during this period. If a cgroup exhausts its allocated CPU quota within the CPU period, the CFS will throttle the cgroup, causing it to halt processing until the next CPU period begins. This mechanism is referred to as CPU throttling.

Given this explanation, in a single threaded process, the CPU time quota (T_0) within a CPU period (T_w) for a specified CPU limit (C_{limit}) can be calculated as follows:

$$T_0 = \frac{C_{limit}}{1000} \times T_w \tag{6.1}$$

In Kubernetes, when a pod is provisioned with a defined CPU limit, the container runtime automatically creates a cgroup for the pod. This cgroup is configured with the specified CPU limit, ensuring enforcement of resource usage according to the defined constraints.

Due to this suspension of request processing caused by CPU throttling, service response time can be adversely affected. To elucidate the impact of CPU throttling on service response time, consider a single-threaded application deployed with a 1000 millicores CPU limit, requiring 100ms to process an incoming request (processing time - t_{base}) as



(a) Application with 100ms processing time with 1 CPU.

(b) Application with 100ms processing time with 400 CPU millicores.

Figure 6.1 – Effects of CPU throttling on application processing time – single-threaded application. (based on [144])

illustrated in the Figure 6.1a. In this scenario, the entire request can be processed within a single CPU period.

However, if the same application is deployed with a CPU limit of 400 millicores, the CPU quota allocated per CPU period, based on the equation 6.1, will be 40ms. Assuming the CFS schedules this CPU quota at the beginning of each CPU period, the CPU will be throttled for the remaining 60ms of the period. Consequently, the request cannot be fully processed within a single CPU period. Therefore, processing the entire request will span 2 CPU periods plus an additional 20ms, totaling 220ms as illustrated in the Figure 6.1b. This results in a service response time increase of 120ms due to throttling.

As previously mentioned, these calculations assume that the CFS allocates the CPU quota for this cgroup at the beginning of each CPU period. However, in practice, the CFS uses the red-black tree (rbtree) algorithm [145] to schedule CPU quotas throughout the

CPU period. The CPU period is divided into time slices, which are short intervals used by the CFS to schedule processes. Thus, the placement of the CPU quota can vary within the CPU period, which can cause fluctuations in service response time.

An experiment was conducted to analyze the impact of CPU throttling on the service response time of a web application. The objective was to measure the variation in service response time due to CPU throttling for known service times under a specified CPU limit.

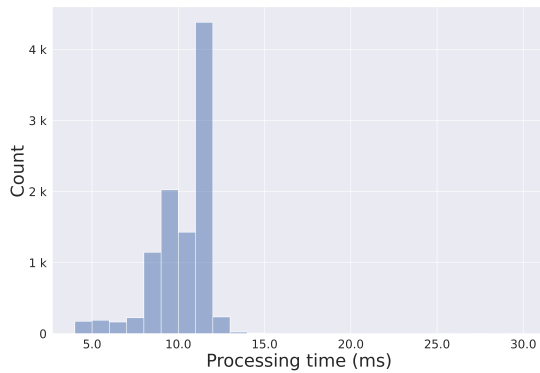
The experiment utilized the same testbed described in Section 4.2.1, with modifications to the web applications deployed on the cluster. The web application was modified to measure request processing time, thereby revealing the delay effects of CPU throttling when applied. Specifically, a timer was set at the beginning of the main function and another timer at the end of the main function to measure the time difference, representing the total time taken to process a request. This method reveals how processing time varies due to CPU throttling when applied and the CFS scheduling process.

The processing time data was then gathered through a monitoring service within the Kubernetes cluster. In addition to the aforementioned modifications, the main function in the web application was designed to allow the processing time (t_{base}) to be varied by adjusting the complexity of the function through a parameter. This will allow to measure processing time difference for different t_{base} values.

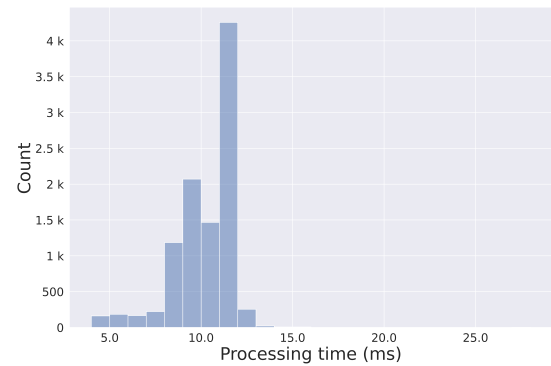
The modified web application was deployed in the Kubernetes cluster with two resource configurations. The first configuration had a CPU request of 1000 millicores CPU with no CPU limit denoted as C_{1000}, and the second configuration had a CPU request and CPU limit of 400 millicores CPU denoted as C_{400}, providing a CPU quota of 40ms within each CPU period. The objective of the first configuration is to measure the t_{base} in the absence of CPU throttling, while the second configuration aims to observe variations in t_{base} due to the presence of CPU throttling imposed by the CPU limit.

Subsequently, a steady stream of HTTP requests was injected into the pod. Each request was processed, and the processing time for each request was measured. To ensure robustness in the results, 10,000 requests were injected with a 1-second interval between consecutive requests for each case, and the results were collected. Given the t_{base} values under investigation, this interval was sufficiently large to prevent request congestion in the queues or overloading processes that could affect the processing time measurements. The experiment was repeated for all t_{base} values by adjusting the parameter in the web application.

To analyze the impact of CPU throttling and the CFS on processing time, four t_{base}

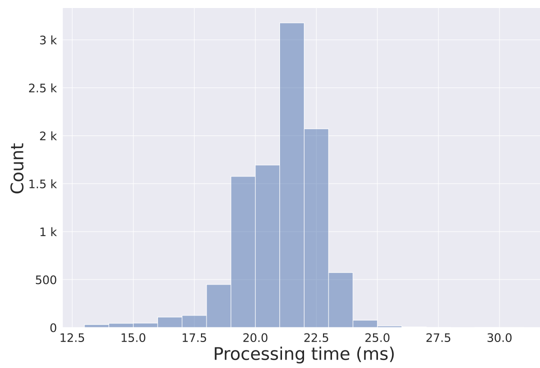


(a) Under C_{1000} configuration.

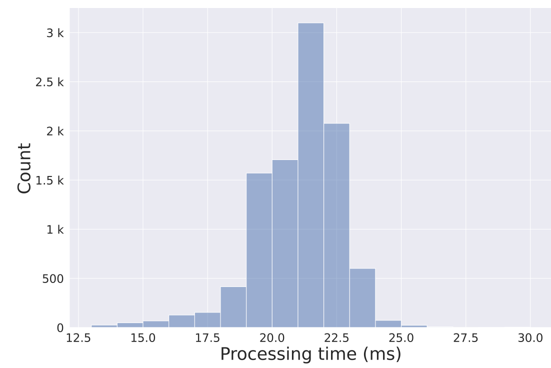


(b) Under C_{400} configuration.

(c) Processing time (t_{base}) $\approx 10ms$

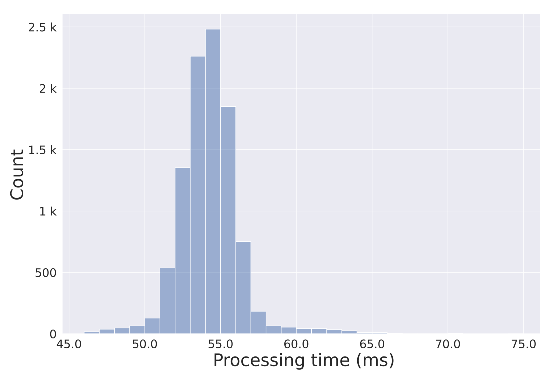


(d) Under C_{1000} configuration.

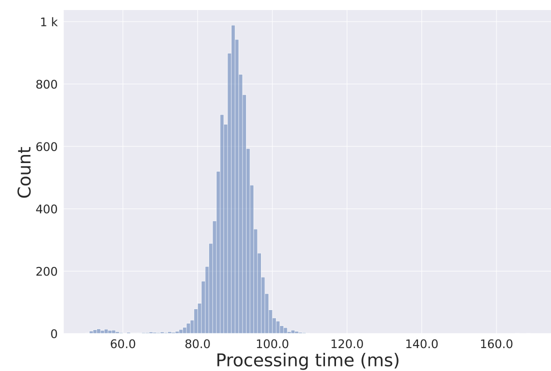


(e) Under C_{400} configuration.

(f) Processing time (t_{base}) $\approx 20ms$

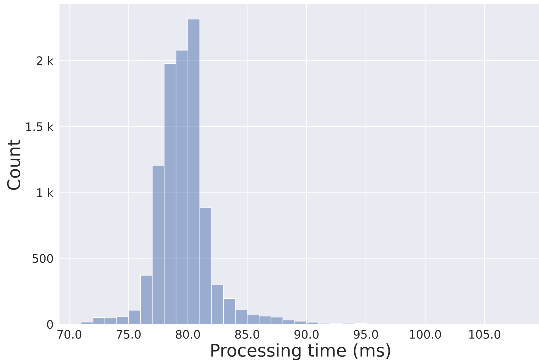


(g) Under C_{1000} configuration.

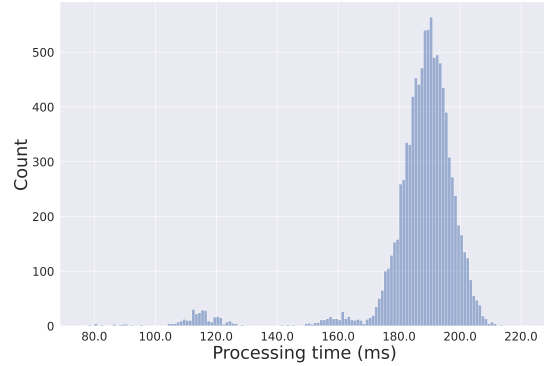


(h) Under C_{400} configuration.

(i) Processing time (t_{base}) $\approx 50ms$



(j) Under C_{1000} configuration.



(k) Under C_{400} configuration.

(l) Processing time (t_{base}) $\approx 80ms$

Figure 6.2 – Comparison of processing time variation with and without CPU throttling.

values were selected: 10ms, 20ms, 50ms, and 80ms. It is important to note that accurately tuning application parameters to yield precise t_{base} values is challenging due to the behavior of the CFS. As a result, each reported t_{base} value represents an average derived from 50 iterations for the selected parameter within the application. For each selected t_{base} value, a steady stream of requests was injected into the C_{1000} resource configuration, upon completion, another request stream directed to the second resource configuration, C_{400}.

The collected processing time data, as shown in Figure 6.2, reveals a notable observation: across all t_{base} values in the C_{1000} configuration, where CPU throttling is not anticipated, variability in processing times is present during processing. The distribution of processing times exhibits a single Gaussian pattern or combination of multiple Gaussian patterns especially in C_{400} cases. The observed variability is likely attributable to several factors, including compiler optimization techniques and cache memory optimization strategies, which may reduce the base execution time t_{base}. Additionally, delays introduced by the CFS due to background processes could further contribute, as CPU resources are allocated to the entire pod rather than being dedicated exclusively to the application within the pod, potentially increasing t_{base}.

For t_{base} values of 10ms and 20ms, the service response time remains near identical under both resource configurations. This consistency arises because of the absence of a CPU limit, there is no CPU throttling under C_{1000}, and under a CPU limit of 400 millicores C_{400}, t_{base} is less than the CPU time quota ($t_{base} < 40ms$), thus preventing the need for CPU throttling processes to engage. Consequently, the processes proceed without

interruptions.

However, for t_{base} values of 50ms and 80ms, which exceed the CPU time quota in C_{400}, CPU throttling was expected. In this scenario, a single request could not be processed within a single CPU period, causing it to span multiple CPU periods. As a result, the processing time with CPU throttling increased significantly as shown in the Figure 6.2h and 6.2k. Under both resource configurations, for the selected t_{base} values, the experimental processing time distribution exhibits a distinct mean value, regardless of CFS scheduling and CPU throttling mechanisms.

This experiment highlights the complexity of kernel-level scheduling and its impact on processing time when CPU throttling is introduced. The experiment investigated scenarios in which t_{base} is both lower and higher than the CPU time quota. In standard cloud engineering practice, a resource configuration where t_{base} exceeds the CPU time quota is suboptimal, as it causes CPU throttling even under low workloads. To address this, it is necessary to adjust the CPU time quota. Typically, in most common resource configurations t_{base} is much less than the CPU time quota, allowing the application to function without encountering CPU throttling under low workloads.

However, even when t_{base} is significantly lower than the CPU time quota, under high workloads, this quota can be exceeded depending on the number of requests processed within a given CPU period. Theoretically, under high traffic conditions, CPU throttling is influenced by how rapidly the CPU time quota is consumed within a CPU period. Under this premise, the request rate plays a critical role in the CPU throttling process. Given that each request demands a specific CPU processing time, CPU usage also has a substantial impact on the extent of CPU throttling.

From an autoscaling standpoint, the primary objective is to scale the replicas proactively, ensuring that scaling occurs prior to the engagement of the CPU throttling mechanism, thereby preventing any negative impact on service response times particularly for deployments with $t_{base} << CPU\ time\ quota$. Consequently, accurately identifying the onset of CPU throttling is essential.

To investigate the behavior of CPU throttling in a real-world deployment in $t_{base} << CPU\ time\ quota$ cases, particularly in relation to varying incoming request rates and CPU usage patterns, a series of controlled experiments were conducted under different traffic scenarios. The web application previously discussed in Section 4.2.2 was redeployed within the same Kubernetes testbed environment, with CPU requests and limits configured at 0.35 millicores. In this context, the t_{base} was approximately 2 milliseconds, which

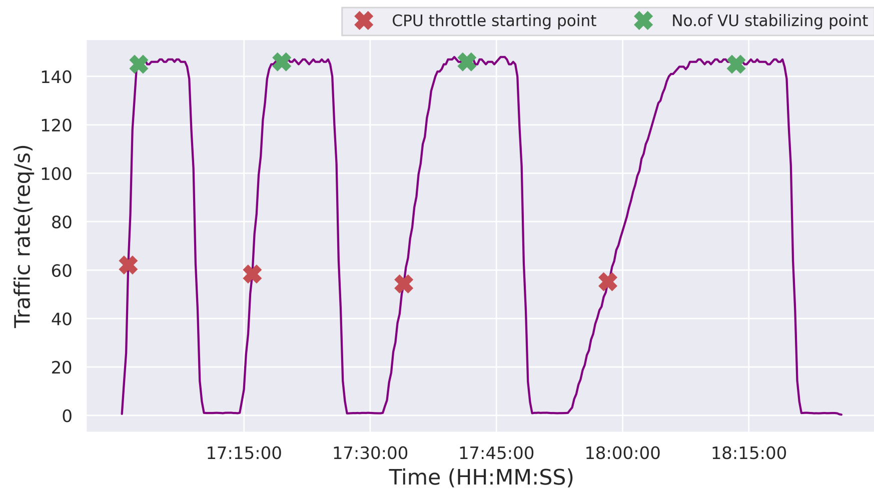
is considerably lower than the corresponding CPU time quota of 35 milliseconds. This t_{base} value is selected to enable high traffic loads, triggering CPU throttling only during periods of high CPU utilization. The experiment's goal was to incrementally increase the traffic directed to the web application exceed its maximum capacity. This approach allowed for close monitoring of the initiation of CPU throttling and an analysis of its correlation with traffic rate and CPU usage.

For the experiment, following traffic scenarios were introduced as a continuous traffic flow. In this context, the traffic generator instantiates Virtual Users (VUs) according to the predefined scenarios as follows:

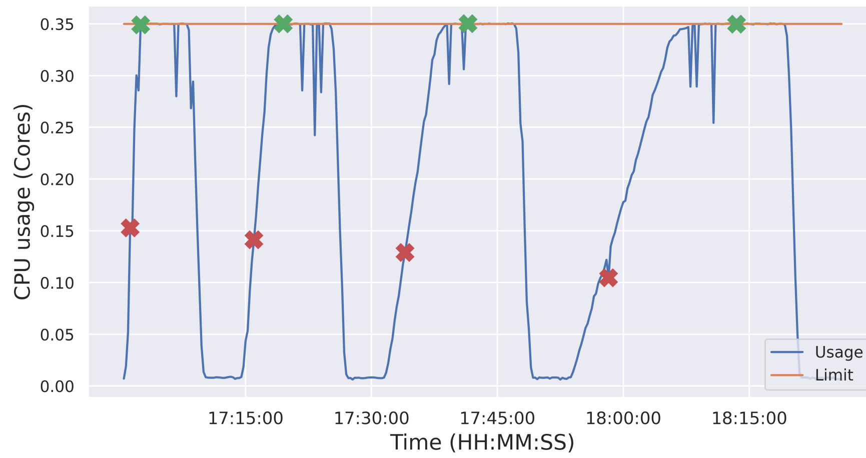
- **Scenario 1:** Reach 250 VUs in 2 minutes → maintain current traffic rate for 5 minutes → decrease to 1 VU in 2 minutes → maintain for current traffic rate 5 minutes
- **Scenario 2:** Reach 250 VUs in 5 minutes → maintain current traffic rate for 5 minutes → decrease to 1 VU in 2 minutes → maintain for current traffic rate 5 minutes
- **Scenario 3:** Reach 250 VUs in 10 minutes → maintain current traffic rate for 5 minutes → decrease to 1 VU in 2 minutes → maintain for current traffic rate 5 minutes
- **Scenario 4:** Reach 250 VUs in 20 minutes → maintain current traffic rate for 5 minutes → decrease to 1 VU in 2 minutes → maintain for current traffic rate 5 minutes

Each VU autonomously enters a continuous loop, initiating requests to the web application. A randomized interval, ranging from 0 to 2 seconds, is applied between successive transmissions to simulate variable request arrivals.

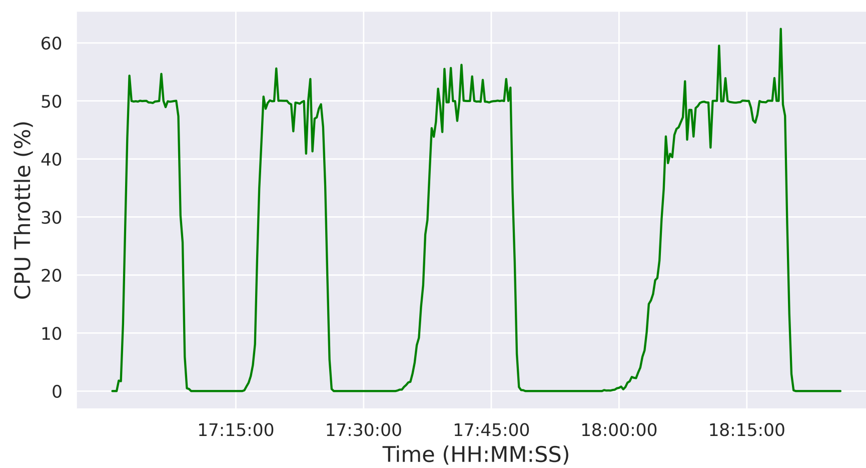
Data collected from the Kubernetes metrics APIs, including inbound request rate, pod CPU usage, and pod CPU throttling, is presented in Figure 6.3. In Kubernetes, detailed CPU throttling-related metrics, such as the number of CPU periods allocated, the number of throttled CPU periods, and the throttled CPU time, are typically available via Prometheus. To assess CPU throttling and determine when throttling is engaged, the CPU throttling values depicted in the plot are derived from the following equation 6.2, which considers both throttled CPU periods and the total CPU periods allocated to the pod over a 1-minute moving window. This metric indicates the percentage of CPU periods that were throttled relative to the total number of allocated CPU periods for the given cgroup.



(a) Incoming request rate



(b) CPU usage.



(c) CPU throttling

Figure 6.3 – Influence of incoming request rate, CPU usage on CPU throttling. (from [6]) (Copyright © 2024 IEEE)

$$CPU\ throttling = \frac{Throttled\ periods}{Total\ allocated\ CPU\ periods} \times 100\% \tag{6.2}$$

Figure 6.3a demonstrates that the incoming traffic rate peaked at 140 requests per second, while Figure 6.3b shows that CPU usage reached a maximum of 350 millicores under each scenario, as expected. Figure 6.3c illustrates the CPU throttling behavior observed across the different scenarios. In Scenario 1, with the highest VU initiation rate, the maximum incoming request rate was achieved earlier than in other Scenarios, with the time to reach this peak gradually increasing in subsequent Scenarios. Despite Scenario 1 demonstrated a rapid acceleration in request rate, it processed the fewest total requests during this acceleration phase. This outcome is attributed to the scenario achieving the peak request rate within a shorter time frame, thereby limiting the total number of requests processed.

An additional observation is that the time required to achieve the maximum request rate (time duration between request rate 0 to 140 in Figure 6.3b) in each scenario does not correspond to the VU initiation duration specified in the script (time duration between request rate 0 to green crosses marking in Figure 6.3b). As indicated by the green crosses marking, the stabilization point of VU initiation, the request rate in Scenario 4 stabilizes earlier than in Scenario 1, with Scenarios 2 and 3 exhibiting intermediate behavior. The stabilization of the request rate at 140 requests per second across all scenarios, despite continued VU initiation—particularly in Scenarios 2, 3, and 4—is attributed to the CPU reaching its limit, which prevents handling requests beyond this threshold.

CPU throttling analysis indicates that, in each scenario, the percentage of CPU periods throttled relative to the total allocated CPU periods stabilizes at 50% to keep CPU usage within the designated limits. However, this percentage may be influenced by the CFS process, necessitating further investigation to clarify its impact.

From an autoscaling perspective, the key observation is the initiation point of CPU throttling. As marked by the red crosses, the onset of CPU throttling varies across different traffic rates and CPU usage levels in each scenario. Scenarios with higher traffic rates tend to trigger CPU throttling earlier than those with lower traffic rates. Additionally, CPU throttling is initiated earlier at higher CPU usage levels within scenarios with higher traffic rates compared to those with lower CPU usage and traffic rates.

The relationship between incoming traffic rate, CPU usage, and CPU throttling is complex and challenging to precisely quantify. As a result, identifying the exact threshold at which the Linux kernel initiates throttling based on these parameters is difficult. This

complexity is further exacerbated by factors such as the CPU time quota allocated to the pod and the CFS process. The challenge of determining CPU throttling activation is particularly pronounced in environments with variable traffic patterns and multithreaded applications. It is important to note that this study does not address CPU throttling in the context of multithreaded applications.

6.3 CPU throttling aware triggering

The previously implemented autoscaling mechanism employed a seven-step CPU usage forecasting model to inform scaling decisions. This approach was based on the premise that predicting the need for scaling seven steps ahead could prevent CPU usage from surpassing its allocated capacity, thereby mitigating potential increases in service response times during scaling events. The timing for scaling-up was determined through an experimental, iterative process. However, this method proved sub-optimal, as the evaluation revealed that the proposed solution still encountered increases in service response time during scaling events.

The primary issue stemmed from the initiation of CPU throttling before CPU usage reached the predefined limit. Consequently, even though new replicas were in operation well in advance of the scale up delay, service response time had already deteriorated. To maintain service response time without degradation during scaling, the autoscaling mechanism must trigger the scale up process before CPU throttling is engaged. However, prioritizing QoS and initiating scaling earlier than the previously proposed solution could result in prolonged replica operation, thereby increasing operational costs. To preemptively trigger scaling before CPU throttling impacts service response time, it is crucial to predict when CPU throttling is expected to occur. However, as demonstrated by the experiments in Section 2, accurately modeling the CPU throttling process is challenging due to its intricate interactions with the allocated CPU quota, Linux kernel scheduling, incoming request rate, and pod CPU usage.

In the field of AI, deep learning models have shown exceptional capability in recognizing complex patterns within multivariate temporal data. These models have facilitated the generation of highly accurate forecasts, as demonstrated by their performance in prior solutions. Consequently, deep learning can be leveraged to predict the onset of CPU throttling by accounting for the intricate interactions among the aforementioned factors. To enhance the decision-making process within the autoscaling solution, a novel proac-

tive autoscaling methodology is proposed, incorporating a deep learning-based trigger mechanism. This approach aims to proactively determine the optimal timing for scaling decisions, thereby mitigating the impact of CPU throttling on service response time. The newly proposed autoscaling solution builds upon the foundation of the previous implementation and consists of three distinct subsystems.

The first subsystem, the CPU usage forecasting model, employs a deep learning-based time series forecasting model to predict future CPU usage behavior for a service deployed in a Kubernetes cluster, similar to the previous solution.

The second subsystem, the dynamic scaling model, calculates the required resources based on the forecasts generated by the deep learning model in the first subsystem. This subsystem includes slight modifications to the previous dynamic scaling model to accommodate the new trigger model.

The third subsystem, the trigger model, is a new addition that utilizes a deep learning-based forecasting model to predict CPU throttling events and trigger scaling-up processes accordingly. Detailed explanations of these subsystems and their interactions will be provided in the subsequent subsections.

6.3.1 CPU throttling forecasting

The objective of CPU throttling forecasting is to provide predictive insights into future CPU throttling behavior, enabling the autoscaling mechanism to preemptively scale up replicas before CPU throttling adversely affects service response times.

As demonstrated by the experiment in Section 6.2, the occurrence of CPU throttling is influenced by the rate of incoming requests and the CPU utilization of the pod. Consequently, three key input features were identified for the forecasting model: average inbound request rate, average CPU usage, and average CPU throttled seconds. The use of average values for these features is justified by the fact that CPU throttling occurs based on the conditions of individual pods. Given that Kubernetes utilizes a round-robin algorithm for traffic distribution among replicas, it is assumed that conditions among all replicas are uniformly distributed. The average values for all features were calculated based on the metrics retrieved per service. Specifically, to calculate the average CPU throttling among replicas, CPU throttled seconds per service metric was utilized.

The primary goal is to scale up replicas before CPU throttling occurs. However, the uniform behavior among replicas can be disrupted during scaling events. When the trigger model detects an increase in CPU throttling in advance, the dynamic scaling model

initiates the scaling process. Once the new replicas are operational, the round-robin load balancer begins distributing requests across both the existing and new replicas. To prevent CPU throttling in the existing replicas, it is essential to initiate scaling early, accounting for the scaling-up delay, which will be discussed later in this section.

From the perspective of the forecasting model, which relies on average metric values, these values are expected to decrease sharply once a new replica becomes operational after a scaling event. This decrease happens because, prior to scaling, the existing replicas have high average metric values, while the new replicas show lower metric values for a brief period until all replicas stabilize. Consequently, when calculating the overall average metric value, it can drop significantly. Nevertheless, the extent of the decrease in the average metric value is contingent upon the number of active replicas associated with each service. This abrupt change in input metrics can negatively affect the forecasting model's prediction accuracy. To mitigate this, scaling events should be included as a feature in the forecasting model. Therefore, current pod count metrics are integrated as a feature to improve predictive accuracy in such scenarios.

Given the proven effectiveness of the selected deep learning models (LSTM, GRU, and CNN) in multivariate time series forecasting, as shown by the total CPU usage forecasting results in Chapter 4, these models were similarly applied to evaluate the forecasting performance for average CPU throttling seconds in the trigger model.

For the selected forecasting models, the determination of optimal past and future prediction window sizes was critical. Specifically, establishing an appropriate prediction horizon is vital for generating actionable insights into the autoscaler mechanism. However, consistent with the approach detailed in Section 4.3.4, the past value window for CPU throttling was fixed at 10 steps. The selection of the forecasting horizon must account for the scale up delay, ensuring that when CPU throttling is detected, the autoscaler can initiate new replicas promptly. This allows the replicas to become operational before the anticipated increase in CPU throttling, thereby mitigating its impact on service response time. However, due to the data extraction granularity limitations, accurately measuring the scale up delay is challenging. For this experiment, the time required to provision a new replica in the target web application is shorter than the data collection interval of 15 seconds per step. Consequently, a single-step prediction model was selected.

For the model training process, a new dataset was constructed using selected features based on criteria similar to those outlined in Section 4.3.4. However, incorporating scaling events into the training data was essential. This inclusion was necessary because scaling

actions, triggered by the dynamic scaling model, can alter the input data for the forecasting model, potentially leading to inaccurate forecasts if not accounted for. Consequently, the training dataset needed to encompass scaling events to allow the model to learn how features behave during these operations. To address this requirement, the proactive scaling solution from previous work was leveraged during the data collection process. Since the autoscaling solution made several scaling decisions throughout the experiment, the dataset was ensured to capture relevant information on scaling events.

The new dataset was preprocessed and split into training and test sets, with a 60% and 40% allocation, respectively. The training dataset was utilized to train all models, and forecasting accuracy was evaluated using RMSE and MAE metrics. The detailed results of these evaluations are presented in a subsequent section. The model with the best performance was then selected as the CPU throttling prediction model for integration into the new trigger module.

6.3.2 Trigger module integration

To integrate the newly introduced trigger mechanism into the previously proposed autoscaling solution, it is required to modify the dynamic autoscaler model. The updated model delegates all scaling-up events to the trigger mechanism. Specifically, the autoscaler will initiate scaling-up replicas when a rise in predicted CPU throttling values is detected. scaling-down and no-scaling events will continue to be managed by the dynamic scaling model, maintaining consistency with the previously established solution.

In this configuration, the new autoscaling solution retrieves input data from monitoring services at 15-second intervals. The CPU throttling model predicts throttling values one step ahead. The CPU usage forecasting model, however, now estimates future CPU usage ten steps ahead, as opposed to the previous seven steps. The decision timing no longer depends on the number of forecasting steps, as scaling-up decisions are now initiated based on insights from the new trigger model.

The predicted CPU usage values are initially filtered through the scaling-down threshold to assess whether any value in the forecasted sequence exceeds this threshold. If any value surpasses the threshold, further analysis is required to determine whether the autoscaler should initiate a scale up or maintain the current state. If all values in the forecasted sequence are equal to or below the scaling-down threshold, the autoscaler will scale down by one pod at a time. Any value in the forecasted sequence that exceeds the scaling-down threshold is used to calculate the required number of replicas using equation

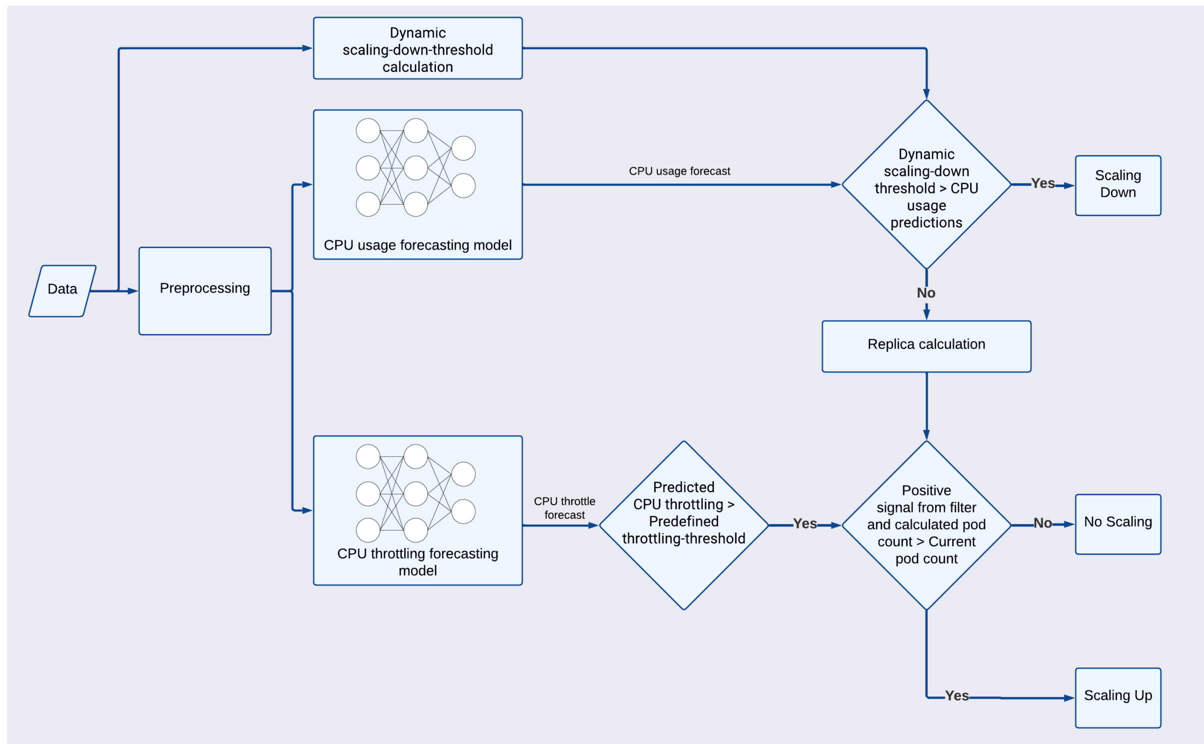


Figure 6.4 – Proposed CPU throttling aware autoscaling architecture. (from [6]) (Copyright © 2024 IEEE)

5.1, following the approach used in the previous solution. Once the required number of replicas is calculated, it is compared with the current replica count for that service. If the required and current replica counts are equal, no scaling action is taken. However, if the required replica count exceeds the current count, the autoscaler will wait for insights from the CPU throttling forecasting model before initiating a scale up action.

During the initial fine-tuning phase of the autoscaling solution, it was observed that very low levels of CPU throttling had an insignificant effect on service response time, making scale up actions in response to such low throttling levels unnecessary and likely to increase operational time. To address this, a filtering mechanism was implemented that applies a predefined threshold of 0.05 millicores to the CPU throttling values provided by the CPU throttling forecasting model, filtering out any throttling values below this threshold. A scaling-up decision will be triggered by any forecasted CPU throttling value that surpasses this threshold.

In the scaling-up decision process, two binary variables are considered, leading to several possible scenarios. If the calculated replica count exceeds the current replica count

and CPU throttling is expected to surpass the predefined threshold, a scaling-up action is required. This ensures that the solution scales up replicas with optimal timing, preventing CPU throttling from negatively impacting service response time.

However, if the calculated number of replicas exceeds the current number of replicas but CPU throttling remains below the threshold, this indicates that scaling-up is anticipated, but the expected workload not expected to caused any QoS degradation due to CPU throttling. In this case, no immediate scaling action is necessary, avoiding premature scaling.

On the other hand, if the expected number of replicas equals the current number of replicas while CPU throttling increases beyond the threshold, this may suggest that the CPU throttling is increasing due to an issue other than resource starvation, and therefore no scaling action is required. Such situations are rare and can be caused by issues like a software bug in older Linux kernels (bug 512ac999), which caused CPU throttling even when CPU usage was low [146]. Modern Linux kernels have already patched this problem.

Lastly, if the calculated replica count is equal to the current number of replicas and the predicted CPU throttling is below the predefined threshold, no scaling-up actions are required. The complete proposed autoscaling solution is illustrated in Figure 6.4.

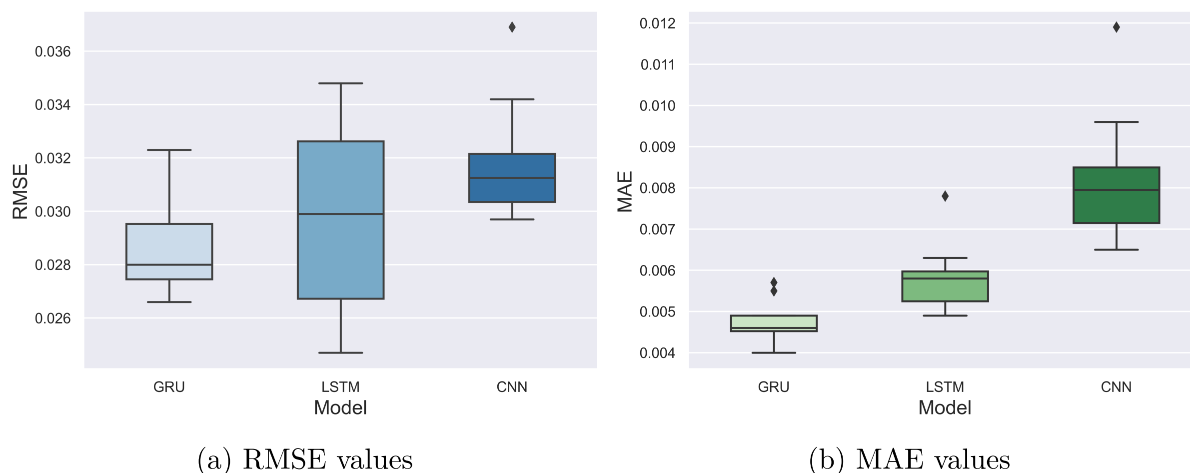
The newly proposed solution was deployed in a Kubernetes cluster within the same testbed and configuration, utilizing the same traffic pattern as the previous solution. The evaluation of the new autoscaling solution focused on service response time, replica count, and operational time to assess its effectiveness in balancing the cost-QoS trade-off. A comprehensive analysis of the results will be presented in the following section.

6.4 Results

6.4.1 Forecasting model evaluation

The evaluation of CPU throttling forecasting models was performed using a test dataset, where model performance was assessed by calculating the RMSE and MAE values. This evaluation was repeated ten times to ensure the robustness of the results. The distributions of the RMSE and MAE values across these iterations are illustrated in the box plots presented in Figures 6.5a and 6.5b.

Analysis of the median RMSE and MAE values revealed that the GRU model exhibited the lowest median values for both metrics. Additionally, the low variability in RMSE



(a) RMSE values

(b) MAE values

Figure 6.5 – RMSE and MAE values of CPU throttling forecasting models. (from [6]) (Copyright © 2024 IEEE)

and MAE values indicates that the GRU model's performance is consistently reliable. In contrast, the LSTM model demonstrated moderate median RMSE and MAE values but exhibited the highest variance in RMSE, suggesting less consistent performance. The CNN model displayed the highest median RMSE and MAE values, with moderate variability, yet it also produced significant outliers in both metrics, indicating instances of poor performance. Based on these findings, the GRU model outperformed the other models in predicting CPU throttle seconds and was selected for the autoscaling application.

6.4.2 Autoscaling solution evaluation

To evaluate the performance of the newly proposed autoscaling solution, critical metrics were analyzed, including the 95th percentile of service response time to represent QoS, as well as pod count and their operational durations to reflect operational costs. These results were benchmarked against the previous autoscaling solution and the Kubernetes HPA set at an 80% scaling threshold as shown in the Figure 6.6. The 80% threshold was chosen for the HPA because it offers a comparable profile in service response time, replica count, and operational time to both the previous and the newly proposed solutions.

Figure 6.6a presents the service response time profile and pod count graphs for the HPA configured with an 80% threshold. Throughout the experiment, only a single scaling-up event and a single scaling-down event were observed. Although CPU usage did not reach its defined limit during the scaling-up process, CPU throttling was triggered and

persisted throughout the start-up delay of the second pod. This throttling resulted in a substantial increase in service response time, with a peak reaching 195ms.

As shown in Figure 6.6b, the previous autoscaling solution, which relied on predicted CPU usage, effectively detected early increases in CPU demand and scaled up additional replicas to manage the increased workload. It also responded to short-term decreases in CPU usage by scaling-down replicas, thereby reducing pod count and operational time. However, despite these advantages, the solution experienced several service response time spikes during the scaling-up process, with a maximum peak of 50ms. This issue is primarily attributed to the difficulty of detecting CPU throttling solely through predicted CPU usage, even when extending the prediction horizon by 7 steps. Nevertheless, these response time spikes were considerably lower than those observed with the HPA configured at the 80% threshold.

The CPU throttle-aware autoscaling solution, as depicted in Figure 6.6c, demonstrated a significant improvement in reducing service response time, achieving a peak of only 24ms. This improvement is attributed to the solution's effective CPU throttle prediction, which enabled the initiation of early scale up actions, thereby minimizing the impact of CPU throttling on service response times. Additionally, the autoscaler avoided frequent scaling-down of replicas in response to short-term decreases in CPU usage, resulting in fewer scale down events compared to the previous solution.

Figure 6.7 illustrates the operational time of each pod across different approaches, highlighting variations in the duration for which the second pod was active. In all approaches, a maximum of two pods was deployed to handle the workload. The first pod remained operational throughout the entire experiment, making the total operational time dependent on the duration of the second pod's activity. In the HPA solution, the second pod maintained continuous operation for 119 minutes. By comparison, the previous solution, the second pod was active for a total of 104 minutes. Despite periods of brief scaling-down, the CPU throttling aware autoscaling solution sustained the second pod's operation for 119 minutes. Notably, the CPU throttle-aware autoscaling solution initiated the scaling-up of the second pod earlier than the other approaches. This behavior can be attributed to the CPU throttling mechanism, which activates before CPU usage reaches its threshold limits.

The results indicate that reducing service response time spikes typically requires an extended operational time due to early scale up. However, the new approach successfully achieved the lowest service response time spikes, even with an operational time comparable

(a) Kubernetes HPA 80%

(b) Previous autoscaling solution.

(c) CPU throttling aware autoscaling solution.

Figure 6.6 – Service response time during scaling. (from [6]) (Copyright © 2024 IEEE)

to the HPA solution. In light of the primary objective—to optimize the trade-off between cost and QoS, the proposed CPU throttle-aware autoscaling solution effectively minimized service response time spikes while incurring only a marginal increase in pod operational time, thereby outperforming other approaches.

6.5 Summary

This chapter delves into the CPU throttling process within the Linux kernel, particularly its impact on service response times during the scaling process. The focus is on understanding how CPU throttling occurs when CPU limits are defined, and how it affects the processing times of requests. The chapter also examines the relationship between incoming request rates, CPU usage, and the engagement of CPU throttling.

To address the negative impact of CPU throttling on service performance, a new autoscaling solution is proposed. This solution leverages deep learning models to predict and mitigate CPU throttling in real-time. The new solution was implemented and tested in

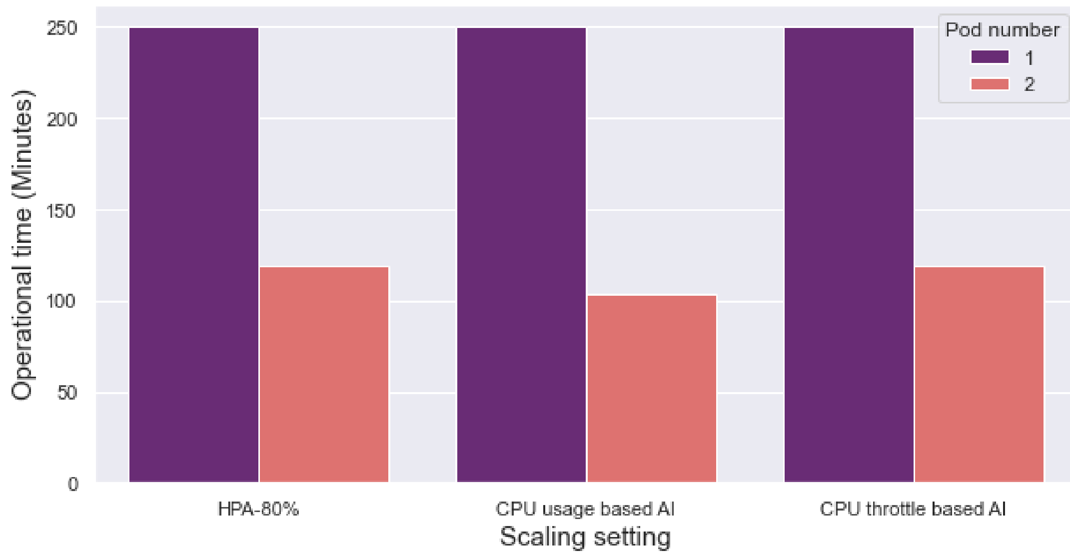


Figure 6.7 – Operational time comparison. (from [6]) (Copyright © 2024 IEEE)

a real-world Kubernetes environment, where it demonstrated an ability to reduce service response times during scaling, offering a better cost-QoS trade-off compared to previous autoscaling methods and Kubernetes HPA.

The research is limited to single-threaded applications with a single service, whereas real-world applications are often multi-threaded and involve multiple services. Expanding the research to encompass more complex applications and testing under heavy and intricate traffic patterns could improve its applicability.

Chapter 7

Conclusion

Summary

As 5G mobile networks increasingly adopt cloud-native architectures, the shift toward cloudification brings both opportunities and challenges. While it enhances service reliability, scalability, availability and lower cost, the complexity of managing these cloud-based systems grows significantly, particularly given the high traffic volumes and diverse use cases that demand tailored network configurations. Central to this complexity is the trade-off between ensuring high QoS and managing operational costs effectively. Without proper resource management, CSPs risk either over-provisioning, which drives up costs, or under-provisioning, which can degrade performance.

This thesis has addressed this critical cost-QoS trade-off by developing and evaluating a proactive autoscaling solution for cloud-native 5G NFs. The goal was to efficiently allocate resources to handle incoming workloads without compromising QoS or incurring unnecessary costs.

The first stage of the proposed solution introduced a novel approach to resource usage forecasting based on deep learning. By generating highly accurate long-term forecasts, this method provides critical insights for the subsequent autoscaling stage, allowing for better preparation against QoS degradation caused by scale up delays and minimizing the oscillations in scaling decisions due to fluctuating traffic patterns. The results demonstrated that this deep learning-based approach outperforms traditional single-variable forecasting methods, offering more accurate predictions that consider both system-level and application-level requirements and their correlations.

The second stage focused on translating these resource usage forecasts into precise autoscaling actions. By determining the appropriate times for scaling-up, scaling down, or maintaining current resource levels, the dynamic autoscaling solution, which leverages both static and dynamic thresholds, optimized resource utilization. This approach resulted in significant cost savings while maintaining high QoS levels, as shown by the comparative

analysis with existing solutions.

Recognizing the impact of CPU throttling in Kubernetes environments, particularly during scaling events, an enhancement to the autoscaling mechanism was proposed that specifically addresses this issue. By incorporating a new trigger mechanism based on deep learning forecasts, the solution effectively mitigates the negative effects of CPU throttling on QoS. The results indicated a notable reduction in QoS degradation, further balancing the cost-QoS trade-off.

The findings of this thesis contribute to a deeper understanding of the intricacies involved in managing cloud-native 5G networks, particularly in relation to the cost-QoS trade-off. The proposed solutions offer practical insights and methodologies that can be adopted by CSPs to enhance the efficiency of their network management practices.

In conclusion, the proactive autoscaling solutions developed in this thesis provide a robust framework for addressing the challenges of resource management in cloud-native 5G networks. By carefully balancing the trade-off between cost and QoS, these solutions pave the way for more sustainable and efficient network operations in the era of 5G and beyond.

Future directions

The findings of this research serve as a robust foundation for several potential avenues of exploration within the autoscaling domain in cloud-native 5G environments. While this study has provided insights into optimizing the cost-QoS trade-off for 5G networks deployed on public clouds under specific pricing models, there are multiple directions for future work that can further enhance the efficiency, sustainability, and security of autoscaling mechanisms.

One promising area for future research is the exploration of energy consumption and performance trade-offs in autoscaling, particularly from the perspective of cloud service providers or private cloud owners. While the focus has been on CSPs as customers, optimizing autoscaling strategies to conserve energy for cloud providers themselves remains an open challenge. For instance, future work could investigate methods for scheduling new replicas to minimize the number of operational nodes, either by removing idle nodes or putting them into sleep mode. This could lead to significant energy savings. However, this approach introduces potential challenges, such as performance degradation due to node-level CPU throttling and OOM processes when too many replicas are packed into

a single node. Addressing these challenges will require innovative solutions that carefully balance energy efficiency with maintaining acceptable performance levels.

Another important direction for future research is the consideration of inter-dependen -cies between NFs within 5G CN. While this research focused on resource usage forecasting for individual NFs, each service in the 5G CN involves multiple interconnected NFs. To address the complexities of such application architectures, Graph Neural Networks (GNNs) could be leveraged to perform time series forecasting that takes into account the correlations and interrelationships between NFs. This approach could lead to more accurate resource usage predictions and, consequently, more effective autoscaling. However, the dynamic nature of 5G networks, particularly with horizontal scaling where replicas are frequently added and removed, presents challenges in maintaining a stable and accurate graph representation. Exploring solutions to manage these dynamically changing graphs in real-time would be a valuable contribution to the field.

Additionally, future work could explore the integration of security mechanisms into autoscaling strategies, particularly in response to emerging threats such as Distributed Denial of Service (DDoS) attacks targeting cloud environments. These attacks are not necessarily designed to disrupt application performance but rather to inflict economic damage by artificially inflating resource usage, leading to unnecessary scaling. Developing autoscaling solutions capable of detecting and mitigating such attacks in real time could help balance the cost-QoS trade-off more effectively and protect cloud deployments from economic exploitation. This line of research would be crucial in enhancing the resilience and cost-effectiveness of autoscaling in cloud-native 5G environments.

In summary, future work can build upon this research by exploring energy-efficient autoscaling strategies, leveraging advanced forecasting techniques like GNNs to account for NF inter-dependencies, and integrating security measures to protect against cost-targeted attacks. We believe these research directions hold significant potential to advance the autoscaling domain in cloud-native 5G networks, addressing both current challenges and emerging threats.

PUBLICATIONS

1. M. P. J. Kuranage, L. Nuaymi, A. Bouabdallah, T. Ferrandiz and P. Bertin, "Deep learning based resource forecasting for 5G core network scaling in Kubernetes environment," IEEE 8th International Conference on Network Softwarization (NetSoft), 2022.
2. M. P. Jayasuriya Kuranage, E. Hanser, L. Nuaymi, A. Bouabdallah, P. Bertin and A. Al-Dulaimi, "AI-assisted proactive scaling solution for CNFs deployed in Kubernetes," IEEE 12th International Conference on Cloud Networking (CloudNet), 2023.
3. M. P. Jayasuriya Kuranage, E. Hanser, A. Bouabdallah, L. Nuaymi and P. Bertin, "CPU Throttling-Aware AI-Based Autoscaling for Kubernetes," IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), 2024.

BIBLIOGRAPHY

[1] Ericsson, « Mobility Report », Tech. Rep., Jun. 2024. [Online]. Available: https://www.ericsson.com/en/reports-and-papers/mobility-report/ (visited on 08/13/2024).

[2] Flexera, « Essential Strategies for Managing Cloud Costs », Tech. Rep., 2019. [Online]. Available: https://info.flexera.com/CM-WP-Essential-Strategies-for-Managing-Cloud-Costs (visited on 08/13/2024).

[3] D. Ramsay, « Autonomous networks: exploring the evolution from level 0 to level 5 », Tech. Rep., Dec. 2021. [Online]. Available: https://inform.tmforum.org/research-and-analysis/reports/autonomous-networks-exploring-the-evolution-from-level-0-to-level-5 (visited on 08/25/2024).

[4] M. P. Jayasuriya Kuranage, L. Nuaymi, A. Bouabdallah, *et al.*, « Deep learning based resource forecasting for 5G core network scaling in Kubernetes environment », in *IEEE 8th International Conference on Network Softwarization (NetSoft)*, Jun. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9844056.

[5] M. P. Jayasuriya Kuranage, E. Hanser, L. Nuaymi, *et al.*, « AI-assisted proactive scaling solution for CNFs deployed in Kubernetes », in *IEEE 12th International Conference on Cloud Networking (CloudNet)*, Nov. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10490067.

[6] M. P. Jayasuriya Kuranage, E. Hanser, A. Bouabdallah, *et al.*, « CPU throttling-aware AI-based autoscaling for Kubernetes », in *IEEE 35th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, Sep. 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10817283.

[7] A. Basta, A. Blenk, K. Hoffmann, *et al.*, « Towards a Cost Optimal Design for a 5G Mobile Core Network Based on SDN and NFV », *IEEE Transactions on Network and Service Management*, Dec. 2017. [Online]. Available: https://ieeexplore.ieee.org/document/7994617.

[8] N. Li, X. Xu, Q. Sun, *et al.*, « Transforming the 5G RAN With Innovation: The Confluence of Cloud Native and Intelligence », *IEEE Access*, 2023. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/10006796`.

[9] S. D. A. Shah, M. A. Gregory, and S. Li, « Cloud-Native Network Slicing Using Software Defined Networking Based Multi-Access Edge Computing: A Survey », *IEEE Access*, 2021. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/9317860`.

[10] 3GPP, *System architecture for the 5G System (5GS), Release 15*, TS 23.501, 2003. [Online]. Available: `https://portal.3gpp.org/`.

[11] G. Mayer, « RESTful APIs for the 5G Service Based Architecture », *Journal of ICT Standardization*, 2018. [Online]. Available: `https://ieeexplore.ieee.org/document/10258072`.

[12] S. Imadali and A. Bousselmi, « Cloud Native 5G Virtual Network Functions: Design Principles and Use Cases », in *IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, Nov. 2018. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/8567377`.

[13] K. Trantzas, C. Tranoris, S. Denazis, *et al.*, « An automated CI/CD process for testing and deployment of Network Applications over 5G infrastructure », in *IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, Sep. 2021. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/9647628`.

[14] E. Guttman and I. Ali, « Path to 5G: A Control Plane Perspective », *Journal of ICT Standardization*, 2018. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/10258071`.

[15] ETSI, *Network Functions Virtualisation (NFV); Management and Orchestration, v1*, GS NFV-MAN 001, Dec. 2014. [Online]. Available: `https://www.etsi.org/deliver/etsi_gs/nfv-man/`.

[16] S. Yrjölä, P. Ahokangas, and M. Matinmikko-Blue, « Novel Context and Platform Driven Business Models via 5G Networks », in *IEEE 29th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, Sep. 2018. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/8580819`.

[17] NGMN, *Description of Network Slicing Concept,v1.0.8*, Sep. 2016. [Online]. Available: https://www.ngmn.org/.

[18] M. Chahbar, G. Diaz, A. Dandoush, *et al.*, « A Comprehensive Survey on the E2E 5G Network Slicing Model », *IEEE Transactions on Network and Service Management*, Mar. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9295415.

[19] L. U. Khan, I. Yaqoob, N. H. Tran, *et al.*, « Network Slicing: Recent Advances, Taxonomy, Requirements, and Open Research Challenges », *IEEE Access*, 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9003208.

[20] K. Kozłowski, S. Kukliński, and L. Tomaszewski, « Open issues in network slicing », in *9th International Conference on the Network of the Future (NOF)*, Nov. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8598130.

[21] X. Foukas, G. Patounas, A. Elmokashfi, *et al.*, « Network Slicing in 5G: Survey and Challenges », *IEEE Communications Magazine*, May 2017. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7926923.

[22] H. Zhang, N. Liu, X. Chu, *et al.*, « Network Slicing Based 5G and Future Mobile Networks: Mobility, Resource Management, and Challenges », *IEEE Communications Magazine*, Aug. 2017. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8004168.

[23] F. Z. Yousaf, M. Bredel, S. Schaller, *et al.*, « NFV and SDN—Key Technology Enablers for 5G Networks », *IEEE Journal on Selected Areas in Communications*, Nov. 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8060513.

[24] A. de la Oliva, X. Li, X. Costa-Perez, *et al.*, « 5G-TRANSFORMER: Slicing and Orchestrating Transport Networks for Industry Verticals », *IEEE Communications Magazine*, Aug. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8436050.

[25] A. Kaloxylos, A. Gavras, D. Camps Mur, *et al.*, « AI and ML – Enablers for Beyond 5G Networks », Zenodo, Tech. Rep., Dec. 2020. [Online]. Available: https://zenodo.org/records/4299895 (visited on 07/25/2024).

[26] M. Ryder and C. Downs, « Rethinking reflective practice: John Boyd's OODA loop as an alternative to Kolb », *The International Journal of Management Education*, Nov. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1472811722001057.

[27] J. Kephart and D. Chess, « The vision of autonomic computing », *Computer*, Jan. 2003. [Online]. Available: https://ieeexplore.ieee.org/document/1160055.

[28] 3GPP, *Telecommunication management; Self-Organizing Networks (SON); Concepts and requirements, Release 8*, TS 32.500. [Online]. Available: https://portal.3gpp.org/.

[29] P. T. Endo, M. S. Batista, G. E. Gonçalves, *et al.*, « Self-organizing strategies for resource management in Cloud Computing: State-of-the-art and challenges », in *2nd IEEE Latin American Conference on Cloud Computing and Communications*, Dec. 2013. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6842215.

[30] N. Marchetti, N. R. Prasad, J. Johansson, *et al.*, « Self-Organizing Networks: State-of-the-art, challenges and perspectives », in *8th International Conference on Communications*, Jun. 2010. [Online]. Available: https://ieeexplore.ieee.org/document/5509022.

[31] E. Coronado, R. Behravesh, T. Subramanya, *et al.*, « Zero Touch Management: A Survey of Network Automation Solutions for 5G and 6G Networks », *IEEE Communications Surveys & Tutorials*, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9913206.

[32] ETSI, *Zero-touch network and Service Management (ZSM); Requirements based on documented scenarios, V1.1.1*, GS ZSM 001, Oct. 2019. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/ZSM/001_099/001/.

[33] ETSI, *Zero-touch network and Service Management (ZSM); Reference Architecture, V1.1.1*, GS ZSM 002, Aug. 2019. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/ZSM/001_099/002/.

[34] C. Benzaid and T. Taleb, « AI-Driven Zero Touch Network and Service Management in 5G and Beyond: Challenges and Research Directions », *IEEE Network*, Mar. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8994961.

[35] M. Liyanage, Q.-V. Pham, K. Dev, *et al.*, « A survey on Zero touch network and Service Management (ZSM) for 5G and beyond networks », *Journal of Network and Computer Applications*, Jul. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804522000297.

[36] ONAP, *Open Network Automation Platform.* [Online]. Available: https://www.onap.org/ (visited on 07/26/2024).

[37] R. Rokui, H. Yu, L. Deng, *et al.*, « A Standards-Based, Model-Driven Solution for 5G Transport Slice Automation and Assurance », in *6th IEEE Conference on Network Softwarization (NetSoft)*, Jun. 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9165451.

[38] ETSI, *Zero-touch network and Service Management (ZSM); Closed-Loop Automation; Part 1: Enablers, V1.1.1*, GS ZSM 009-1, Jun. 2021. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/ZSM/001_099//.

[39] ETSI, *Zero-touch network and Service Management (ZSM); Closed-Loop Automation; Part 2: Solutions for automation of E2E service and network management use cases, V1.1.1*, GS ZSM 009-2, Jun. 2022. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/ZSM/001_099/.

[40] ETSI, *Zero-touch network and Service Management (ZSM); Closed-Loop Automation; Part 3: Advanced topics, V1.1.1*, GR ZSM 009-3, Aug. 2023. [Online]. Available: https://www.etsi.org/deliver/etsi_gr/ZSM/001_099/.

[41] G. Lin, D. Fu, J. Zhu, *et al.*, « Cloud Computing: IT as a Service », *IT Professional*, Mar. 2009. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4804041.

[42] C. Gong, J. Liu, Q. Zhang, *et al.*, « The Characteristics of Cloud Computing », in *39th International Conference on Parallel Processing Workshops*, Sep. 2010. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5599083.

[43] Google, *What are the different types of cloud computing?* [Online]. Available: https://cloud.google.com/discover/types-of-cloud-computing (visited on 07/26/2024).

[44] Google, *Google Cloud Platform.* [Online]. Available: https://cloud.google.com/ (visited on 09/08/2024).

[45] Microsoft, *Microsoft Azure*. [Online]. Available: https://azure.microsoft.com/ (visited on 07/26/2024).

[46] Amazon, *Amazon Web Services (AWS)*. [Online]. Available: https://aws.amazon.com/ (visited on 07/26/2024).

[47] Y. Jadeja and K. Modi, « Cloud computing - concepts, architecture and challenges », in *International Conference on Computing, Electronics and Electrical Technologies (ICCEET)*, Mar. 2012. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6203873.

[48] Google, *PaaS vs IaaS vs SaaS: What's the difference?* [Online]. Available: https://cloud.google.com/learn/paas-vs-iaas-vs-saas (visited on 07/23/2024).

[49] C. Wu, R. Buyya, and K. Ramamohanarao, « Cloud Pricing Models: Taxonomy, Survey, and Interdisciplinary Challenges », *ACM Comput. Surv.*, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3342103.

[50] S. Kansal, G. Singh, H. Kumar, *et al.*, « Pricing Models in Cloud Computing », in *International Conference on Information and Communication Technology for Competitive Strategies*, Oct. 2014. [Online]. Available: https://doi.org/10.1145/2677855.2677888.

[51] S.-H. Chun and B.-S. Choi, « Service models and pricing schemes for cloud computing », *Cluster Computing*, Jun. 2014. [Online]. Available: https://doi.org/10.1007/s10586-013-0296-1.

[52] Google, *Google Kubernetes Engine (GKE)*. [Online]. Available: https://cloud.google.com/kubernetes-engine (visited on 07/28/2024).

[53] Amazon, *Elastic Kubernetes Service*. [Online]. Available: https://aws.amazon.com/eks/ (visited on 07/28/2024).

[54] Kubernetes, *Production-Grade Container Orchestration*. [Online]. Available: https://kubernetes.io/ (visited on 07/28/2024).

[55] A. Verma, L. Pedrosa, M. Korupolu, *et al.*, « Large-scale cluster management at Google with Borg », in *10th European Conference on Computer Systems*, Apr. 2015. [Online]. Available: https://dl.acm.org/doi/10.1145/2741948.2741964.

[56] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, *et al.*, « Omega: flexible, scalable schedulers for large compute clusters », in *8th ACM European Conference on Computer Systems*, Apr. 2013. [Online]. Available: https://dl.acm.org/doi/10.1145/2465351.2465386.

[57] CNCF, *Cloud Native Computing Foundation*. [Online]. Available: https://www.cncf.io/ (visited on 07/28/2024).

[58] *The Linux Foundation*. [Online]. Available: https://www.linuxfoundation.org (visited on 07/28/2024).

[59] OpenStack, *Open Source Cloud Computing Infrastructure*. [Online]. Available: https://www.openstack.org/ (visited on 09/08/2024).

[60] T.-T. Nguyen, Y.-J. Yeom, T. Kim, *et al.*, « Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration », *Sensors*, Jan. 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/16/4621.

[61] Docker, *Accelerated Container Application Development*, May 2022. [Online]. Available: https://www.docker.com/ (visited on 07/28/2024).

[62] *Containerd*. [Online]. Available: https://containerd.io/ (visited on 07/28/2024).

[63] *CRI-O*. [Online]. Available: https://cri-o.io/ (visited on 07/28/2024).

[64] H. D. Trinh, N. Bui, J. Widmer, *et al.*, « Analysis and modeling of mobile traffic using real traces », in *IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, Oct. 2017. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8292200.

[65] Kubernetes, *Horizontal Pod Autoscaling*. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (visited on 07/30/2024).

[66] Google, *Vertical Pod autoscaling*. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler (visited on 07/30/2024).

[67] Kubernetes, *Cluster Autoscaling*. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/cluster-autoscaling/ (visited on 07/30/2024).

[68] F. L. Ferraris, D. Franceschelli, M. P. Gioiosa, *et al.*, « Evaluating the Auto Scaling Performance of Flexiscale and Amazon EC2 Clouds », in *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Sep. 2012. [Online]. Available: `https://ieeexplore.ieee.org/document/6481061`.

[69] Google, *Autopilot overview*. [Online]. Available: `https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview` (visited on 04/21/2024).

[70] K. Rzadca, P. Findeisen, J. Swiderski, *et al.*, « Autopilot: workload autoscaling at Google », in *15th European Conference on Computer Systems*, Apr. 2020. [Online]. Available: `https://dl.acm.org/doi/10.1145/3342195.3387524`.

[71] Y. Zhang, Y. Yu, W. Wang, *et al.*, « Workload consolidation in alibaba clusters: the good, the bad, and the ugly », in *13th Symposium on Cloud Computing*, Nov. 2022. [Online]. Available: `https://dl.acm.org/doi/10.1145/3542929.3563465`.

[72] C. Qu, R. N. Calheiros, and R. Buyya, « Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey », *ACM Computing Surveys*, Jul. 2018. [Online]. Available: `https://dl.acm.org/doi/10.1145/3148149`.

[73] T. Chen, R. Bahsoon, and X. Yao, « A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems », *ACM Computing Surveys*, Jun. 2018. [Online]. Available: `https://dl.acm.org/doi/10.1145/3190507`.

[74] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, « A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments », *Journal of Grid Computing*, Dec. 2014. [Online]. Available: `https://doi.org/10.1007/s10723-014-9314-7`.

[75] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, *et al.*, « Elasticity in Cloud Computing: State of the Art and Research Challenges », *IEEE Transactions on Services Computing*, Mar. 2018. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/7937885`.

[76] C. Barna, M. Fokaefs, M. Litoiu, *et al.*, « Cloud Adaptation with Control Theory in Industrial Clouds », in *IEEE International Conference on Cloud Engineering Workshop (IC2EW)*, Apr. 2016. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/7527853`.

[77] D. Niu, H. Xu, and B. Li, « Resource Auto-Scaling and Sparse Content Replication for Video Storage Systems », *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, Nov. 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3079045.

[78] I. Alawe, A. Ksentini, Y. Hadjadj-Aoul, *et al.*, « Improving Traffic Forecasting for 5G Core Network Scalability: A Machine Learning Approach », *IEEE Network*, Nov. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8553653.

[79] M. Mekki, B. Brik, A. Ksentini, *et al.*, « XAI-Enabled Fine Granular Vertical Resources Autoscaler », in *IEEE 9th International Conference on Network Softwarization (NetSoft)*, Jun. 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10175438.

[80] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, *et al.*, « Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER », in *IEEE 10th International Conference on Cloud Computing (CLOUD)*, Jun. 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8030623.

[81] W. A. Hanafy, Q. Liang, N. Bashir, *et al.*, « CarbonScaler: Leveraging Cloud Workload Elasticity for Optimizing Carbon-Efficiency », *ACM on Measurement and Analysis of Computing Systems*, Dec. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3626788.

[82] M. A. Tamiru, J. Tordsson, E. Elmroth, *et al.*, « An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud », in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec. 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9407312.

[83] M. Wang, D. Zhang, and B. Wu, « A Cluster Autoscaler Based on Multiple Node Types in Kubernetes », in *IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, Jun. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9084706.

[84] Q. Wu, J. Yu, L. Lu, *et al.*, « Dynamically Adjusting Scale of a Kubernetes Cluster under QoS Guarantee », in *IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, Dec. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8975761.

[85] T. V. Doan, G. T. Nguyen, H. Salah, *et al.*, « Containers vs Virtual Machines: Choosing the Right Virtualization Technology for Mobile Edge Cloud », in *IEEE 2nd 5G World Forum (5GWF)*, Sep. 2019. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/8911715`.

[86] I. Sarrigiannis, K. Ramantas, E. Kartsakli, *et al.*, « Online VNF Lifecycle Management in an MEC-Enabled 5G IoT Architecture », *IEEE Internet of Things Journal*, May 2020. [Online]. Available: `https://ieeexplore.ieee.org/document/8854289`.

[87] L. M. Ruíz, P. P. Pueyo, J. Mateo-Fornés, *et al.*, « Autoscaling Pods on an On-Premise Kubernetes Infrastructure QoS-Aware », *IEEE Access*, 2022. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/9732997`.

[88] E. Karypiadis, A. Nikolakopoulos, A. Marinakis, *et al.*, « SCAL-E: An Auto Scaling Agent for Optimum Big Data Load Balancing in Kubernetes Environments », in *International Conference on Computer, Information and Telecommunication Systems (CITS)*, Jul. 2022. [Online]. Available: `https://ieeexplore.ieee.org/document/9832990`.

[89] L. H. Phuc, L.-A. Phan, and T. Kim, « Traffic-Aware Horizontal Pod Autoscaler in Kubernetes-Based Edge Computing Infrastructure », *IEEE Access*, 2022. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/9709810`.

[90] F. Tonini, C. Natalino, L. Wosinska, *et al.*, « Demonstrating the Benefits of Service-Aware Pod Autoscaling with Shared Resources », in *IEEE 9th International Conference on Network Softwarization (NetSoft)*, Jun. 2023. [Online]. Available: `https://ieeexplore.ieee.org/document/10175413`.

[91] G. Rattihalli, M. Govindaraju, H. Lu, *et al.*, « Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes », in *IEEE 12th International Conference on Cloud Computing (CLOUD)*, Jul. 2019. [Online]. Available: `https://ieeexplore.ieee.org/document/8814504`.

[92] C. H. T. Arteaga, F. B. Anacona, K. T. T. Ortega, *et al.*, « A Scaling Mechanism for an Evolved Packet Core Based on Network Functions Virtualization », *IEEE Transactions on Network and Service Management*, Jun. 2020. [Online]. Available: `https://ieeexplore.ieee.org/document/8941016`.

[93] T. V. K. Buyakar, A. K. Rangisetti, A. A. Franklin, *et al.*, « Auto scaling of data plane VNFs in 5G networks », in *13th International Conference on Network and Service Management (CNSM)*, Nov. 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8256027.

[94] LXC, *Linux Containers*. [Online]. Available: https://linuxcontainers.org/ (visited on 08/14/2024).

[95] V.-G. Nguyen, K.-J. Grinnemo, J. Taheri, *et al.*, « On Auto-scaling and Load Balancing for User-plane Gateways in a Softwarized 5G Network », in *17th International Conference on Network and Service Management (CNSM)*, Oct. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9615536.

[96] Y. Ren, T. Phung-Duc, Y.-K. Liu, *et al.*, « ASA: Adaptive VNF Scaling Algorithm for 5G Mobile Networks », in *IEEE 7th International Conference on Cloud Networking (CloudNet)*, Oct. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8549542.

[97] J. Tong, M. Wei, M. Pan, *et al.*, « A Holistic Auto-Scaling Algorithm for Multi-Service Applications Based on Balanced Queuing Network », in *IEEE International Conference on Web Services (ICWS)*, Sep. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9590244.

[98] Z. Ding and Q. Huang, « COPA: A Combined Autoscaling Method for Kubernetes », in *IEEE International Conference on Web Services (ICWS)*, Sep. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9590262.

[99] J. Prados-Garzon, A. Laghrissi, M. Bagaa, *et al.*, « A Queuing Based Dynamic Auto Scaling Algorithm for the LTE EPC Control Plane », in *IEEE Global Communications Conference (GLOBECOM)*, Dec. 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8648023.

[100] S. Burroughs, H. Dickel, M. van Zijl, *et al.*, « Towards Autoscaling with Guarantees on Kubernetes Clusters », in *IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion*, Sep. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9599238.

[101] S. Bhandari, M. Patrou, N. Chahal, *et al.*, « Supervisory Event Loop-based Autoscaling of Node.js Deployments », in *IEEE International Conference on High

Performance Big Data and Intelligent Systems (HDIS), Dec. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9991325.

[102] S. Gong, B. Yin, and K.-y. Cai, « An Adaptive PID Control for QoS Management in Cloud Computing System », in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8539182.

[103] F.-H. Tseng, M.-S. Tsai, C.-W. Tseng, *et al.*, « A Lightweight Autoscaling Mechanism for Fog Computing in Industrial Applications », *IEEE Transactions on Industrial Informatics*, Oct. 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8272512.

[104] P. Benedetti, M. Femminella, G. Reali, *et al.*, « Reinforcement Learning Applicability for Resource-Based Auto-scaling in Serverless Edge Applications », in *IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, Mar. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9767437.

[105] C. Ayimba, P. Casari, and V. Mancuso, « Adaptive Resource Provisioning based on Application State », in *IEEE International Conference on Computing, Networking and Communications (ICNC)*, Feb. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8685605/.

[106] C. Ayimba, P. Casari, and V. Mancuso, « SQLR: Short-Term Memory Q-Learning for Elastic Provisioning », *IEEE Transactions on Network and Service Management*, Jun. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9416291.

[107] Z. Xiao and S. Hu, « DScaler: A Horizontal Autoscaler of Microservice Based on Deep Reinforcement Learning », in *IEEE 23rd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Sep. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9919994.

[108] A. A. Khaleq and I. Ra, « Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications », *IEEE Access*, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9361549.

[109] L. Lin, L. Pan, and S. Liu, « Learning to make auto-scaling decisions with heterogeneous spot and on-demand instances via reinforcement learning », *Information Sciences*, Oct. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025522011902.

[110] S. Vakilinia, C. Truchan, J. Kempf, *et al.*, « Automated Enforcement of SLA for Cloud Services », in *IEEE 11th International Conference on Cloud Computing (CLOUD)*, Jul. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8457782.

[111] Y. Zhao and A. Uta, « Tiny Autoscalers for Tiny Workloads: Dynamic CPU Allocation for Serverless Functions », in *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9825916 (visited on 02/07/2024).

[112] H. Zhao, H. Lim, M. Hanif, *et al.*, « Predictive Container Auto-Scaling for Cloud-Native Applications », in *International Conference on Information and Communication Technology Convergence (ICTC)*, Oct. 2019. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8939932.

[113] T. Wang, S. Ferlin, and M. Chiesa, « Predicting CPU usage for proactive autoscaling », in *1st Workshop on Machine Learning and Systems*, Apr. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3437984.3458831.

[114] A. Mudvari, N. Makris, and L. Tassiulas, « ML-driven scaling of 5G Cloud-Native RANs », in *IEEE Global Communications Conference (GLOBECOM)*, Dec. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9685874.

[115] N.-M. Dang-Quang and M. Yoo, « Deep Learning-Based Autoscaling Using Bidirectional Long Short-Term Memory for Kubernetes », *Applied Sciences*, Jan. 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/9/3835.

[116] E. H. Beni, E. Truyen, B. Lagaisse, *et al.*, « Reducing cold starts during elastic scaling of containers in kubernetes », in *36th Annual ACM Symposium on Applied Computing*, Apr. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3412841.3441887.

[117] ONF, *SD-Core - documentation, v 1.3*. [Online]. Available: https://docs.sd-core.opennetworking.org/master/release/1.3.html (visited on 06/05/2024).

[118] ONF, *Open Networking Foundation*. [Online]. Available: https://opennetworking.org/ (visited on 06/05/2024).

[119] Prometheus, *Monitoring system & time series database*. [Online]. Available: https://prometheus.io/ (visited on 06/05/2024).

[120] Grafana, *The open observability platform*. [Online]. Available: https://grafana.com/ (visited on 06/05/2024).

[121] A. Güngör, *UERANSIM*, Jun. 2024. [Online]. Available: https://github.com/aligungr/UERANSIM (visited on 06/05/2024).

[122] 3GPP, *Procedures for the 5G System (5GS), Release 15*, TS 23.502, 2016. [Online]. Available: https://portal.3gpp.org/.

[123] E. Goshi, M. Jarschel, R. Pries, *et al.*, « Investigating Inter-NF Dependencies in Cloud-Native 5G Core Networks », in *17th International Conference on Network and Service Management (CNSM)*, Oct. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9615565.

[124] R. Makhlouf, « Cloudy transaction costs: a dive into cloud computing economics », *Journal of Cloud Computing*, Jan. 2020. [Online]. Available: https://doi.org/10.1186/s13677-019-0149-4.

[125] L. Toka, G. Dobreff, B. Fodor, *et al.*, « Machine Learning-Based Scaling Management for Kubernetes Edge Clusters », *IEEE Transactions on Network and Service Management*, Mar. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9328525.

[126] T. Subramanya and R. Riggio, « Centralized and Federated Learning for Predictive VNF Autoscaling in Multi-Domain 5G Networks and Beyond », *IEEE Transactions on Network and Service Management*, Mar. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9319704.

[127] S. P. Sone, J. J. Lehtomäki, and Z. Khan, « Wireless Traffic Usage Forecasting Using Real Enterprise Network Data: Analysis and Methods », *IEEE Open Journal of the Communications Society*, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9108216.

[128] A. Kumar Dubey, A. Kumar, V. García-Díaz, *et al.*, « Study and analysis of SARIMA and LSTM in forecasting time series data », *Sustainable Energy Technologies and Assessments*, Oct. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2213138821004847.

[129] S. Ouhame and Y. Hadi, « Multivariate workload prediction using Vector Autoregressive and Stacked LSTM models », in *New Challenges in Data Sciences: Acts of the Second Conference of the Moroccan Classification Society*, Mar. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3314074.3314084.

[130] S. Rinchen, A. Yassine, K. Schwartzentruber, *et al.*, « Integrating Small Scale Green Energy into Smart Grids: Prediction for Peak Load Reduction », in *International Conference on Computer and Applications (ICCA)*, Aug. 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8460222.

[131] T. Xie and J. Ding, « Forecasting with Multiple Seasonality », in *IEEE International Conference on Big Data (Big Data)*, Dec. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9378072.

[132] F. V. Atabay, R. M. Pagkalinawan, S. D. Pajarillo, *et al.*, « Multivariate Time Series Forecasting using ARIMAX, SARIMAX, and RNN-based Deep Learning Models on Electricity Consumption », in *3rd International Informatics and Software Engineering Conference (IISEC)*, Dec. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9998301.

[133] T. Italia, *Telecommunications - SMS, Call, Internet - TN*, Feb. 2020. [Online]. Available: https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/QLCABU (visited on 02/04/2024).

[134] KongHQ, *Kong Gateway*. [Online]. Available: https://docs.konghq.com (visited on 06/06/2024).

[135] X. Wu, B. Xiang, H. Lu, *et al.*, « Optimizing Recurrent Neural Networks: A Study on Gradient Normalization of Weights for Enhanced Training Efficiency », *Applied Sciences*, Jan. 2024. [Online]. Available: https://www.mdpi.com/2076-3417/14/15/6578.

[136] V. Gupta and R. Hewett, « Adaptive Normalization in Streaming Data », in *3rd International Conference on Big Data Research*, Jan. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3372454.3372466.

[137] Y. Shynkevich, T. M. McGinnity, S. A. Coleman, *et al.*, « Forecasting price movements using technical indicators: Investigating the impact of varying input window length », *Neurocomputing*, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231217311074.

[138] N. Nguyen and T. Kim, « Toward Highly Scalable Load Balancing in Kubernetes Clusters », *IEEE Communications Magazine*, Jul. 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9161999.

[139] A. Renault, A. Bondu, A. Cornuéjols, *et al.*, *Early Classification of Time Series: Taxonomy and Benchmark*, Jul. 2024. [Online]. Available: http://arxiv.org/abs/2406.18332 (visited on 08/13/2024).

[140] A. Bondu, Y. Achenchabe, A. Bifet, *et al.*, « Open challenges for Machine Learning based Early Decision-Making research », *SIGKDD Explor. Newsl.*, Dec. 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3575637.3575643.

[141] P.-E. Zafar, Y. Achenchabe, A. Bondu, *et al.*, « Early Classification of Time Series: Cost-based multiclass Algorithms », in *IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*, Oct. 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9564134.

[142] Linux, *CFS Scheduler — The Linux Kernel documentation.* [Online]. Available: https://docs.kernel.org/scheduler/sched-design-CFS.html (visited on 09/08/2024).

[143] Linux, *Control Groups — The Linux Kernel documentation.* [Online]. Available: https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html (visited on 09/08/2024).

[144] Amazon, *Using Prometheus to Avoid Disasters with Kubernetes CPU Limits | Containers*, Section: Amazon Elastic Kubernetes Service, Sep. 2022. [Online]. Available: https://aws.amazon.com/blogs/containers/using-prometheus-to-avoid-disasters-with-kubernetes-cpu-limits/ (visited on 09/10/2024).

[145] Linux, *Red-black Trees — The Linux Kernel documentation.* [Online]. Available: https://www.kernel.org/doc/Documentation/rbtree.txt (visited on 09/08/2024).

[146] C.-C. Chuang and Y.-C. Tsai, « Performance Evaluation and Improvement of a Cloud-Native Data Analysis System Application », in *International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*, Dec. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9686398.

COLLEGES BRETAGNE LOIRE | SCIENCES POUR L'INGENIEUR ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Solutions Zero-Touch basées sur l'IA pour la gestion des ressources dans les réseaux 5G natifs du cloud

Mot clés : 5G, Mise à l'échelle automatique, Kubernetes, Cloud natif, Apprentissage profond, Limitation du processeur

Résumé : Le déploiement des réseaux 5G a introduit des architectures cloud-native et des systèmes de gestion automatisés, offrant aux fournisseurs de services de communication une infrastructure évolutive, flexible et agile. Ces avancées permettent une allocation dynamique des ressources, augmentant celles-ci en période de forte demande et les réduisant en période de faible utilisation, optimisant ainsi les CapEx et OpEx. Cependant, une observabilité limitée et une caractérisation insuffisante des charges de travail entravent la gestion des ressources. Une surprovisionnement pendant les périodes creuses augmente les coûts, tandis qu'un sous-provisionnement dégrade la QoS lors des pics de demande. Malgré les solutions existantes dans l'industrie, le compromis entre efficacité des coûts et optimisation de la QoS reste difficile. Cette thèse aborde ces défis en proposant des solutions d'autoscaling proactives pour les fonctions réseau dans un environnement cloud-native 5G. Elle se concentre sur la prévision précise de l'utilisation des ressources, l'identification des opérations de changement d'échelle à mettre en œuvre, et l'optimisation des instants auxquels opérer ces ajustements pour préserver l'équilibre entre coût et QoS. De plus, une approche novatrice permet de tenir compte de façon efficace du throttling de la CPU. Le cadre développé assure une allocation efficace des ressources, réduisant les coûts opérationnels tout en maintenant une QoS élevée. Ces contributions établissent une base pour des opérations réseau 5G durables et efficaces et proposent une base pour les futures architectures cloud-native.

Title: AI-Driven Zero-Touch Solutions for Resource Management in Cloud-Native 5G Networks

Keywords: 5G, Autoscaling, Kubernetes, Cloud-native, Deep learning, CPU Throttling

Abstract: The deployment of 5G networks has introduced cloud-native architectures and automated management systems, offering communication service providers scalable, flexible, and agile infrastructure. These advancements enable dynamic resource allocation, scaling resources up during high demand and down during low usage, optimizing CapEx and OpEx. However, limited observability and poor workload characterization hinder resource management. Overprovisioning during off-peak periods raises costs, while under-provisioning during peak demand degrades QoS. Despite industry solutions, the trade-off between cost efficiency and QoS remains unresolved. This thesis addresses these challenges by proposing proactive autoscaling solutions for network functions in cloud-native 5G. It focuses on accurately forecasting resource usage, intelligently differentiating scaling events (scaling up, down, or none), and optimizing timing to achieve a balance between cost and QoS. Additionally, CPU throttling, a significant barrier to this balance, is mitigated through a novel approach. The developed framework ensures efficient resource allocation, reducing operational costs while maintaining high QoS. These contributions establish a foundation for sustainable and efficient 5G network operations, setting a benchmark for future cloud-native architectures.