Intent-Based Cloud CDNs: Rethinking the Communication Mechanism Between Content Providers and Cloud CDN Operators via Intent-Based Northbound Interface

Shiyam Alalmaei, BSc, MSc
School of Computing and Communications
Lancaster University

A thesis submitted for the degree of
Doctor of Philosophy

December, 2023

I dedicate this thesis to my parents, whose sacrifices and encouragement shaped my path. To my family, whose unwavering support and love have been my anchor throughout this challenging journey.

This work is also dedicated to my mentors and advisors, whose guidance and expertise have been invaluable in shaping my intellectual growth. To my colleagues and friends who shared in the highs and lows of this academic adventure, your camaraderie added immeasurable richness to the experience.

May this work contribute, in however small a measure, to the vast body of human understanding in this field. With heartfelt appreciation and deep gratitude, this thesis is dedicated to those who made this academic odyssey a reality.

Declaration

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original and my own work. The material has not been submitted, either in whole or in part, for a degree at this, or any other university. This thesis does not exceed the maximum permitted word length of 80,000 words including appendices and footnotes, but excluding the bibliography. A rough estimate of the word count is:
37067

Shiyam Alalmaei

Intent-Based Cloud CDNs: Rethinking the Communication Mechanism Between Content Providers and Cloud CDN Operators via Intent-Based Northbound Interface

Shiyam Alalmaei, BSc, MSc.
School of Computing and Communications, Lancaster University
A thesis submitted for the degree of *Doctor of Philosophy*. December, 2023

Abstract

Content Delivery Networks (CDNs) are an important solution for easing Internet network traffic congestion and improving response latency. There is a growing trend to deploy Cloud CDNs (CCDNs) to provide more flexibility compared to traditional CDNs. Today, regardless of the current technological advancements, most CCDNs still make their resource and cache management decisions while being unaware of what Content Providers (CPs) want to achieve. For instance, the APIs of prominent CCDNs (such as Amazon CloudFront, Google Cloud CDN, and Microsoft Azure CDN) do not allow CPs to express their high-level targets, e.g., requests/region. Instead, CPs are mainly limited to specifying the desired geographical coverage and the origin content server.

Fortunately, there has been an increasing interest in the Intent-Based Networking (IBN) paradigm, which aims to allow users to express what they want to do (in the form of an intent) instead of how to do it. By employing IBN, CCDNs could move towards a communication scheme that is more adaptive, flexible, and portable, that tries to automate meeting the CP's intent target throughout the pre- and post-deployment phases of a service.

In this thesis, we present a solution to enable such Intent-Based CCDN. We illustrate how microservices modularity could be utilized in the intent translation process, decomposing intents into behavioral abstract policies that in turn get translated into technical realizations as microservices. This provides a more flexible, cost-effective, interoperable solution, and enables stakeholders to compare and alternate between microservice alternatives depending on the CP's intent.

We propose and implement a Low-Cost Intent (LCI) that targets cost reduction while provisioning a CCDN. We evaluate several realizations of this LCI, namely LCI1, LCI2 and LCI3 via selecting different microservice alternatives (i.e., leading to different overall CCDN deployments) for each. Additionally, we propose and implement different LCI refinement algorithms for performance improvement in various traffic scenarios. We conduct our evaluation experiments on Google Kubernetes Engine (GKE), and test it using traces of real CDN traffic obtained from a major ISP.

Evaluating our refined LCI different realizations against a baseline of GKE

demonstrated the differences between these varying options. The best was LCI1 which reduced infrastructure cost by up to 10% in return for a 2% rise in dropped requests. On the other hand, although LCI2 also led to cost reduction up to 10%, it caused an increased request drops by almost 20%. Finally LCI3 led to the highest infrastructure cost reduction by up to 20% but in return for the highest rise in dropped requests by 40%. Interestingly, in a traffic bursts scenario, although the refined LCI resulted no cost reduction, it actually outperformed the baseline by reducing the amount of dropped requests by 30%. Accordingly, we discuss the factors that led to these different variations between the different LCI flavors based on their corresponding selected microservice alternatives.

Publications

Only one publication, shown below, has been created directly from the thesis, from which large portions of this published work are used within Chapter 2 and Chapter 3:

Shiyam Alalmaei et al. (Nov. 2020). "SDN heading north: Towards a declarative intent-based northbound interface". In: *16th International Conference on Network and Service Management (CNSM)*. DOI: 10.23919/CNSM50824.2020.9269118

The following publication has been generated while developing this thesis, and to an extent has guided the thesis into what it has become:

Shiyam Alalmaei et al. (Dec. 2019). "OpenCache: Distributed SDN/NFV based in-network caching as a service". In: *Advances in Data Science, Cyber Security and IT Applications* 1098, pp. 265–277. DOI: 10.1007/978-3-030-36368-0_22

Acknowledgements

I would like to express my heartfelt gratitude to the individuals and institutions that have supported me throughout the completion of my doctoral journey. This work would not have been possible without their unwavering support, encouragement, guidance, and contributions. I am profoundly grateful to my primary advisor, Prof. Nicholas Race, for his great support, tremendous patience, invaluable insights, and relentless dedication to my success. His insightful feedback, constructive criticism, and encouragement have been instrumental in shaping this work.

I am deeply thankful to my co-advisors, Dr. Yehia Elkhatib, Dr. Matthew Broadbent, and Dr. Samia Chelloug, whose expertise, guidance, and diverse perspectives have enriched this research and broadened my horizons which have been an extraordinary source of inspiration. I am indebted to Dr. Yehia Elkhatib for his exceptional mentorship and availability even during his busy schedules. His mentorship, expertise, boundless patience, and belief in my abilities have been the driving force behind this research. His dedication to my growth as a scholar has been truly invaluable.

I am equally appreciative to the members of my thesis committee, Dr. Daphne Tuncer, Dr. Charalampos Rotsos and Prof. Paul Smith, for their rigorous evaluation of my work, valuable feedback, constructive criticism, and insightful suggestions, which greatly enriched the quality of this thesis.

My deepest gratitude extends to my parents, Prof. Amira Faloda and Dr. Mohammed Alalmaei for their unwavering love, encouragement, and support throughout my academic journey and the completion of this PhD thesis. Their sacrifices and belief in me have been the cornerstone of my success. My parents have been my constant source of strength, offering a safe haven in times of uncertainty and a wellspring of motivation in moments of doubt. I am truly fortunate to have parents who instilled in me the values of hard work, resilience, and a love for learning. Their sacrifices and dedication to my well-being and education have laid the foundation upon which I have built my academic career.

I am also extremely grateful for my extended family, Auntie Dr. Ahlam, and cousins Najlaa' and Saleh for their unconditional love and support have been an essential part of my journey. My family stood by me through the challenges and triumphs of this long and demanding endeavor. Their patience, encouragement, and sacrifices have made this achievement possible. I am profoundly thankful for the countless ways in which they have enriched my life. This thesis is not just a reflection of my own hard work, but also a testament to the enduring love and encouragement of my parents and family. I am eternally grateful for their presence in my life.

I would like to thank my friends, fellow researchers and colleagues for their camaraderie, stimulating discussions, and their understanding when I had to prioritize

my work. Your friendship has made this journey more enjoyable. I am especially thankful for my friends and colleagus, Dr. Ahlam Althobaiti, Dr. Dina Alhammadi, Dr. Mehdi Bezahaf, Dr. Shrooq Alsenan and Dr. Lyndon Fawcett. They have been a source of inspiration, and shared knowledge. Our intellectual discussions and collaborative efforts have greatly enriched the quality of this work.

I am deeply also grateful for the financial support provided by Princess Nourah Bint Abdulrahman University. This financial support has not only facilitated the realization of my academic goals but has also contributed to my personal and professional growth. I am immensely thankful for the opportunity they have provided, which has enriched my academic journey in ways that extend far beyond the scope of this thesis.

In conclusion, to all those mentioned and the many others who have helped me along the way, thank you for being an integral part of this academic endeavor. This thesis stands as a testament to our collective efforts, and I am truly grateful for your support and encouragement.

Contents

1 Introduction **1**
1.1 Cloud Content Delivery Network (CCDN) 2
1.2 Policy Based Management Systems 3
1.3 The Shift Towards Autonomic Networks 4
1.4 The move towards Intent-Based Systems 5
1.5 Motivation 7
1.6 Research Questions 10
1.7 Thesis Aims and Contributions 10
1.8 Thesis Structure 12

2 Background and Related Work **14**
2.1 Terminologies 15
2.2 Network Softwarization 16
2.2.1 Network Function Virtualization 17
2.2.2 Containerization 18
2.2.3 Software Defined Networking 19
2.3 Autonomic Networks 21
2.4 Microservices Architecture (MSA) 22
2.4.1 Background of MSA 22
2.4.2 Requirements Engineering in MSA 24
2.4.3 Multiple Criteria Decision Making (MCDM) 25
2.5 Intent-Based Northbound Interfaces 26
2.5.1 Intent Standardization Efforts 27
2.5.2 Intents and Policies 28
2.5.3 Different Intent Types 28
2.5.4 Meta-Analysis for Intent-Based Northbound Solutions 29
2.5.5 Intent-Based Northbound Solutions Limitations 32
2.6 Different CDN Flavors 34
2.7 CCDN Operations 35
2.8 CCDN Use Case Assumptions 36

2.9 CCDNs' Related work and Used Technologies . . . 37
2.10 Summary . . . 40

3 Design **42**
3.1 Design Motivation and Aims . . . 43
3.1.1 Leveraging CDNs by different domains . . . 43
3.1.2 Different CDN stakeholders' collaboration . . . 45
3.1.3 Bi-directional interaction between intent consumers and the CDN 46
3.1.4 Summary of Motivating Factors . . . 47
3.2 Intent Expressions, Syntax, and Descriptors . . . 48
3.2.1 Intent-to-Policy Mappings . . . 49
3.3 Architecture and Design . . . 51
3.3.1 Translation Layer . . . 54
3.3.2 Microservice Layer . . . 59
3.3.3 Database Layer . . . 60
3.4 Multiple Criteria Decision Making for CCDN Deployment . . . 60
3.4.1 Analytical Hierarchy Process (AHP) . . . 60
3.4.2 Analytical Hierarchy Process Computation . . . 64
3.4.3 The AHP Graph for CP's Targeted Workload Intent . . . 66
3.4.4 The Corresponding CCDN Deployment to the CP's Targeted Workload Intent . . . 71
3.5 Different Intent Targets . . . 73
3.6 Intent Refinement . . . 74
3.7 Discussion . . . 76

4 Implementation **77**
4.1 Communication Flow between Content Provider and the CCDN . . . 78
4.2 Implementing the Intent-Based CCDN . . . 80
4.3 Intent Translation . . . 81
4.3.1 Multi-Criteria Decision-Making Module . . . 81
4.3.2 CCDN Deployments Enumeration and Clustering Module . . . 83
4.3.3 Intent Technical Requirements Calculator Module . . . 85
4.3.3.1 Usual traffic behavior with gradual traffic increase . . . 87
4.3.3.2 Bursty Traffic behavior with sudden increase . . . 87
4.4 CCDN Deployment via Google Kubernetes Engine (GKE) . . . 91
4.4.1 Kubernetes Objects . . . 93
4.4.1.1 Kubernetes Deployment . . . 93
4.4.1.2 Horizontal Pod Autoscaler . . . 94
4.5 Summary . . . 95

5 Evaluation **96**
5.1 Experimental Methodology 96
5.1.1 **Intent Translation Evaluation** 96
5.1.2 **Intent Execution and Refinement Evaluation** 97
5.1.3 **CCDN Deployments Cost Calculation** 102
5.2 Intent Translation (CCDN Pre-Deployment Phase) 102
5.2.1 Increasing Number of Criteria Results 104
5.2.2 Increasing Number of Microservices Results 104
5.2.3 Discussion 105
5.3 CCDN Post-Deployment Phase 106
5.3.1 Normal Traffic with Gradual Increase 106
5.3.1.1 Low-Cost Intents Performance Results 107
5.3.1.2 Discussion 112
5.3.1.3 Low-Cost Intents Cost Results 114
5.3.1.4 Discussion 120
5.3.1.5 Low-Cost Intents Performance-to-Cost Score Results 120
5.3.1.6 Discussion 122
5.3.2 Traffic with Bursts 122
5.3.2.1 Low-Cost Intents Performance Results 123
5.3.2.2 Discussion 124
5.3.2.3 Low-Cost Intents Cost Results 126
5.3.2.4 Discussion 126
5.3.2.5 Low-Cost Intents Performance-to-Cost Score Results 128
5.3.2.6 Discussion 129
5.4 Summary 129

6 Conclusions **134**
6.1 Summary 134
6.2 Contributions 136
6.3 Future Work 139
6.3.1 Exploring the standardized Intent Common Model 139
6.3.2 Extending current CCDN with different intent targets and their translation 140
6.3.3 Advancing intent APIs with Natural Language Processing . . 141
6.3.4 Resolving intent conflicts 141

References **142**

Appendix A Evaluation Extended Results **156**
A.1 Detailed Experiment Tables for Normal Traffic Scenario 156
A.2 Detailed Experiment Tables for Traffic Bursts Scenario 172
A.3 Snapshots for CP and CCDN interaction 178

List of Figures

1.1 Network Management Progression Towards Intents-Based Networking. 8

2.1 VMs vs. Containers (Adam Getz, 2021, Retrieved from https://bi-insider.com/posts/virtual-machines-vs-containers/) 19
2.2 SDN Northbound Interfaces Categories 20
2.3 Intents, policies and operational commands. 29
2.4 Intent-Based Solutions Taxonomy 32
2.5 CDN Variation Criteria 34

3.1 CDN Operator and ISP Collaboration. 46
3.2 Bi-directional Interaction between users and the Intent-Based system (Bezahaf et al., 2021) 47
3.3 Abstract Policies Language Snapshot. 50
3.4 Intent-Based Framework. 55
3.5 CP's Approval on Suggested CCDN Deployment. 56
3.6 CP's Rejection of Suggested CCDN Deployment. 56
3.7 CP's Update on Suggested CCDN Deployment. 57
3.8 CP's Update on Suggested Performance Improvement. 59
3.9 Generic AHP Graph. 64
3.10 CP Workload Intent Target's AHP Graph (weights on arrows are omitted for figure clarity). 67
3.11 Different Possible CP Intent Targets. 73
3.12 MAPE-K Loop Mapping to Our Framework Components. 75

4.1 Communication Flow Between Content Provider and the CCDN. . . . 79
4.2 Clustered CCDN Deployments Based on Their Scores Towards the Intent Goal. 84
4.3 Total CCDN Deployments Clusters Based on Their Scores Towards the Intent Goal. 85

5.1 Normal Traffic with Gradual Increase. 98

5.2 Bursty Traffic (the cluster re-sizing behavior was omitted due to its stability throughout the test since the traffic bursts occurred frequently) 99
5.3 CCDN Experiment Setup on Google Cloud Platform. 100
5.4 Intent Translation Time (AHP Calculation and Clustering) for Varying Number of Criteria. 103
5.5 Intent Translation Time (AHP Calculation and Clustering) for Varying Number of Microservice Alternatives. 105
5.6 Average Number of Dropped Requests in a Week. 110
5.7 Dropped Requests Throughout a Week. 111
5.8 Total Dropped Requests in a Week. 112
5.9 Performance Ratio Comparisons Against Baselines (higher is better). 113
5.10 Cost($) Throughout a Week. 115
5.11 Cost($) in a Week. 118
5.12 Cost Ratio Comparisons Against Baselines (higher is better). 119
5.13 Performance-to-Cost Ratio Comparisons Against Baselines (higher is better). 121
5.14 Dropped Requests During Traffic Bursts. 123
5.15 Performance Ratio Comparison Against Baselines During Traffic Bursts (higher is better). 125
5.16 Cost Ratio Comparison Against Baselines During Traffic Bursts (higher is better). 127
5.17 Performance-to-Cost Ratio Comparison Against Baselines During Traffic Bursts (higher is better). 128

A.1 CP's Approval on Suggested CCDN Deployment. 178
A.2 CP's Rejection of Suggested CCDN Deployment. 178
A.3 CP's Update on Suggested CCDN Deployment. 179
A.4 CP's Update on Suggested Performance Improvement. 180

List of Tables

2.1 A comparison between policies and intents. 28
2.2 Summary of the results of our meta-analysis of intent-based solutions. 30
2.3 A comparison between current vCDN solutions. 38

3.1 Basic expression syntax. 50
3.2 Comparison Between AHP and NFR Frameworks. 61
3.3 The AHP Saaty's Original Scale. 62
3.4 AHP Criteria 69
3.5 AHP Scalability Sub-Criteria 69
3.6 AHP Cost Sub-Criteria 70
3.7 AHP Scalability of Microservices under Cache Size Category 70
3.8 AHP Scalability of Microservices under Cache Placement Startup Delay Category 70
3.9 AHP Scalability of Microservices under Refinement Category 70
3.10 AHP Cost of Microservices under Cache Size Cost Category 71
3.11 AHP Cost of Microservices under Cache Placement Cost Category . 71
3.12 AHP Cost of Microservices under Refinements Cost Category 71
3.13 Cache Size Microservice Alternatives Global Score 72
3.14 Cache Placement Microservice Alternatives Global Score 73
3.15 Refinement Microservices Alternatives Global Score 73

5.1 Compute Instances Costs in a Local Zone. 102
5.2 Compute Instances Costs in a Near-by Zone. 103
5.4 Performance Ratio to Baselines (higher is better) 109
5.6 Cost Ratio to Baselines (higher is better) 117
5.7 Performance-to-Cost Ratio to Baselines (higher is better) 117
5.8 Performance Ratio to Baselines in a Traffic Bursts Scenario 124
5.9 Cost Ratio to Baselines in a Traffic Bursts Scenario 126
5.10 Performance-to-Cost Ratio to Baselines in a Traffic Bursts Scenario (higher is better) 129

A.1 Baseline1 (GKE Autopilot) Dropped Requests During Scale-outs. . . 157
A.2 Baseline2 (Fully-Managed GKE) Dropped Requests During Scale-outs. 158
A.3 LCI1 Dropped Requests During Scale-outs. 159
A.4 Optimistically Refined LCI1 Dropped Requests During Scale-outs (green rows represent the refinement occurrence). 160
A.5 Pessimistically Refined LCI1 Dropped Requests During Scale-outs (green rows represent the refinement occurrence). 161
A.6 LCI2 Dropped Requests During Scale-outs (Part1: Sat - Tue). 162
A.7 LCI2 Dropped Requests During Scale-outs (Part2: Wed - Fri). 163
A.8 Optimistically Refined LCI2 Dropped Requests During Scale-outs (Part1: Sat - Tue) (green rows represent the refinement occurrence). . 164
A.9 Optimistically Refined LCI2 Dropped Requests During Scale-outs (Part2: Wed - Fri) (green rows represent the refinement occurrence). . 165
A.10 Pessimistically Refined LCI2 Dropped Requests During Scale-outs (green rows represent the refinement occurrence). 166
A.11 LCI3 Dropped Requests During Scale-outs (Part1: Sat - Tue). 167
A.12 LCI3 Dropped Requests During Scale-outs (Part2: Wed - Fri). 168
A.13 Optimistically Refined LCI3 Dropped Requests During Scale-outs (Part1: Sat - Wed) (green rows represent the refinement occurrence). 169
A.14 Optimistically Refined LCI3 Dropped Requests During Scale-outs (Part2: Thurs - Fri) (green rows represent the refinement occurrence). 170
A.15 Pessimistically Refined LCI3 Dropped Requests During Scale-outs (green rows represent the refinement occurrence). 171
A.16 Baseline1 (GKE Autopilot) Dropped Requests During Traffic Bursts. 173
A.17 Baseline2 (Fully-Managed GKE) Dropped Requests During Traffic Bursts. 174
A.18 LCI2 Dropped Requests During Traffic Bursts. 175
A.19 Refined LCI2 (By Earlier Scaling) Dropped Requests During Traffic Bursts. 176
A.20 Refined LCI2 (By Vertical Upgrade) Dropped Requests During Traffic Bursts. 177

Chapter 1

Introduction

Content providers face enormous pressure to satisfy increasingly stringent requirements. In one respect, low-latency content delivery is of utmost priority. Consider the reports that the volume of Google search requests drop by 0.59% for every 400ms of delay (Brutlag, 2009), and that Amazon profits are cut by 1% for every 100ms increase in latency (Flach et al., 2013). In another respect, users today expect the best possible quality in video experiences. Streaming formats evolve from high-definition (HD) to Ultra HD, 3D, Object Based Media (OBM) (Lyko et al., 2022), and beyond. Overall, the global video streaming market size is predicted to grow to $1902.68 Billion by 2030 (FortuneBusinessInsights, 2023b).

This imposes a significant burden on the underlying network infrastructure. This has led to the emergence of Content Delivery Networks (CDN) for fast and reliable content delivery. The core tenet behind a CDN is to place the actual responding servers (also called surrogates) as close as possible to the users across different regions. Specifically, they manage content delivery and infrastructure decisions such as caching, load balancing, etc. The CDN can push and pull the content to surrogate servers located close to the users; thus, they can obtain the desired content nearby. As such, CDNs are an important solution for easing Internet network traffic congestion, improving response latency, and optimizing user experience.

This huge demand has pressured the industry to implement and deploy numerous CDNs at different scales worldwide. For example, Akamai CDN handles 20%–30% of all web traffic (James, 2020). The CDN market is expected to witness steady growth due to the increasing volumes of exchanged data on the Internet in line with the continuous rollout of high-speed networks. It is predicted to expand at a compound annual growth rate (CAGR) of 23.0% to a revenue of $95.37 Billion in 2030 (GrandViewResearch, 2023). However, the application of CDN is not confined to content delivery; they can also provide a spectrum of services such as smart health (Min Chen et al., 2017), smart cities (Mingkai Chen et al., 2019), e-

learning (Palau et al., 2003), vehicle monitoring (Xiong et al., 2018), and drone and Unmanned Aerial Vehicle (UAV) monitoring (Asheralieva et al., 2019).

With the continuous development of many emerging technologies, such as Cloud Computing, Autonomic Networking, Network Function Virtualization (NFV), Software-Defined Networking (SDN), Microservices Architecture, etc., CDNs face new opportunities and challenges. These advancements could enable a new generation of CDNs. For instance, the increasing growth of the SDN/NFV market is anticipated with a CAGR of 16.75% by 2028 (TheExpressWire, 2023) and a CAGR of 21.6% for the Cloud Microservices market by 2030 (FortuneBusinessInsights, 2023a). Adopting these advancements could maintain and even accelerate the predicted growth of the CDN market.

Therefore, although there is an increasing popularity of CDNs in several domains targeting diverse service consumers, most CDNs still make their resource and cache management decisions while being unaware of the high level targets that service consumers want to achieve. Additionally, CDN operators are facing a challenge of meeting a larger number and wider variety of customization requirements for CDNs based on service consumers demands. However, owing to the high cost and complexity of this traditional approach, it is important to investigate the incorporation of different network management paradigms and autonomic solutions that aim to lift some of the burden of this skill-demanding decision-making process from CDN operators, and facilitate better communication between consumers and CDNs to improve their interaction and achieve better performance of service deployment and content delivery.

1.1 Cloud Content Delivery Network (CCDN)

In this thesis, we focus on Cloud CDNs (CCDNs). Recent advances in Cloud Computing allow leasing resources (i.e., compute, memory, storage, and bandwidth) to build CDNs in the cloud (F. Chen et al., 2012). There is a growing trend to deploy CCDNs. The global CCDN market is estimated at $6.7 Billion in 2022 and is projected to reach $30.5 Billion by 2030 (GlobeNewsWire, 2023a). They alleviate major limitations of traditional CDNs that rely on fixed physical caches, by leveraging virtualization technologies (such as NFV) to provide more flexibility in dynamic infrastructure provisioning, offering a virtual CDN (vCDN) service as a Software-as-a-Service (SaaS) model that can be deployed on demand. This embodies the B2B2X (Business to Business to X) (TMForum, 2014) business model, where in this context, the *First B* are the CCDN operators, and the *Second B* are the Content Providers (CPs) who use the CCDN to provision their content to their end-users (X). Accordingly, this delegates the content distribution and infrastructure management to the CCDN operator since the CP is agnostic to the infrastructure and caching

management details.

CPs are the early adopters of cloud services to meet their end-user demands. In the past few years, popular cloud vendors started to compete in the video market and provide CDN services to cache and distribute content for CPs, like Amazon CloudFront (Amazon, 2023), Google Cloud CDN (Google, 2023c), and Microsoft Azure CDN (Microsoft, 2023b). Adopting CCDN solutions claims to offer a cost-effective solution of CCDN on a pay-per-service basis. Moreover, hosting CDN services over the cloud increases the availability of the services exponentially by leveraging the increased number of points-of-presence (PoPs) (Jia et al., 2017) (Jayakumar et al., 2018). For example, today Amazon Cloudfront maintains over 450 PoPs across 49 countries (Amazon, 2023), Google Cloud CDN operates caches in more than 200 countries across the world (Google, 2023f), and Microsoft Azure CDN has 192 PoPs across 109 metro cities (Microsoft, 2023a).

In comparison to traditional CDNs, CCDNs have increased scalability (Cisco, 2017), flexibility (Cisco, 2017), elasticity (Cisco, 2017), reliability (Altomare, 2023), and security (Altomare, 2023). They have also reduced the prices for content storage and delivery by orders of magnitude (F. Chen et al., 2012) and reduced capital expenditure (CAPEX) and operational expenditure (OPEX). This makes CCDNs an affordable option for small and large-scale content providers, such as small businesses, government, and educational organizations (Broberg et al., 2008).

As a result, cloud service providers and CCDN operators are expected to deliver their cloud solutions and services to more and more service consumers (CPs in this context) while meeting a larger number and wider variety of customization requirements for clouds based on CP demand. This situation calls for numerous highly skilled CCDN operators to orchestrate cloud services in accordance with user requirements. However, due to the high cost and complexity of this traditional approach, it is important to investigate network management paradigms and autonomic solutions that could improve and facilitate this decision-making process.

1.2 Policy Based Management Systems

As network and cloud management progressed, a management paradigm emerged known as Policy-Based Management (PBM), which could lift up some of the burden of the traditional decision-making process from CCDN operators that has been discussed in the previous Section 1.1. It enables network operators, administrators, IT personnel, etc. to define, consume, and translate policies that express precisely what to do and under which circumstances (Du et al., 2017). This management paradigm introduces high-level specifications that a network should meet.

A policy definition is a set of rules that govern the choices in the behavior of

a system (Clemm et al., 2022). In specific, a network policy abides by the tuple: events/conditions/actions (ECA) that guide the network to be configured. Therefore, a policy defines what to do and under which conditions.

However, such systems cannot guarantee the network's sustainability since they cannot include all possible situations that may occur (Leivadeas et al., 2022). To this end, knowledge management systems have been proposed that could facilitate the decision-making process (Agoulmine et al., 2008). For this reason, appropriate information modeling representation can be used to allow structuring the knowledge and giving semantic meaning to network management operations through the use of ontologies (Serrano et al., 2007).

Undoubtedly, the right combination of policies with good knowledge management systems could bring the networks closer to autonomic operations which are discussed in the next section. This could be achieved by allowing learning and reasoning techniques to generate new policies to appropriately reconfigure the network (Jennings et al., 2007).

1.3 The Shift Towards Autonomic Networks

The emergence of Autonomic Networks was due to the need for a self-managed system that reduces the cost of having a well-skilled team that manages complex systems (Behringer et al., 2015). Therefore, networks and computing paradigms have started to change radically from the early 2000's and shift away from traditional manual system configuration and management. The self-managed Autonomic Network vision, besides the management cost reduction, is to adapt and react to environmental changes with no or little intervention of humans (Kephart et al., 2003). It usually operates based on high-level rules or policies formed by the system administrators, which often follow IF-THEN or event-condition-action (ECA) form (Kephart et al., 2004). This concept is inducted from the early efforts on bio-inspired approaches in the human nervous system, which emulates the behavior of the autonomic nervous systems found in the human body and biological systems (e.g. blood sugar, temperature, heartbeats control) without deliberate intervention (Horn, 2001), (IBM, 2006).

The first clear vision of the autonomic computing systems was proposed by IBM (IBM, 2006). IBM characterized the self-management of an autonomic system in four self-* properties, namely: self-configuration, self-healing, self-optimization, and self-protection (Kephart et al., 2003), (Sinreich, 2006), (Parashar et al., 2004), (Sterritt, 2005), (Huebscher et al., 2008).

1.4 The move towards Intent-Based Systems

Current and future network services, applications, and systems are expected to improve our society and lifestyle. Especially the systems that are moving towards being self-configuring, self-managing, and self-optimizing. However, these abundant possibilities offered to end-users, network operators, and administrators have created a cumbersome system configuration process to adjust to all different stakeholders and services (Behringer et al., 2015). Therefore, even with the current advancements in PBM and autonomic systems, there is still a need to simplify the management and configuration of the network in an autonomic way. (Szilágyi, 2021) Intent-Based Networking (IBN) is such a paradigm that realizes simplified, flexible, and agile network management and configuration with minimal external intervention.

Intent can be defined as *a declarative, abstract, and vendor-agnostic way of describing the targeted system state*. It abstracts the objects and capabilities of the system from the perspective of requirements and can be translated into advanced policy rules that are used to provide a full lifecycle (Design/Build/Deploy/Validate) to a system and services it provides (Clemm et al., 2022). Thus, it can automatically convert, verify, deploy, configure, and optimize by itself to achieve the targeted state of the system, and can automatically handle abnormal events to ensure the system's reliability according to the expressed intent (Pang et al., 2020).

Although IBN is a brand-new term and technology that ultimately aims at forming an autonomic network by making it simpler to manage and operate(Clemm et al., 2022), It does not aim to create a new communication paradigm from scratch. Instead, it will be largely based on existing network frameworks that have already helped in automating its provisioning and configuration, in which a network needs to manifest all the necessary "self" properties (i.e., self-configuration, healing, optimization, etc.) (Behringer et al., 2015). Hence, some of the main network technologies related to IBN, such as Software Defined Networking (SDN), Network Function Virtualization (NFV), and Network Policies, will be discussed in Chapter 2, Section 2.2.

IBN can be a programmable and customizable system automation, which can realize *Intent Representation*, *Global System Status Awareness*, and *Closed-loop Optimization*. Details are given as follows.

- **Intent Representation:** Users can tell the network their intent, regardless of the used intent expression, the system needs to understand the intent and translate it into a specific expression.
  - **Intent Expression:** The very first component of IBN is the part that enables users to interact with the IBN system to express their intent. Intents could be either *Prescriptive* or *Declarative*. Each facilitates the expression of the desired system target but at different levels and with different specifications. Regardless

of the intent expression choice, it should be submitted in a human-friendly way (e.g., through natural language expressions, drop-down menus, etc.). In some cases, the system should also interact with the user to guide them through expressing their intents meaningfully.

  - **Intent Translation:** After the high-level intent has been expressed and submitted, obviously, it cannot be used directly to configure the system. Hence, it should be translated into a set of behavioral policies that will be rendered into low-level system configurations that can be technology-specific to the underlying system infrastructure.

- **Global System Status Awareness:** The IBN framework needs to control and adjust the system configurations to meet the intent goals based on the system state. Therefore, the perception of the system status is indispensable and could be achieved by means of monitoring and telemetry to provide adequate measurements to guarantee reactive network adaptations according to the current conditions.

  Accordingly, identifying and defining the metrics, resources status and configuration, events, etc., is an essential part that helps with getting the relevant system awareness with respect to the intent target. So, the target has to be measurable (e.g., requests throughput, cost, etc.) where the goal is determined based on the intent target, and thus, the equivalent metrics are continuously monitored and compared to the expected target and how far they are from meeting it. Additionally, it is important to detect abnormal system behavior (e.g., sudden traffic bursts, resource crashes, etc.) or even a natural change in the service request behavior, in order to take corresponding actions, where these detections could be either based on a comparison against some predefined thresholds (e.g., X% increase in traffic rate), or previous service behavior patterns.

  - **Intent Conflict Resolution:** Different users may submit their intents independently at the same time. This can easily lead to contradictory or conflicting system configurations that could affect new or already deployed intents. Hence, an IBN should have an Intent Conflict Resolution module that will be able to disambiguate conflicting intents by proposing ways to resolve conflicts, negotiate them, and ideally alert the user and/or the system administrator if the resolution is not feasible.

  - **Intent Activation:** After the conflict resolution, the IBN should proceed with the provisioning of the requested service. Since intents may also include personalized desired targets of the users. Accordingly, the IBN needs to deploy each intent in a customized way that will eventually configure the system as intended by the user.

- **Closed-loop Optimization:** According to the feedback of the global system status, the gap between the current system configuration and the intent's desired target system state is inferred. The closed-loop system adjustment (full-lifecycle management) is continuously performed to enhance the learning mode to adjust the system configuration and finally satisfy the user intent.
  - **Intent Assurance:** The previous Intent Activation component will only ensure that the intent will be satisfied at the moment of its deployment. However, in reality, systems present a highly dynamic nature both in time and space. Thus, another indispensable component to creating the anticipated autonomous system behavior (i.e., from the intent consumer's perspective) is Intent Assurance. This specific component will have to make sure that the system complies with the intent throughout its whole lifetime. To do so, it has to take both proactive and reactive measures towards the self-configuration and self-healing of the system and help the users to refine their intent when a gap is detected between their desired target and the system behavior.

Although IBN is a brand new term, from our previous discussion, it is clear that it has strong ties with the past work that has been done around autonomic networks and computing, and policy-based network management (Szilágyi, 2021). Obviously, the early autonomicity directions put significant foundations for the IBN, especially for the network assurance part. Nonetheless, the main focus of these early works was focused on how the network can be auto-configured and self-managed with less interaction between the user and the network itself.

Therefore, IBN surfaced the need for reducing the gap between different types of users, ranging from a simple novice user to a highly experienced network administrator, and an ever-evolving network infrastructure with all of the technological evolution that we are encountering. Figure 1.1 demonstrates the progression towards IBN and how the current network management paradigms, namely, *Policy-Based Networking* in Section 1.2 and *Autonomic Networking* in Section 1.3 are connected with respect to IBN.

1.5 Motivation

Today, regardless of the current technological advancements, especially with the current direction toward IBN, most CCDNs still make their resources and caching management decisions oblivious to what the CPs want to achieve. For instance, the most popular CCDNs (e.g., Amazon CloudFront, Google Cloud CDN, and Microsoft Azure CDN) APIs do not allow CPs to express their high-level targets (e.g., requests/region, latency, etc.) since they are mainly limited to selecting the

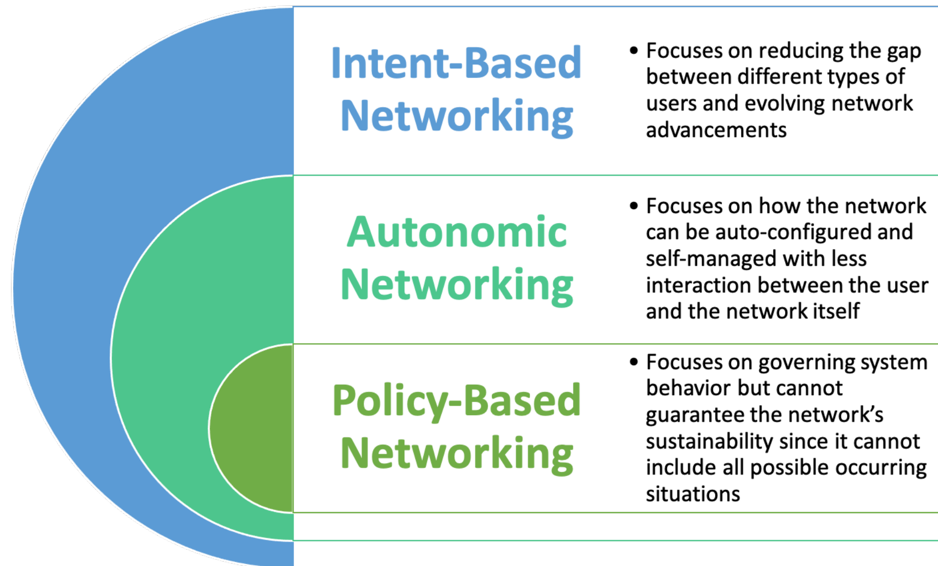

Figure 1.1: Network Management Progression Towards Intents-Based Networking.

desired geographical coverage and specifying the origin content server. Some of them offer APIs to configure some low-level technical details like the backend service load balancing, health checks, session affinity, connections per instance, etc. (Google, 2023b). However, generic CPs with no technical background and who are only interested in expressing their high-level intent, would face some challenges with the current APIs. Therefore, with all of the current technologies evolution, next deployment phase of CCDNs could move towards an interactive interface between CPs and CCDN operators that deploy CCDN, which enables CPs to declare their high-level targets and get updated on the CCDN's ability to achieve these targets throughout the service lifetime. This could possibly lead to better and more efficient resources and caching management decisions by providing more adaptive, flexible, technology-agnostic and portable CCDN that tries to meet the CP high-level target in an automated way throughout the pre- and post- deployment phases of the service.

Moreover, most CCDNs offer pre-defined caching and resource management that could possibly fall behind the rapid technology development and dynamic demand changes. Therefore, the Microservices Architecture, which is a widely adopted paradigm in today's systems should be investigated and adopted in this context. The Microservice Architecture (discussed in Chapter 2, Section 2.4) facilitates faster system management, independent service deployment, increased scalability and agility, the realization of user objectives, and reduced OPEX due to less human involvement. CCDN microservices (e.g. caches, load balancers,

DNS servers, etc.) could be easily upgraded or scaled without affecting the other functionalities. Moreover, this facilitates the realization of the CCDN functionalities in different ways depending on the underlying microservice implementation in light of CPs intent targets, which also enables CCDN operators to compare and alternate between different microservices that provide the same functionality but with different costs, performance, etc. Therefore, this provides a more flexible, cost-effective, and portable solution that simplifies the CDN operator's task and takes them a step closer to autonomicity, and enables upgrading the CCDN with different and updated microservices depending on the CP's target.

Accordingly, with the expected growth of the Intent-Based Networking market with a CAGR of 23.9% to reach $8.8 Billion by 2030 (GlobeNewsWire, 2023b), next deployment phase of CCDNs could provide an Intent-Based interface between CPs and CCDN operators which considers different types of CPs and their high-level targets, and allows bi-directional communication between them, unlike the current limited and rigid interfaces capabilities in CCDNs. However, there are several problems that need to be considered:

1. **Declarative CP intent expression for high-level targets:** Most existing intent expressions are *prescriptive* rather than *declarative*, which are mostly used to express network-level requirements (Alalmaei et al., 2020). However, in the context of CCDNs, there are more complex requirements that are beyond network-level such as caching intents with high-level targets, and as such, cannot be easily expressed via the current prescriptive-level intent expressions. Therefore, CPs need an easier and higher-level expression that is purely declarative, to allow them to express expected targets without the need for stipulating any low-level policies. For example, a CP could express a high-level declarative intent as *I want caching for content X to handle 1,000 requests/second*. In this expression, the CP only states the required target rather than some prescriptive operational policies, and without the need for any lower-level configurations.

2. **Separation between the CCDN behavioral policies and underlying technology:** CCDNs need to keep up with the technology advancements and demand changes. However, this is hard to achieve when the CCDN functionalities are tightly coupled to the underlying technologies and implementations. Therefore, it is important to separate the CCDN behavior from its actual realization. This enables CCDN operators to think at a higher level and articulate abstract behavioral policies without worrying about the implementation complexities. Moreover, this provides a more flexible, cost-effective, and inter-operable solution by allowing the same policy to be applied across different underlying technologies without the need to define multiple variants. Different policy agents can be plugged into the different underlying systems to translate abstract policies into a technology-specific

representation. This also enables comparing and alternating between different realizations for the same abstract policies, which facilitates the collaboration between different stakeholders in the policy definitions.

3. **Translating high-level intents to lower-level policies:** Proposing high-level declarative intents that express desired operational targets necessitates decomposition and translation processes to reach a representation that could both be understood and achieved by the system yet abstract enough to govern the behavior of the system rather than its technical functionality. The decomposed policies represent the expressive system conditions, capabilities, requirements, and constraints defined by the CCDN operator. Afterward, policies get translated to lower-level system operations. Therefore, a mechanism is needed to manage this translation process with respect to the CP's target, which could leverage utilizing useful technologies like the Microservices Architecture.

1.6 Research Questions

In light of the discussed motivation, we form the following key research question (RQ): **How can a CCDN Intent-Based Interface facilitate the communication mechanism between a CP and a CCDN Operator?**

In order to answer this, we break it down into 3 sub-questions that are answered in the course of the presented work:

- **RQ1:** What are the limitations of state-of-the-art technologies (chiefly SDN, NFV, and microservice architectures) in realizing adaptive deployment and management of CCDNs with respect to CP requirements?
- **RQ2:** How can CPs (primarily, non-technical users) and CCDN operators (technical users) express their requirements to the CCDN?
- **RQ3:** What solution is required to support intent translation and refinement, as well as autonomic service deployment in a CCDN use case?

1.7 Thesis Aims and Contributions

This thesis aims to investigate the adoption of an Intent-Based Interface in CCDNs that facilitate communication between a CP and a CCDN Operator. This includes the design, implementation, and evaluation of a working solution for expressing high-level CP intents, and realizing them within the operational constraints of the CCDN. The above research aim leads to five main contributions provided by this thesis, which are summarized as follows:

1. **A specification of Declarative High-level Intent Expression for Content Providers and Behavioral Prescriptive Policy Expression for CCDN Operators:** The thesis examines the expression syntax of both Content Providers (non-technical users) and CCDN operators (technical users). A declarative intent expression will be considered for Content Providers, whereas a behavioral prescriptive policy expression will be considered for CCDN Operators. This is achieved through a meta-analysis of existing intent and policy expressions to identify prospects and gaps.

2. **Design for an Intent-Based CCDN Framework:** The aim of designing an Intent-Based CCDN is achieved by conducting a meta-analysis of existing CCDN solutions, as well as the integration of a variety of emerging technologies and trends. In particular, this thesis makes use of Microservices Architecture and Autonomic Networks. The former helps achieve a scalable, agile, flexible, and portable solution, while the latter leads to less human involvement, cost reduction, and self-managed adaptability. We provide a comprehensive design that contextualizes and encompasses these elements.

3. **An implementation of the Intent-Based CCDN Framework:** A proof-of-concept implementation is developed to evaluate and examine the effectiveness of the above design. This follows the specification of the aforementioned design and forms the basis for evaluating the integration of intents in the CCDN scenario. This implementation will also help demonstrate the benefits of leveraging microservices advancements.

4. **An evaluation of the intent translation overhead and feasibility:** Using the proof-of-concept implementation, the thesis contributes an assessment of overhead (in terms of delay) of the translation of high-level intent targets to lower-level commands that deploy the corresponding CCDN. Specifically, we evaluate the feasibility of the translation process with increasing problem sizes and different dimensions (e.g., number of microservices, number of evaluation criteria).

5. **An implementation, and performance and cost tradeoffs evaluation of different Low-Cost Intent realization alternatives and their refinements:** Based on the implementation of our Intent-Based CCDN design, we create and implement several Low-Cost Intent realization alternatives that aim at provisioning Content Providers with lower-cost CCDNs. We evaluate and compare their performance (in terms of dropped requests) and cost tradeoffs. We also implement and evaluate their refinements to improve their performance.

1.8 Thesis Structure

This thesis is organized into six chapters. Following this introduction, we go through some of the background of related technologies in Chapter 2. Specifically, we discuss the trend toward the softwarization and modularization of networks and their services via Microservice Architectures. This chapter also discusses Intent-Based Northbound Interfaces in detail, where we distinguish between intents and policies, and discuss different intent types based on their users and problem domains. Then, we discuss thoroughly their current limitations. Additionally, we provide a meta-analysis of some of the current Intent-Based solutions to summarize their characteristics and limitations. Moreover, this chapter includes a classification for different CCDN flavors based on different criteria such as their infrastructure and management schemes. Finally, this chapter examines the current limitations of CCDNs, highlighting their used technologies, and concluding their current limitations.

The following chapter, Chapter 3, presents our proposed design of an Intent-Based CCDN platform that seeks to utilize emerging technologies to improve the communication and management process between CPs and CCDN operators. This includes our detailed motivation and aims, which have been influenced by a multitude of sources, and result in a multi-layered architecture capable of meeting the CP's intent target throughout the service lifetime with respect to certain evaluation criteria and behavioral policies defined by the CCDN operators and different possible stakeholders.

Chapter 4 starts with presenting the whole communication flow between the CP and the CCDN system. Then it provides details of a proof-of-concept implementation of the aforementioned design. This includes the Intent Translation modules in that Layer, where we demonstrate the creation of the corresponding CCDN deployment selection graph via the Analytic Hierarchy Process (AHP) which is our chosen Multi-Criteria Decision Making (MCDM) tool. Further, we discuss the prioritization, enumeration, and clustering of all possible CCDN deployments with respect to the CP's intent. Finally, we present the algorithms and implement our proposed alternatives of Low-Cost Intents along with their refinement algorithms that aim to improve their performances in different traffic situations.

In Chapter 5, we present our evaluation of the proof-of-concept implementation and Low-Cost intent realizations and refinements in different traffic behavior scenarios. Each of these evaluations takes place on a Google Kubernetes Engine (GKE) Platform which is one of the popular Google Cloud services that benefit from its wide infrastructure coverage and underlying technologies. In the first instance, we demonstrate the overhead (in terms of delay) of the intent translation processes that include the AHP graph creation and calculation with respect to the CP's intent to eventually prioritize all possible included microservices that form up a CCDN deployment. We then examine the enumeration and clustering of all CCDN

deployments into different scored-clusters depending on their capability on achieving the intent target. The performance and cost tradeoffs for all proposed Low-Cost intents and the level of improvement after their refinements are then evaluated and compared. Eventually, we evaluate these intents in different traffic-intensity situations.

Finally, in Chapter 6, we present the contributions, impacts of this work, in addition to outlining future avenues of research created as a result.

Chapter 2

Background and Related Work

There is a growing trend to deploy CCDNs that leverage the cloud computing advancements. In 2022, the global CCDN market was estimated at $6.7 Billion, and is projected to reach $30.5 Billion by 2030 (GlobeNewsWire, 2023a). This leads the cloud service providers and CCDN operators to deliver their cloud solutions and services to a larger number and wider variety of service consumers while meeting more complex and diverse customization requirements for clouds based on the service consumer's demand. The challenge calls for many highly skilled CCDN operators to orchestrate cloud services in accordance with users' requirements. Today CCDN operators can leverage the current technical advancements in the field of Softwarization that deliver different services and applications with greater flexibility, adaptability, agility, cost effectiveness and total reconfiguration of a network on the fly in a matter of minutes rather than days (Manzalini et al., 2016), (Sousa et al., 2018), (Cerroni et al., 2020), (Barakabitze et al., 2022) and (Alwis et al., 2022). However, regardless of this promising paradigm, there are some limitations that we address here. Therefore, in this chapter, we start with listing some of the main terminologies used throughout the thesis in Section 2.1. Then, we present the trend of Network Softwarization, and how it has evolved to help CCDN operators meet the changing needs of service consumers and systems behavior. In specific, we tackle Network Function Virtualization (NFV), Containerization, and Software-Defined Networks (SDN). This progression has led to a number of significant benefits and challenges, in both present and future, which are outlined in Section 2.2.

Another network management paradigm shift has changed the traditional management approach, namely, the Autonomic Network paradigm which is discussed in Section 2.3.

Today most CCDNs offer pre-defined caching and resource management that could possibly fall behind the rapid technology development and dynamic demand changes. Therefore, the Microservices Architecture, which is a widely adopted paradigm in

today's systems should be investigated and adopted in this context. In Section 2.4, we address this highly popular and useful system paradigm that could benefit CCDN operators by changing the deployment and management of current systems via modularity as opposed to traditional monolithic systems. We discuss its advantages and challenges as well.

Although these technological advancements are very helpful in improving the current CCDNs, there is still a gap between the CP's requirements and the CCDN requirements and goals. This is because most CCDNs still make their resources and caching management decisions oblivious to what the CPs want to achieve. Hence, there is a current interest in system management schemes that utilize user intents which allow them to express their high-level requirements to the system to consider, which are facilitated through their Intent-Based Northbound Interfaces (NBIs). Therefore, we break down and compare the current existing NBIs, and in particular, we provide a meta-analysis of the current Intent-Based solutions and discuss their limitations in Section 2.5. A meta-analysis is a type of research that is often used to determine general information from a systematic review procedure. It combines the findings of other empirical studies into a summary study of available data, findings, and results on the given topic.

Moving forward to CCDNs, we classify them based on several dimensions. Then we state our own assumptions for the CCDN use case that we focus and work on in Section 2.6, 2.7. and Section 2.8.

Finally, in Section 2.9, we present our meta-analysis of the current CCDN solutions along with their used technologies, and we discuss their overall limitations. According to the current gap in this field, we break down some of its shortcomings in this section.

2.1 Terminologies

In this section, we list some of the main terminologies that will be referred to throughout this thesis.

- **Content Delivery Network (CDN):** Is a geographically distributed group of servers (also called surrogates) that caches content close to end users across different regions. Specifically, they manage content delivery and infrastructure decisions such as caching, load balancing, etc., for fast and reliable content delivery.

- **Cloud Content Delivery Network (CCDN):** Provides a specific form of CDNs, which build CDNs in the cloud to alleviate major limitations of traditional CDNs that rely on fixed physical caches, by leveraging virtualization technologies to provide more flexibility in dynamic infrastructure provisioning, offering a virtual CDN service as a Software-as-a-Service (SaaS) model that can be deployed on demand.

- **Intent:** A declarative, abstract, and vendor-agnostic way of describing the targeted system state, operational goals, or expected outcomes that the system should deliver. Therefore, ultimately, intents could be used in many domains and by different users, even non-expert users who do not have a lot of system knowledge.
- **Intent-Based Northbound Interface:** An interface that facilitates the interactions between consumer and provider systems via intents.
- **Intent Handler:** A logical component (or components) that receives intents and handles them in the domain that is responsible for that intent's fulfillment (i.e., translation to lower-level presentations that are understood by the underlying system).
- **Intent Life Cycle:** A full intent lifecycle starts from the initial request of intent until it gets terminated or stopped. It passes through the(Design/Build/Deploy/Validate) stages. Thus, it can automatically convert, verify, deploy, configure, and optimize by itself to achieve the targeted state of the system, and can automatically handle abnormal events to ensure the system's reliability according to the expressed intent target.
- **Declarative Intent:** A high-level intent expression where users can express "what" they want to achieve rather than "how" to achieve it.
- **Prescriptive Policy:** A rule expression that regulates the system behavior under different situations.
- **Microservice Architecture:** A modular architectural style that structures an application or a system as a collection of microservices that are: organized around business capabilities, independently deployable, loosely coupled, technology-independent, and can communicate with each other via universal APIs.
- **Microservice:** The actual running instance of a modular microservice component which provides a specific functionality.

2.2 Network Softwarization

Network softwarization is an approach that aims to design, architect, deploy and manage network components, by separating the software implementing network functions, protocols and services from the hardware running them. This approach will improve how network and computing infrastructures are designed and operated to deliver different services and applications with greater flexibility, adaptability, agility, cost-effectiveness and total reconfiguration of a network on the fly in a matter of

minutes rather than days (Manzalini et al., 2016), (Sousa et al., 2018), (Cerroni et al., 2020), (Barakabitze et al., 2022) and (Alwis et al., 2022). NFV, Containerization, and SDN are expressions of network softwarization. Hence, Softwarization is expected to impact several aspects of network development and services such as CDNs (Frangoudis et al., 2016) (Retal et al., 2017).

Although these rising softwarization paradigms are very promising, these abundant possibilities have imposed additional challenges by creating a cumbersome system configuration process to adjust to all different CDN stakeholders, users, and services (Leivadeas et al., 2022). Therefore, despite today's level of programmability of softwarized networks, they need experienced programmers (i.e., network managers, admins, operators, IT personnel, etc.) who can orchestrate the services in accordance with different and conflicting customization requirements for the system and the consumers (Leivadeas et al., 2022) (Alalmaei et al., 2020)(Tuncer et al., 2018). Hence, network programmability is a key enabler to truly leverage the benefits of the current softwarization approaches which could facilitate faster, more efficient, and autonomous network management, service deployment, realization of user objectives, and reduced OPEX due to less human involvement, which could lift up some burden from the IT personnel in charge, and thus reduce the need for their continuous involvement which could lead to cost reduction.

2.2.1 Network Function Virtualization

NFV (Mijumbi et al., 2015) is a paradigm that virtualizes network functions which are normally deployed on dedicated hardware (e.g. routers, load balancers, firewalls, etc.). The network functions are then decoupled from their dedicated hardware to run as virtual appliances on commodity hardware instead. It is fundamentally changing how network services are deployed and managed by providing flexibility, agile service delivery, auto-scalability and optimal resource usage. These services are provided over the same common infrastructure. The European Telecommunications Standards Institute (ETSI) has defined a framework for NFV and Management and Orchestration Architectures (MANO) (ETSI, 2013). These open-source architectures are broadly defined to allow development, extension, and testing in proprietary ways.

Authors in (Sousa et al., 2018) compare several NFV MANO projects. However, OSM (OSM, 2023), is an open-source NFV MANO platform that is considered as the reference implementation for the NFV MANO since it is hosted by ETSI and aligned with its information models to meet the requirements of production NFV.

2.2.2 Containerization

Containerization has become a major trend in network softwarization as an alternative or companion to virtualization. It is an increasingly popular method of designing and deploying Microservice applications. Containers are lighter weight and a more agile way of handling virtualization. In virtualization, a virtual machine (VM) is an emulation of a physical computer that enables running multiple machines, with multiple Operating Systems (OS), on the same physical computer. The VM technology depends on a lightweight software layer called hypervisors, which separate VMs from each other and allocate physical resources (e.g., processors, memory, storage) among them.

On the other hand, rather than spinning up an entire VM, a container encapsulates and packages together everything needed to run a small piece of software, including all the code, its dependencies, configuration files, and even the OS itself (unlike the traditional VM applications where the code is developed in a specific computing environment). This container is abstracted away from the host OS, and hence, it stands alone and becomes portable by allowing running applications uniformly and consistently on almost any infrastructure (e.g., desktop computer, traditional IT infrastructure, the cloud). Containerization technology uses a form of OS virtualization as opposed to traditional virtualization, which virtualizes the underlying hardware. Containerization leverages features of the OS to isolate processes and control their access to physical resources (e.g., CPUs, memory, and desk space). So, essentially, the main difference between containers and VMs is that each VM runs its own OS under the Hypervisor whereas containers share the host OS and run under the container engine layer.

Containerization brings several features like efficiency, portability, security, flexibility, etc. These factors help containers become a welcome alternative to the extensive use of VMs in today's systems. In fact, Research firm Gartner projects that by 2022, more than 75% of global organizations will use containerization technology, up from less than 30% in 2019 (Gartner, 2020). As for a recent survey published by the Cloud Native Computing Foundation (CNCF) (CNCF, 2020), Containers are used in production by 92% of the global cloud user community where containers usage in production has increased by 300% since 2016.

Containerization facilitates the creation and operation of stable modules of small artifacts which fit the description of a Microservice Architecture of decoupling applications into smaller-size services. Therefore, Containers serve as a perfect vessel for deploying such microservices on a large scale, and that is why Docker (Docker, 2023) and Kubernetes (Kubernetes, 2023d) are rising in popularity day by day. An overall comparison between VMs and containers is depicted in Figure 2.1

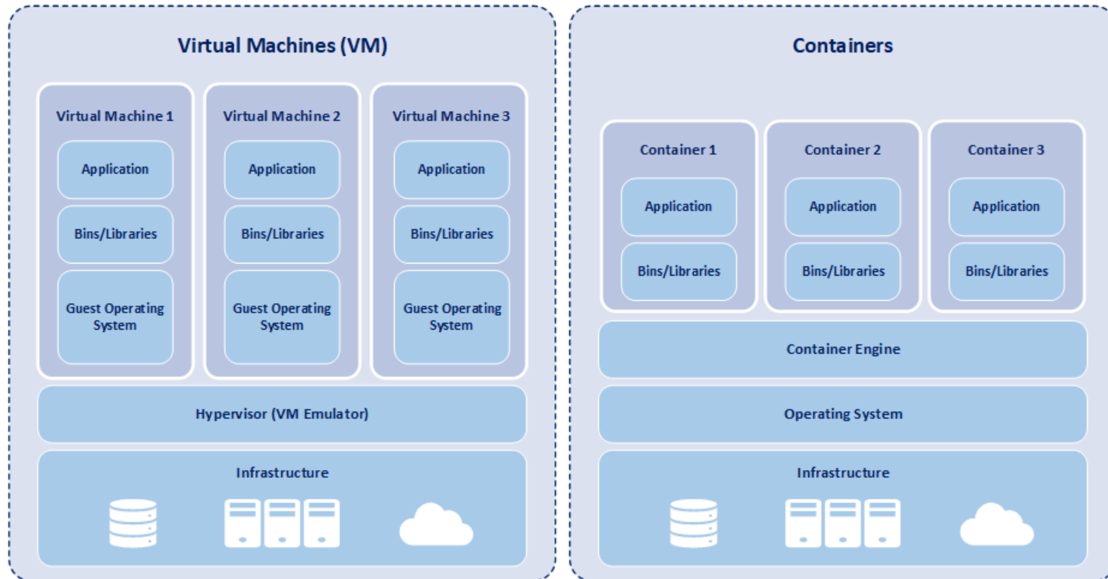

Figure 2.1: VMs vs. Containers (Adam Getz, 2021, Retrieved from https://bi-insider.com/posts/virtual-machines-vs-containers/)

2.2.3 Software Defined Networking

The SDN paradigm emerged to overcome the limitations of traditional network infrastructures (Kreutz et al., 2014). It enables new ways to design, build and operate networks by decoupling the control from the data plane and providing logical centralization of network control, management, and programmability in a fast and automatic fashion. According to the Open Networking Foundation (ONF), the SDN architecture consists of three planes: *Data, Control, and Application*.

However, the success of SDN relies on the ability of application developers to leverage the underlying network infrastructure to design and build new services, which relies on a *Northbound Interface* (NBI). This interface is the enabler for the realization of the ultimate SDN promise. Recognizing the significance of the NBI, the ONF created the *Northbound Interface Working Group*. Their aim for this group is to enable application developers and SDN users to focus on their applications, rather than concerning themselves with lower-level details and migrating between proprietary APIs (ONF, 2015b). Without an intuitive and efficient NBI, SDN will continue to struggle with gaining momentum with current network users and application developers who lack sufficient networking backgrounds. Despite the large effort of standardization and relevant progress in the study of the *Southbound* interface that connects the control and data planes, there has been less progress in the development of the NBI. Current efforts (depicted in Figure 2.2) are broadly summarized as:

- **SDN controller NBI:** the network is programmed using the controller's APIs which are low-level, limited, inflexible, and require many lines of code.
- **SDN programming languages** (Trois et al., 2016): use the controller's API to provide domain-specific high-level abstractions and constructs that focus on certain SDN aspects.
- **Intent-Based NBI:** applications express their high-level requirements as intents but are limited in capability and maturity.

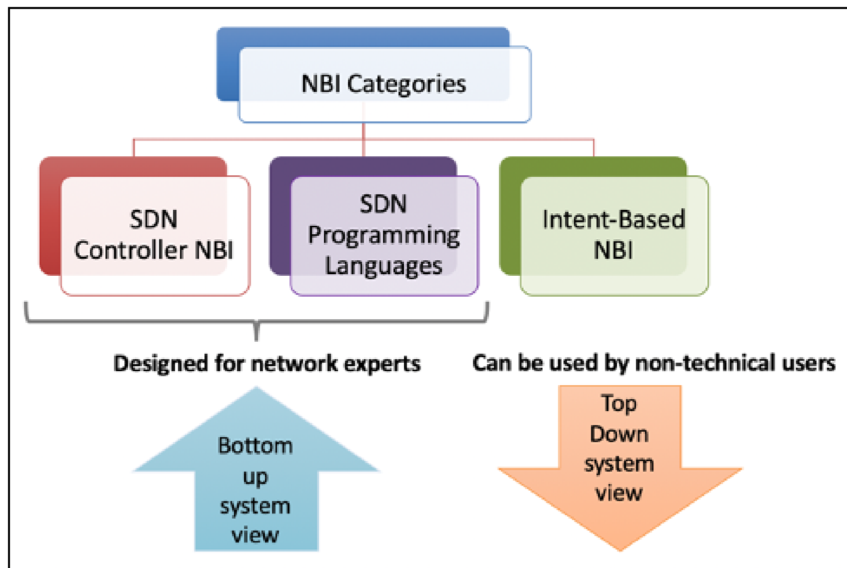

Figure 2.2: SDN Northbound Interfaces Categories

Current NBIs present challenges in the form of diverse capabilities, strengths, weaknesses and intuitiveness (Cox et al., 2017). In specific, SDN controller NBIs and SDN programming languages impose two main challenges. *First*, these are designed for network domain experts and require deep knowledge of network functionality, and programming constructs and abstractions. The *second* challenge is the tight coupling with SDN controllers, which complicates application portability between different SDN technologies, and requires mastering low-level details for each individual controller. Therefore, these NBIs in their current state fail to offer the means to create applications in a systematic and intuitive format that is also SDN controller-agnostic (Trois et al., 2016)(Lopes et al., 2015).

In contrast, Intent-Based NBIs can be used by network application developers/users who might not be network experts. They provide a more appropriate mechanism to express specific service demands (as *intents*) in an intuitive way. It

is, therefore, unsurprising that Intent-Based NBIs are gaining a lot of attention both in industry and academia.

However, most of the current Intent-Based NBIs are still limited in capability, some are ad hoc, and in some cases vendor-specific. They are unable to support emerging applications and services as most of them focus only on network-level operations, lack functionalities such as service partitioning and composition, and do not provide means to express high-level requirements.

2.3 Autonomic Networks

Owing to the need for a self-managed system that reduces the cost of having a well-skilled team that manages complex systems (Behringer et al., 2015), Autonomic Networks have emerged to shift away from traditional manual system configuration and management. The first clear vision of the autonomic computing systems was proposed by IBM (IBM, 2006). IBM characterized the self-management of an autonomic system in four self-* properties, namely: self-configuration, self-healing, self-optimization, and self-protection (Kephart et al., 2003), (Sinreich, 2006), (Parashar et al., 2004), (Sterritt, 2005), (Huebscher et al., 2008). These could be triggered by either a threat, an optimization purpose, or a change in high-level policies. We now review these self-* properties:

- **Self-configuration.** The system's ability to configure itself in response to its environment changes based on high-level policies that define what the system should do.

- **Self-healing.** The ability to detect problems, and (possibly) recover or correct itself by adjusting its components.

- **Self-optimization.** The ability to maximize resource utilization and monitor performance against an ideal case. The high-level policies that guide this may define a utility function for the system to prioritize some tasks over others (Khemka et al., 2014). Moreover, it may need to acquire/evacuate resources to achieve the optimal state.

- **Self-protection.** The ability to detect internal or external threats and protect its resources with appropriate defense actions to ensure its security and privacy.

Accordingly, IBM introduced the notion of the Monitor, Analyze, Plan, Execute, and Knowledge (MAPE-K) loop to allow the system to show the aforementioned and necessary self-* properties (Kephart et al., 2003). These functionalities are performed and controlled by the "Autonomic Manager", the main component of the autonomic system. The detailed functional components of MAPE-K are as follows:

- **Monitor.** It collects metrics and data from the managed resources or elements. This data could be the managed element's state, configuration, or the events generated by this element.
- **Analyze.** It analyzes collected data by the Monitor Function. However, the data size could be large, therefore, it may need to be filtered, normalized or processed, and structured in a way to be ready for analysis. This function analyzes and observes the system state. The analysis aims to detect if some policies were not met by the system, or if the system's performance was not as it should be. Accordingly, it issues a change with all necessary details and sends them to the Plan Function.
- **Plan.** Based on the requirement of changes indicated by the Analyze Function, a plan to change may be formed to the existing system's state or configuration, which defines the work to be done in a suitable form and sends it to the Execute Function.
- **Execute.** It performs a series of actions on the system according to the changes required by the Plan Function. An action could be changing the system's configuration or adding/removing resources, etc.
- **Knowledge.** The knowledge source contains the history of the collected or generated data by the above functions such as the system state, configurations, plans, policies, and actions.

Finally, it is worth mentioning that there is a difference between *autonomicity* and *automaticity* in network management. *Automaticity* means "automating routine tasks that are handled by the networks, either on a network element level (i.e., router) or at the administration level by using tools or scripts to automatically collect data from those elements through certain protocols (i.e., SNMP)" (Case et al., 1989). In contrast, *autonomicity* involves self-optimization of the network performance driven by high-level policies or goals.

2.4 Microservices Architecture (MSA)

In this section, we discuss this popular system paradigm which changed current systems management as opposed to traditional monolithic systems.

2.4.1 Background of MSA

There has been a significant current trend towards the MSA (Dragoni et al., 2017) especially in the cloud (Saboor et al., 2021), (Linthicum, 2016), (Qassem et al., 2022), (Villamizar et al., 2016), (Marie-Magdelaine et al., 2019), and (Singh et al., 2017),

which is a modular design pattern where complex systems are decomposed into a set of smaller microservices (which could be deployed as VMs or containers) that are minimal, complete, independent, and able to communicate with each other via universal APIs (i.e., REST API). These microservice components run each application process as a service that has been built for business capabilities and performs a single function well. Therefore, each service could be deployed, updated, and scaled to meet the demand for specific functions of an application.

MSA model splits the actors into three independent but collaborative entities: the *microservice application developers*, the *service brokers*, and the *service providers* (Tsai, 2005). The responsibility of *microservice developers* is to develop software services that are loosely coupled. The *service brokers* facilitate microservice publishing and discovery. The *service providers* find the available microservices through service brokers and use them to develop new applications. This process is done via discovery and composition rather than traditional monolithic design and coding (Tsai, 2005).

Consequently, the MSA model provides the following benefits:

Scalability: Due to the independence of microservices, each can be scaled independently to meet increased demand for the application feature it supports, without affecting other services.

Agility: MSA approach fosters an organization of independent small teams that own certain services. These development teams can work in parallel where small teams can move faster than large teams due to small and well-understood context. Since microservices are independent, it's easier to improve and maintain their code in shorter development cycles. It allows developers and engineers to focus on a single service and maintain and update code continuously using continuous integration/continuous delivery (CI/CD). This shortened development cycle times can significantly benefit organizations with the aggregate throughput of the overall microservices.

Resilience: Microservice independence increases an application's resistance to failure. This advantage occurs contrary to a monolithic architecture, where the entire application fails if a single component fails. On the other hand, with microservices, applications could avoid total service failure by degrading functionality without crashing the entire application.

Technology-independence: Developer teams are free to choose the best tool to solve their specific problems. As a consequence, they can choose the best tool for each job while building their microservices.

Reusability: The small, well-defined nature of microservice modules enables developer teams to use these functions for multiple purposes. A microservice written for a certain function can be used as a building block for another feature. This allows developers to create new capabilities without writing code from scratch.

Modularity: The modular design of microservices helps manage system complexity. It facilitates deploying, scaling, and updating individual microservices

independently, and avoiding long development cycles.

Although microservices offer great benefits, they come with their own challenges as well. They introduce a constantly evolving infrastructure of software components that are ephemeral and may change location, and communicate with each other in non-intuitive ways. In fact, the deployment of new microservice software releases to production environments can occur very frequently, with reportedly thousands of deployments per day (L. Chen, 2015), the challenge of managing and monitoring this large number of independent microservices can quickly become complex and require specialized tools and skills.

Accordingly, one of the popular approaches is the use of container runtimes and orchestration platforms such as Kubernetes (Kubernetes, 2023d), the de facto standard that provides a framework for managing containers, allowing developers to deploy, manage, and scale microservices with ease. Moreover, it provides features such as service discovery, load balancing, and rolling deployments that simplify microservices management.

However, in the CDN research field, MSA integration has not been discussed sufficiently. Adopting this approach in CCDNs, could lead to a faster and more independent deployment as the CP's demand varies (Chowdhury et al., 2019). These microservices (e.g. caches, load balancers, DNS servers, etc.) could be easily upgraded or scaled without affecting other functionalities. Moreover, unlike traditional CDNs, this approach facilitates the realization of the CDN functionalities in different ways depending on the underlying microservice implementation, which enables CDN operators to alternate between different microservices that provide the same functionality but with different costs, performance, etc., and compare them based on the CP's requested target. Therefore, articulating behavioral CDN abstract policies that could be implemented differently by microservices provides a more flexible, cost-effective, and portable solution that simplifies the CDN operator's task, and enables upgrading the CCDN with different and updated microservices, unlike traditional CDNs that have specific and predefined functionalities (Chowdhury et al., 2019).

2.4.2 Requirements Engineering in MSA

Requirements Engineering (RE) in a traditional cloud architecture is different from MSA. Applications and services in traditional clouds are owned by the cloud providers, so users can only execute what is offered, which could be very limited, stand-alone, and cannot be enhanced by the composition of multiple up-to-date microservices. Conversely, in MSA model, applications and systems could be constructed by selecting and composing different reusable and loosely coupled microservices. This selection

and composition could be made from within a large space of candidate microservices.

According to IEEE, a requirement in RE can be described as *a condition or capability to which a system must conform.* It is derived either from user needs directly, or stated in a contract, specifications, or standards (IEEEStandards, 1990). There are two types of system requirements: ***functional requirements (FRs)*** and ***nonfunctional requirements (NFRs)***. The former describes what the system is expected to do and its functions, and the latter defines how it will do it by dictating its general properties, which are also known as software qualities (e.g., scalability, security, efficiency, etc.). System requirements are usually specified by stakeholders and service providers based on empirical observation and/or technical expertise. NFRs at a high level (the whole system's level) lead to FRs at lower levels (i.e., microservices). For example, a scalability requirement, which is a conventional NFR may lead to microservices that scale the system as a whole which aims at scaling out the system under stressful loads. This scalability NFR could result in different FRs. In this case, an essential FR example is (a reactive load balancer). In more detail, the system shall balance the load reactively via a load balancing microservice (e.g., Least-connections load balancer) when stressful incoming load hits, which helps with the system's overall scalability. Other FR alternatives (i.e., load balancer microservices with other algorithms) could be deployed, compared, and alternated between with correspondence to their level of contribution towards the scalability NFR.

2.4.3 Multiple Criteria Decision Making (MCDM)

The incorporation of NFRs in a system typically involves trade-offs of some sort due to the often interaction between them. This is mainly in the sense that attempts to achieve one NFR could either help or hinder the achievement of other NFRs. Therefore, a decision support system is needed to help in a better trade-off among alternative functionalities in the potential solution space for the NFRs. Previous works have already identified some decision support systems and addressed some of the challenges of ranking and selection based on multiple criteria (Triantaphyllou, 2000). It is defined as a multiple-criteria decision-making (MCDM) problem. For example, selecting the most suitable microservice among a list of candidates based on several quality metrics.

Existing trade-off decision support systems are categorized as model-based or mathematical-based (Karlsson et al., 1998):

- **Model based** decision support systems rely on constructing a graphical model for illustrating the relations between trade-off entities. These models facilitate the system's information gathering and structuring. It also enables the distribution and communication of the gathered knowledge in the model between stakeholders in a convenient way. The most popular model-based technique is *NFRs framework*

(Chung et al., 2000) It identifies, handles, and illustrates the trade-off between soft goals (NFRs) and their operationalizations (solutions) expressed by stakeholders. In this framework, NFRs are represented as "softgoals" which have no clear-cut definition. NFR framework uses the term "satisficed" softgoals rather than "satisfied". The term "satisfice" was introduced by Herbert Simon (Simon, 1996), which is a decision-making strategy that attempts to meet criteria for adequacy but without necessarily producing an optimal solution. However, some of this technique's drawbacks is that it mostly handles qualitative information, so these models are not always suited to handle large amounts of data, as expressing and viewing it in a model can be cumbersome. Furthermore, these models do not provide quantitative results on a more detailed scale, typically absolute scale, ratio, or interval. This is a limitation when a concrete trade-off value is required, as the model-based trade-off techniques may not be able to reliably produce such a value. It is considered as a structured decision support material.

- **Mathematical based** decision support systems; on the other hand, rely on a mathematical formula for the trade-off construction and representation, thus enabling feeding the mathematical construct with appropriate values and receiving the best solution with regards to certain criteria (i.e., maximization, minimization, or optimal) (Berander et al., 2005). A popular and widely used example of these techniques is the *Analytical Hierarchy Process (AHP)* (Thomas L Saaty et al., 2012), (Whitaker, 1987), (Thomas L Saaty, 1985), (Thomas L. Saaty, 1994b), (Thomas L Saaty et al., 1991), and (Thomas L Saaty, 1990). Opposed to model-based techniques, mathematical-based trade-off techniques can handle large amounts of data and variables. It can also come up with results that are generally more accurate than common sense. Moreover, it enables structured analysis and repeatability since the process could be replicated over several rounds of data tweaking with respect to the system's needs. This could give an organization better consistency and overview (Berander et al., 2005).

2.5 Intent-Based Northbound Interfaces

Intent-Based NBI is a crucial mechanism that enables users to express what they want rather than how to do it via intents. It facilitates faster network management, service deployment, realization of user objectives and reduced OPEX due to less human involvement. Intent-Based NBIs can be used by network application developers/users who might not be network experts. They provide a more appropriate mechanism to express specific service demands (as intents) in an intuitive way. It is, therefore, unsurprising that Intent-Based NBIs are gaining a lot of attention both in industry and academia. However, although there has been a lot of attention around Intent-Based

NBIs that focus on solving and abstracting problems in the networking domain at a higher level (e.g., providing connectivity between endpoints, creating network slices, network service chaining, etc.), there has not been much progress in the development of Intent-Based NBIs that serve domains beyond networking and allow users to express high-level targets to capture their business objectives, nor linking them to lower-level management policies and operations beyond the networking domain. Hence, there is a need for different level of intent expressions and translations in the broader domains since most of the current proposed intent solutions and expressions cannot be utilized in this domain as they could be more prescriptive rather than declarative from the point of view of a generic non-technical user (i.e., unlike typical users in the networking domain who could have some technical expertise). Therefore, there is a need to investigate the utilization of intents beyond the networking domain.

2.5.1 Intent Standardization Efforts

There have been some intent standardization activities that have been analyzed and discussed in (Zeydan et al., 2020). Some of the considered scenarios are related to intent-driven network provisioning, network optimization, coverage, capacity management, and network automation. Recently, TeleManagement Forum (TMForum) has standardized an intent common model (ICM) in 2022 (TMForum, 2022). This model presents a basic intent description template, represented in a modeling framework named Resource Description Framework (RDF). It consists of a set of *expectations*, each of which is defined in terms of *parameters*, *targets* and associated *restrictions* for the network. Moreover, intent language models have been proposed with predefined vocabulary inspired by network or operational named entities (TMForum, 2021). However, these ongoing standardization activities propose intents as high-level goals but without enough specifications on how to execute their corresponding concrete actions (Zeydan et al., 2020). Today, regardless of the current efforts in IBN, most CCDNs still make their resources and caching management decisions oblivious to what the CPs want to achieve. For instance, the most popular CCDNs (e.g., Amazon CloudFront, Google Cloud CDN, and Microsoft Azure CDN) APIs do not allow CPs to express their high-level targets (e.g., requests/region) since they are mainly limited to selecting the desired geographical coverage and specifying the origin content server. Some of them offer APIs to configure some low-level technical details like the backend service load balancing, health checks, session affinity, connections per instance, etc. (Google, 2023b). However, generic CPs with no technical background and who are only interested in expressing their high-level intent would face some challenges with the current APIs. Therefore, Intent-Based CCDNs have to be further investigated.

2.5.2 Intents and Policies

It is important to highlight the difference between intents and policies as they should not be used interchangeably in this context as shown in Table 2.1. An Intent is a declarative expression by the user of the operational goals or expected outcomes that the system should deliver. Therefore, ultimately, intents could be used in many domains and by different users, even non-expert users who do not have a lot of system knowledge.

On the other hand, a Policy is a rule (or set of rules) that governs system behavior (Katchabaw et al., 1996). Typically, a rule consists of a variety of events, conditions, and actions (ECA), where events are rule triggers, conditions get assessed and if they hold then some actions are executed. Accordingly, a management paradigm emerged known as Policy-Based Management (PBM), which is prescriptive since policies let users (e.g., network operators, controllers) specify precisely what to do and under which circumstances (ONF, 2014).

Unlike intents, policies do not specify desired operational goals. However, both notions provide an abstraction of a network or system that does not necessarily involve technology-specifics. This is achieved by the separation of the rules that govern the system behavior from the functionality of the system. We elaborate on the differences between intents and policies with some examples that apply to the CCDN context in Section 3.2.

2.5.3 Different Intent Types

There are different perspectives on what an intent is depending on *whom* it is serving (i.e., technical or non-technical users), *how* it should be used (i.e., abstract or technology-specific), and for *which* domain (i.e., data center, cloud, etc.). Hence,

Policies	**Intents**
Prescriptive rules	Declarative expressions
Specify a set of ECA rules and determine precisely what to do under different circumstances and triggers (*how* to do?)	Express desired outcome (*what* to do?)
Used by system experts who can articulate the set of rules	Can be used by different users including service consumers and non-experts without enumerating rules
System behavior is defined proactively	Could be a learning reactive system

Table 2.1: A comparison between policies and intents.

there are different intent use-cases with different expectations, requirements, and priorities. Thereby, stakeholders consider an intent differently: ECA policy, business policy, network service, consumer service, etc. For instance, a data center network administrator (user with technical knowledge) can use an intent to **set the maximum load of a specific network link to be below 70%**. In contrast, in a CDN scenario, an intent user can be a content provider who wishes to start a caching service for certain content with **a target to serve 10,000 users/region**. From another perspective, intent user requirements can be classified as *client-facing* service-layer requirements and *operator-facing* resource-layer requirements. Network clients tend to express their high-level requirements in a service-layer expression that is concerned with performance expectations in accordance to the offered services (e.g., throughput expectation, availability time, etc.), whereas network operators use a resource-layer expression to express internal resource and operational requirements (e.g., energy-savings, cost management, etc.). Due to the inherent nature of each, service-layer requirements should be expressed in a declarative way that non-technical users can leverage, whereas resource-layer requirements are expressed in a prescriptive fashion (i.e., from non-technical users perspective) since the user has to describe their resource-level requirements that could be too technical and detailed for generic and non-technical users who are unaware of these levels, and mainly interested in expressing their overall service expected targets beyond the underlying resource-layer. Therefore, a mechanism is needed to decompose and translate high-level declarative intents to abstract prescriptive policies that define service behavior, which in turn gets translated to lower-level commands and resource configurations (as shown in Figure 2.3).

Figure 2.3: Intents, policies and operational commands.

2.5.4 Meta-Analysis for Intent-Based Northbound Solutions

We summarize the different Intent-Based solutions via a meta-analysis. The prefix meta-, when added to the name of a subject or a discipline, forms the name of a new subject that analyzes the first one at a more abstract or higher level. So, it is a field of study that analyzes already existing analyses. We include Intent-Based solutions that have been proposed by standard development organizations ((ONF, 2015a), (ONOS, 2015), (OpenDaylight, 2015), (OpenDaylight, 2016), (OpenStack,

Intent-Based Solution	Intent Expression	Domain	Level
(DOVE) by IBM (Cohen et al., 2013)	Not specified	Netw./NFV	–
Boulder (ONF, 2015a)	*Subject, Predicate, Object: {Constraints, Conditions}*	Netw.	Presc.
(NIC) by HP (OpenDaylight, 2015)	*Source Composite Endpoint, Destination Composite Endpoint, Traffic operation and constraints*	Netw./NFV	Presc.
ONOS Intent Framework (ONOS, 2015)	*Network Resource, Constraints, Criteria, Instructions*	Netw.	Presc.
Group-based Policy (OpenStack, 2016)	*Endpoint group, contract {subject: {rules: {classifier and action set}}}*	Netw./NFV	Presc.
(NEMO) by Huawei (OpenDaylight, 2016)	*Object + Operation* or *Object + Result* (under test and not used yet)	Netw.	Presc. Decl.
Intent-based virtualisation Platform (Han et al., 2016)	*Resources, Conditions, Priority, and Instructions*	Netw./NFV	Presc.
Service-oriented Intent-based NBI (Pham et al., 2016)	application-specific language	Netw.	–
(INSpIRE) (Scheid et al., 2017)	*Traffic Type, Source, Destination, Context level, Contexts list*	Netw./NFV	Presc.
Intent-based Negotiation (Marsico et al., 2017)	*Verbs, Nouns, Modifiers*	Netw./NFV	Presc.
(MD-IDN) (Arezoumand et al., 2017)	*Action, Endpoint 1, Traffic type, Endpoint 2*	Netw.	Presc.
Janus system (Abhashkumar et al., 2017)	*Endpoint-Group1, Connection attributes: {protocol, port, bandwidth, latency, middlebox }, Endpoint-Group 2*	Netw.	Presc.
Adaptive Service Deployment (Elkhatib et al., 2017; Elhabbash et al., 2018)	*Verb, Object, Modifiers, Subject*	General, e.g. storage, IDS	Presc.
(iNDIRA) (Kiran et al., 2018)	*Subject (Service or Condition), Relationship (has Arguments), Objects (multiple parameters)*	Netw.	Presc.
(SENSE) (Monga et al., 2018)	*Service type, Service alias, Connections: {name,terminals, bandwidth: { qos_class, capacity, unit}}, schedule: {start, end, duration}*	Netw./NFV	Presc.
Northbound Interface (Tuncer et al., 2018)	*Predicate, Commodity, Target (resources), Constraint, Condition*	Netw.	Presc.
OSDF (Comer et al., 2018)	*Instructions, Network Resource, Criteria, Constraints*	Netw.	Presc.
Customer-Oriented QoS (Beshley et al., 2020)	*QoE metric on a scale from 1 to 5*	General	Decl.
CompRess (X. Chen et al., 2020)	*SPARQL query: VNF Forwarding Graph, endpoints Ingress, Egress, VNFs to be traversed*	NFV	Decl.
VNF placement (Leivadeas et al., 2021)	*XML sytnax Service Name, Attributes QoS Level, Security Level, Start Time, Duration*	NFV	Decl.
Intents for Network Slice (Gritli et al., 2021)	*Network Slice QoE Requirements*	Netw./NFV	Decl.
Ontology-based Intent Refinement (Ouyang et al., 2022)	*Domain, Attribute, Object, Operation, Result*	Netw.	Decl.
Intent-based closed-loop (Baktir et al., 2022)	TM Forum RDF expression (TMForum, 2022)	Netw.	Decl.
Asset Administration Shell (Ustok et al., 2022)	TM Forum RDF expression (TMForum, 2022)	Netw.	Decl.
Intent-Driven Network Slicing (Xie et al., 2022)	TM Forum RDF expression (TMForum, 2022)	Netw.	Decl.
Intent-Driven Satellite Network (Li et al., 2023)	*Domain, Operation, Object, Outcome*	Netw.	Decl.
From Automation to Autonomous (F5G) (Zheng et al., 2023)	TM Forum RDF expression (TMForum, 2022)	Netw.	Decl.
Knowledge-based Intent Modeling (Mehmood et al., 2023)	TM Forum RDF expression (TMForum, 2022)	Netw.	Decl.
Intent Management in 6G (Daroui et al., 2023)	TM Forum RDF expression (TMForum, 2022)	Netw.	Decl.
Intent-based VNF Chains (Massa et al., 2023)	TM Forum RDF expression (TMForum, 2022)	Netw./NFV	Decl.

Table 2.2: Summary of the results of our meta-analysis of intent-based solutions.

2016), and (TMForum, 2022)) and academic researchers ((Cohen et al., 2013), (Han et al., 2016), (Pham et al., 2016), (Scheid et al., 2017), (Marsico et al., 2017), (Arezoumand et al., 2017), (Abhashkumar et al., 2017), (Elkhatib et al., 2017; Elhabbash et al., 2018), (Kiran et al., 2018), (Monga et al., 2018), (Tuncer et al., 2018), (Comer et al., 2018), (Beshley et al., 2020), (X. Chen et al., 2020), (Leivadeas et al., 2021), (Gritli et al., 2021), (Ouyang et al., 2022), (Baktir et al., 2022), (Ustok et al., 2022), (Xie et al., 2022), (Li et al., 2023), (Zheng et al., 2023), (Mehmood et al., 2023), (Daroui et al., 2023), and (Massa et al., 2023)). These solutions are listed in chronological order in Table 2.2 which shows an overview of our comparison based on the following criteria: *Intent Expression* shows how intents are expressed, *Domain* indicates the domain that is being addressed by the intent, namely, ***Networking*** (i.e., endpoints connectivity), ***NFV*** (i.e., virtualization operations, network slicing and service chaining) and ***General*** (beyond network connectivity and NFV operations), and *Level* classifies intent expressions as either prescriptive (Presc.) or declarative (Decl.).

By looking at the taxonomy in Figure 2.4, which has been derived from Table 2.2, we observe that most of the found intent-based solutions focused on the Networking and NFV domains with different levels of intent expressions. Although these works could provide important solutions to users (i.e., technical or non-technical), they are still restricted to these domains and naturally cannot be generalized to other problem domains due to the problem-specific intent expressions that aim at expressing endpoints connectivity, or virtualization operations. Therefore, we mainly focus on solutions that serve General domains beyond networking and NFV that could potentially facilitate the intent expression in our context which is concerned with the ”Declarative” level of intent expression since we assume that CPs are generic users with no technical background. However, by taking a closer look at Table 2.2, we can see that there are very limited solutions that address this area (Beshley et al., 2020), and their proposed intent expression does not facilitate capturing the CP's intent goal in a CCDN context.

In general, most of the current Intent-Based NBIs are still limited in capability, some are ad-hoc, and to an extent vendor-specific. Even though they offer important prescriptive policies for network services like end-to-end connectivity, chains of VNFs or virtual network slices, the majority of them still do not provide the ability to express declarative intents that handle other requirements beyond the network level.

We observe in Table 2.2 that the Intent-Based solutions from 2020 onward started to introduce *declarative* intent expressions. Although several more recent research in 2022 and 2023 have adopted the recent TMForum standardized ICM, they still haven't used it beyond the networking domain.

Moreover, most of the current NBIs offer a pre-defined set of intents and do not provide the tools to create new intents and translate/map them to lower-level

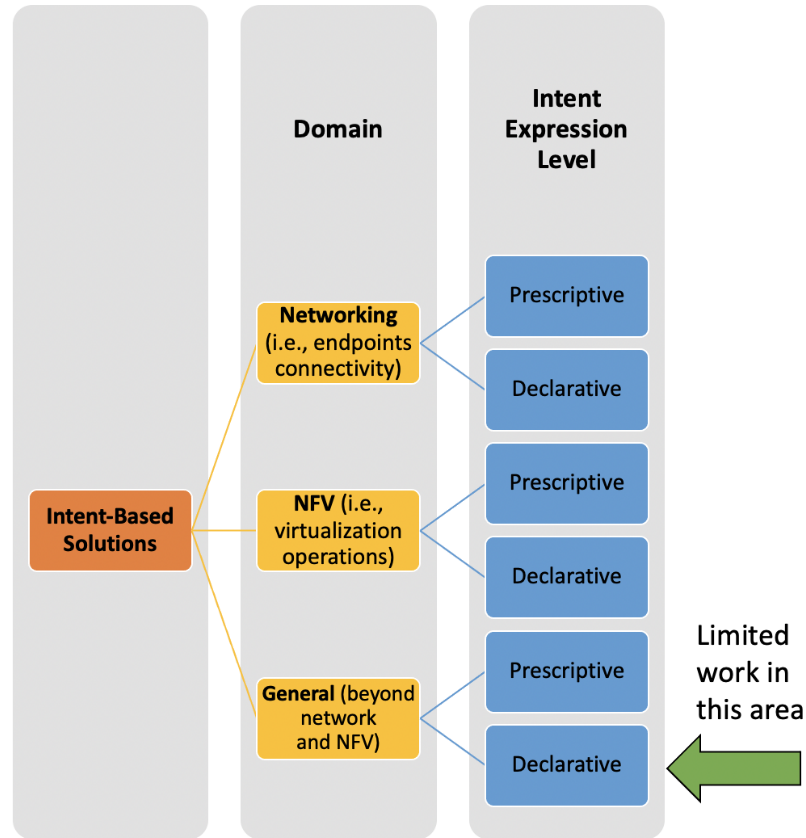

Figure 2.4: Intent-Based Solutions Taxonomy

policies (Alalmaei et al., 2020). The intent translation is still a challenging topic that needs to be investigated especially with the trending adoption of current technological advancements (i.e., microservices) that could adjust the control's level of granularity beyond the low-level networking commands. We discuss this further in Section 2.5.5.

2.5.5 Intent-Based Northbound Solutions Limitations

In this section, we present an overview of the current issues with Intent-Based NBIs and discuss the main challenges faced by intent users and developers.

- **Prescriptive Policy Expressions:** Service consumers require declarative rather than prescriptive policy expressions. However, most existing intent expressions are prescriptive, which are mostly used to express network-level requirements (OpenDaylight, 2015), (ONOS, 2015), (Scheid et al., 2017) and (Kiran et al., 2018).

However, there are more complex consumer requirements that are beyond network-level such as management intents (i.e. load balancing, placement, etc.) and, as such, cannot be easily expressed via prescriptive policies (Leivadeas et al., 2022). Therefore, consumers need an easier and higher-level expression that is purely declarative. This allows them to express the expected intent target without the need for stipulating any policies (Zeydan et al., 2020). For example, a consumer that requires a load balancing service can express a high-level declarative intent as **I want to load balance my traffic to handle 1,000 requests/second**. In this expression, the consumer only states the required target rather than some prescriptive operational policies.

- **Translation from Service-oriented Intents to Policies:** Proposing high-level declarative intents that express desired operational targets necessitates decomposition and translation processes to reach a representation that could both be understood and achieved by the system, yet abstract enough to govern the behavior of the system rather than its technical functionality (Ouyang et al., 2022), (Ouyang et al., 2021). The decomposed policies represent the expressive system capabilities, requirements, and constraints defined by system experts (i.e., network operator, service provider, controller, etc.). Afterward, policies get mapped to lower-level system operations. The benefit of having this intermediate level (abstract policies), sitting between high-level service-oriented intents and their equivalent low-level operations, is to allow system experts, service providers, and stakeholders (intent and policy developers) to think at a higher level independently from the underlying technologies. This reduces maintenance costs, improves operational scale, flexibility, adaptability, and reusability by allowing the same policy to be applied across different underlying technologies without the need to define multiple variants. Different policy agents can be plugged into the different underlying systems to translate abstract policies into a technology-specific representation(Zeydan et al., 2020).

- **Platform dependence and domain-specific limitation:** Intent-Based NBIs should be implemented as discrete engines lying above different platforms without being tightly coupled to them (Leivadeas et al., 2022) and (Pang et al., 2020). It allows a platform-independent NBI and releases the intent developers/users from the burden of using complicated NBIs that require a broader knowledge of the underlying platform functionalities and programming constructs. Moreover, extensibility is important to facilitate extending the current intents space by leveraging the composition of multiple policies to jointly create a higher-level intent, and leverage the current technology advancements (i.e., Microservices Architecture). Composing policies allows their re-usability to create more complicated but abstract intents. Therefore, suitable tools for policy composition (i.e., parallel or sequential)

have to be provided (Trois et al., 2016).

2.6 Different CDN Flavors

There are different variations of CDNs as surveyed in (Jia et al., 2017), for instance, different criteria can result in different CDN flavors (shown in Figure 2.5) as described below.

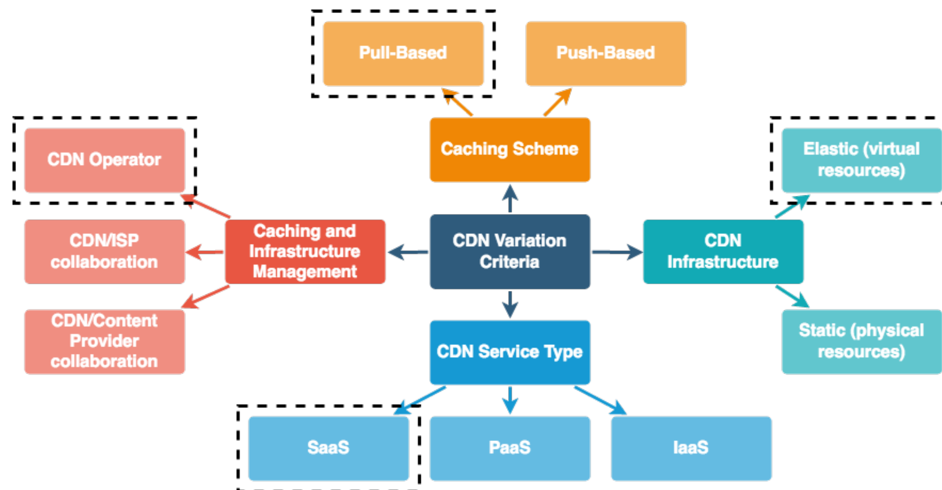

Figure 2.5: CDN Variation Criteria

- **CDN Service Type:** Determines the caching service level (i.e., software, platform, infrastructure). In the case of Software-as-a-Service (SaaS) level, the CP can request the caching service in a higher-level without getting involved in the cache management, caching mechanism, infrastructure management, etc. The CP delegates the caching responsibility to the CDN. On the other hand, Platform-as-a-Service (PaaS) CDN allows the CPs to deploy their caching service and provide them with the platform and cache management tools to assist the caching process. As for the Infrastructure-as-a-Service (IaaS) level, the caching service infrastructure (i.e., physical or virtual) is provided to the CP, who would be responsible for the whole caching and infrastructure management.

- **Caching and Infrastructure Management:** Decides the different possible roles and collaborations in managing the caching and CDN infrastructure. The CDN could be managed exclusively by the CDN operator where the CP delegates the caching responsibility to the CDN operator without getting involved in the decision-making and management process. Alternatively, another management

model requires collaboration between the CDN operator and the CP for better caching decisions depending on the additional information provided by the CP (e.g., requests pattern, request history, users locations, etc.). Moreover, another management collaboration model exists, which enables the CDN operator and the ISP operator to collaborate for better and more efficient caching and infrastructure management (Frank et al., 2013), by leveraging the ISP's direct access and control over the infrastructure, and the awareness of the users' requests details.

In these collaboration models, the knowledge-sharing of one stakeholder could help with the decision-making of the other. Given the different levels of visibility for each, a stakeholder could leverage the better system/resources exposure of another stakeholder to make better decisions.

- **CDN Infrastructure:** Provides either physical or virtual infrastructure resources. Physical resources could be mostly static and require longer time scales to scale-up (i.e., days or weeks). Conversely, virtual resources are dynamic and leverage the virtualization capabilities, hence, they require much shorter time scales to scale them out/in (i.e., hours or even minutes).
- **Caching Scheme:** Could be either pull-based or push-based. The most popular caching scheme is pull-based, which pulls the contents to the caches from the origin server reactively based on the end-users requests. Whereas the push-based scheme preemptively stores contents to meet estimated demand.

2.7 CCDN Operations

CCDNs typically manage several operations to deliver their caching services (Frangoudis et al., 2017), these operations can be broadly listed as follows:

- **Resource Allocation:** To allocate the cache servers based on several factors like the geographical coverage requested by the CP, available resources, requested demand, etc. This operation can be deployed using different underlying technologies (i.e., virtualization or containerization).
- **End-Users Content Request Redirection:** After fetching the contents from the origin server to the local caches, end-user requests are redirected to the local caches via different approaches like traditional DNS-based or SDN-based. The former (i.e., Domain Name System) redirects traffic to its destination domain name rather than its IP address, which simplifies reachability for users since they send their requests to the destination name, and the DNS dedicated server translates it to its corresponding IP address (Frank et al., 2013). Whereas opposed to the

former traditional hardware-based approach, the SDN-based approach is a software-based traffic redirection that relies on the decoupling between the data plane and the control plane, and provides centralized control over all forwarding elements that could run on commodity hardware rather than dedicated ones. The unified logical view and centralized control over the data plane facilitate making better request forwarding decisions to the caches based on different goals (i.e., revenue maximization) (Duan et al., 2018), and (i.e., cache response time) (Tran et al., 2019).

- **Caching Service Scaling up/down:** Based on the demand, user targets, current system state, available resources, etc. This could be achieved by leveraging the virtualization or containerization capability, and direct access and control of the infrastructure and caching service. Consequently, the load distribution across the available caches is maintained by the load balancer which works in conjunction with service scaling and tries to avoid overloading individual caches and ensures that users can access the application without experiencing downtime or performance issues. These load balancers can be configured to use different algorithms, such as round-robin, IP hash, and session affinity, to distribute traffic based on specific criteria.

2.8 CCDN Use Case Assumptions

For demonstration purposes, we focus on the CCDN flavor with the features surrounded by the dashed boxes in Figure 2.5.

Adopting CCDN solutions claims to offer cost-effective solutions on a pay-per-service basis. Moreover, hosting CDN services over the cloud increases the availability of the services exponentially (Jayakumar et al., 2018). Therefore, our CCDN use case will be based on the following assumptions:

1. CCDN service type is *SaaS*, which provides the CP with a software-based caching service that hides away all of the infrastructure and platform details. The CP can express his expected targets which could be demand requirements or QoS requirements.

2. CCDN provides an *elastic virtual* resources that could dynamically scale out/in based on the demand, system status, etc.

3. The caching and infrastructure management in the CCDN is done by the *CCDN operator* who has direct access and control over them, so the CP delegates the caching decisions to the CCDN operator.

4. The caching scheme is *Pull-based* (like most of the existing caching solutions), which leads to a reactive content placement in the caches according to the end-users demand.

In this use case, we will discuss how the CP and a CCDN operator can communicate via our proposed intent expressions and framework in a CCDN scenario. CPs tend to rely on the CCDN to manage content caching and end-user traffic redirection. However, currently, CPs only set up and specify content servers (content origin) and the geographical coverage that they are interested in. So far, current popular CCDNs like Amazon CloudFront (Amazon, 2023), Google Cloud CDN (Google, 2023c), and Microsoft Azure CDN (Microsoft, 2023b) do not consider the high-level CP targets (e.g., requests/region, latency, etc.) in their caching management. Therefore, in a next deployment phase of a CCDN, it is important to consider them for better caching decisions and more involvement of the CPs in the process.

2.9 CCDNs' Related work and Used Technologies

Current CCDN related works leverage several technologies such as Cloud Computing, NFV, SDN, Multi-access Edge Computing (MEC), Blockchain, Named Data Networking (NDN), and Containerization. These works are listed in chronological order in Table 2.3.

CCDN	Contribution	Cloud Computing	Virtualization	SDN	Edge Computing	Named Data Netw.	Blockchain	Containerization
Cloud-Oriented CDN (Papagianni et al., 2013)	It presented a scalable hierarchical framework over a multi-provider cloud by decomposing the problem into defined graph partitioning and replica placement	✓	✓					
Virtualized Programmable CDN (Woo et al., 2014)	a CCDN that is deployed via SDN and NDN to achieve programmability in content delivery	✓	✓	✓		✓		
Server Selection (Roy et al., 2015)	a Server Selection algorithm and simulation using Voronoi Diagram And Fuzzy Based Dynamic Load Balancing	✓						
Joint Content Replication (Hu et al., 2016)	a social video distribution over a CCDN, that minimizes the cost, and satisfys the averaged time delay	✓		✓				
Optimal Planning under Uncertain Traffic (Mangili et al., 2016)	a two-stage stochastic optimal planning model for CDN operators to deploy physical and/or virtual CDN nodes		✓					
Storage Cost Optimization (Sajithabanu et al., 2016)	a content placement and delivery in CCDN based on a Genetic Algorithm with an efficient storage model		✓					
CDN Slicing (Retal et al., 2017)	formulated the VMs placement problem as two Linear Integer problem solutions, to minimize the cost and maximize the QoE	✓	✓					
CDN-as-a-Service (Frangoudis et al., 2017)	a (CDNaaS) architecture that allows a telecom operator to offer different CDN flavors to CPs on demand	✓	✓					
2-Tiered Cloud CDN (Gupta et al., 2017)	a 2-tier structure, which uses CCDN with priority based round robin scheduling, and a dedicated Peer to Peer network for video delivery	✓						
Optimal VNFs placement (Benkacem et al., 2018)	a placement model of VNFs for CDN slices to meet performance requirements with minimum cost, across multi-domain cloud	✓	✓					
Software-Driven CDN (Duan et al., 2018)	an SDN-based CDN that achieves a win-win situation between IP and CPs based on a suitable revenue sharing scheme			✓				
Video Transcoding (Minggang Chen et al., 2018)	a data-driven parallel video transcoding for CDN	✓						
Content Rating (Deep et al., 2018)	a content filtering technique coupled with weighted slope one scheme to improve network latency							
MABRESE (Tran et al., 2019)	a server selection algorithm with an average response time and reward score for an SDN-based CDN			✓				
Resource Reservation (Fan et al., 2019)	a multi-objective optimization problem on resource reservation for a CCDN, to seek a trade-off between the rental cost and the user experience	✓						
BCDN (Ak et al., 2019)	a Blockchain-aided CDN model to prevent the overloading on the surrogate servers and provide dynamically changing virtual instances						✓	
CDN Slicing (Taleb et al., 2020)	a design and mechanisms of dynamic instantiation and management of CDN slices in 5G mobile networks with respect to QoE	✓	✓		✓			
WAE (Ak et al., 2018)	a Workload Automation Engine which enables dynamic resource management and scaling, with the least cost							✓
Edge vCDN (B. Chen et al., 2023)	architecture and a dynamic deployment method for vCDNs based on edge computing to address the connectivity issues caused by real-time hot content		✓		✓			✓

Table 2.3: A comparison between current vCDN solutions.

Cloud Computing allows leasing resources (i.e., compute, memory, storage, and bandwidth) to build CDNs in the cloud to alleviate major limitations of traditional CDNs that rely on fixed physical caches, by leveraging virtualization technologies (Benkacem et al., 2018), (Taleb et al., 2020), (Papagianni et al., 2013), (Gupta et al., 2017), (Roy et al., 2015), (Minggang Chen et al., 2018), (Deep et al., 2018), (Sajithabanu et al., 2016), (Retal et al., 2017), (Frangoudis et al., 2017), (Woo et al., 2014), (Fan et al., 2019), and (Hu et al., 2016). **NFV** technology allows the decoupling of CDN functionalities from the underlying hardware, which offers on-demand scaling and automatic reconfiguration (Benkacem et al., 2018), (Taleb et al., 2020), (Papagianni et al., 2013), (Retal et al., 2017), (Frangoudis et al., 2017), (Woo et al., 2014), (Mangili et al., 2016), (Hu et al., 2016), and (B. Chen et al., 2023). The **SDN** paradigm provides logically centralized control, management, and programmability over the network (Woo et al., 2014), (Duan et al., 2018), and (Tran et al., 2019). **MEC** harnesses the power of cloud computing outside cloud data centers by deploying application services at the edge of mobile networks (Taleb et al., 2020), and (B. Chen et al., 2023). **Blockchain** methodology provides features of distributed contracts and fully automated reliable flow generation to prevent overloading on the surrogate servers (Ak et al., 2019). **NDN** is a network architecture that directly uses application data names to communicate, treating networking, storage, and computing resources in the same manner, by adopting a request-reply communication model that directly uses application data names at the network layer (Woo et al., 2014).**Containerization** provides lighter-weight virtualization (Ak et al., 2018).

Even though the aforementioned technologies have been leveraged to solve specific CDN problems; server selection, placement, traffic routing, etc., they still have not considered the communication gap between the CP and the CCDN, which allows the CP to request high-level intent targets.

Therefore, adopting intents in the CCDN domain would lead to better interaction and decision-making between the CP and the CCDN operator. Moreover, there are no sufficient works that explore how microservices could be deployed in a CCDN, and how can a CCDN operator design and integrate a set of microservices that collectively comprise the whole CCDN along with their varying interactions and performance. This could be leveraged and deployed as part of the translation and decomposition process from high-level declarative intents to their corresponding lower-level policies that would map eventually to different microservice alternatives.

It is through this process that we have identified some limitations in the area (Alalmaei et al., 2020). These are as follows:

- The limited involvement of CPs in the CDN decision-making and rigid one-directional communication between the CP and the CCDN. Current solutions focus on the system more than the user.

- CPs currently cannot express their high-level intents to the CCDN. Instead, current CCDNs like Amazon CloudFront (Amazon, 2023), Google Cloud CDN (Google, 2023c), and Microsoft Azure CDN (Microsoft, 2023b) have very limited APIs that allow them to identify the desired region and the cached content. Some offer more technical APIs that include some level of caching and infrastructure configurations (Google, 2023b). However, these are not meant to be used by non-technical CPs who are not interested in this level of detail.

- To the best of our knowledge, there is no sufficient research in the literature that investigated the adoption of microservices in the CCDN context. Since many traditional CDNs may fall behind with their pre-defined caching and management decisions, it is very important to explore how can a CCDN operator design and integrate a set of microservices that collectively comprise the whole CCDN along with their interactions and performance with respect to the CP's intents.

- Intent translation has been mostly presented as a black box that has not been discussed in a systematic way in the current literature, so, it is essential to discuss how the translation and decomposition processes are held which convert high-level declarative intents to their corresponding lower-level policies that would map eventually to microservices.

2.10 Summary

This chapter has presented a landscape of how CCDN operators could leverage some technologies, background material, related work, and their current limitations. To begin with identifying a particular trend in the field of networking, which moves towards *Softwarization*: moving away from fixed hardware-based functionality towards more flexible and agile software alternatives. Importantly, this is achieved with improved scalability, availability, cost-reduction, resource utilization and agility. This softwarization movement has progressed over the past few years and become widely adopted in industry and academia. We discussed NFV, Containerization, and SDN, along with some of their advantages, challenges and limitations.

Another technology described in this chapter is the *Microservices Architecture*. In particular, we compared its advantages against the traditional monolithic systems. We presented a set of benefits offered by this technology that improved today's systems. However, it also induces some challenges that could be somewhat partially relieved via the current microservices orchestrator de facto Kubernetes.

Later, based on the previous technologies, a recent trending paradigm has been discussed in detail. The *Intent-Based System*. In specific, we broke down the current Northbound Interfaces that are available for service consumers and compared them.

Opposed to the Intent-Based Northbound Interfaces, the others are still low-level, technology-specific, and ad-hoc which were not designed for novice users with no technical background. Then we compared intents and policies, and we even described different types of intents based on their users and problem domain. Therefore, we focused on the current Intent-Based solutions that aim to allow users to express their intent targets at a high level. However, these solutions are still lacking behind their expected outcome. We provided a meta-analysis of the most important Intent-Based solutions and summarized their shortcomings.

Moving forward to *CCDNs*, we introduced our classification of CDN types based on several dimensions like their service type, infrastructure, caching scheme, etc. Accordingly, based on our classification for CCDNs, we stated our own assumptions for the CCDN use case that we focus on and work on.

Finally, we focused on the current *related work in the CCDN research field*. A meta-analysis was done for the current CCDN solutions, their adopted technologies, and the problem description. We discussed their overall limitations that motivated us to take a step forward towards future CCDNs by leveraging the current technology advancements, which have not been widely investigated in the current research in the CCDN domain.

Chapter 3

Design

Today, most CCDNs make their resource and caching management decisions in a way that is oblivious to what the CPs want to achieve. Their APIs do not allow CPs to express their high-level targets (e.g., expected requests throughput, latency, cost saving, etc.) since they are mainly limited to selecting the desired geographical coverage and specifying the origin content server. In light of the expected growth of the Intent-Based Networking market with a CAGR of 23.9% to reach $8.8 billion by 2030 (GlobeNewsWire, 2023b), it is important to investigate how the next deployment phase of CCDNs could maintain or accelerate this growth rate by moving towards an Intent-Based NBI interactive communication scheme between CPs and CCDN operators, which enables CPs to declare their high-level targets and get updated on the CCDN's ability to achieve these targets throughout the service lifetime. This could possibly lead to better and more efficient resource and caching management decisions by providing more adaptive, flexible, and portable CCDN that tries to meet the CP high-level target in an automated way throughout the pre-, post-, and during the service deployment phases. Moreover, many CCDNs offer pre-defined caching and resource management that could possibly fall behind the rapid technology development and dynamic demand changes.

In this chapter, we discuss the motivation behind the need for Intent-Based solutions central to both present and future CCDNs design in Section 3.1. Then, in Section 3.2, we present our proposed multi-level intent expressions. Further to this, we present our Intent-Based framework for achieving this, illustrated through a comprehensive design in Section 3.3. Following that, in Section 3.4, we discuss the MCDM process in detail, which assists with the CCDN deployment selection based on the intent target, In Section 3.5 we introduce different intent targets that could be interesting to CPs. Finally, we discuss the intent refinement process which aims at improving the intent performance and map it to a MAPE-K autonomic approach in Section 3.6.

3.1 Design Motivation and Aims

Given the background and related work described in Chapter 2, it is clear that there is significant effort centered around CCDNs, and the technologies and infrastructures that support them. Although the discussed technological advancements are very helpful in improving the current CCDNs, there is still a gap between the CP's requirements and the CCDN requirements and goals. This is because most CCDNs still make their resource and caching management decisions oblivious to what the CPs want to achieve. Hence, there is a current interest in system management schemes that utilize user's high-level intents. In this section, we consider the motivation behind an Intent-Based CCDN design and the aimed features for each.

3.1.1 Leveraging CDNs by different domains

CDN services are not limited to content delivery, in fact, a spectrum of other domains can leverage CDNs as well such as e-learning (Palau et al., 2003), smart cities (Mingkai Chen et al., 2019), smart health (Min Chen et al., 2017), drone and Unmanned Aerial Vehicle (UAV) monitoring, and vehicle monitoring (Asheralieva et al., 2019). However, advances in these domains are not sufficiently supported by academic research in several aspects as discussed in this survey (Zolfaghari et al., 2020).

From the perspective of a CDN operator, administrator, and IT personnel in general, multiple technology advancements could be leveraged. For instance, SDN (Kreutz et al., 2014) and NFV (Mijumbi et al., 2015) have recently redefined the vision of designing, deploying, and managing networking systems and services. Combined together, they provide network managers with complete, programmatic, and flexible control of a dynamic view of the network which could greatly improve the performance and management of the aforementioned domains. Therefore, Telecommunications providers, CDN operators, and over-the-top (OTT) content providers have taken a great interest in leveraging these technologies (Alalmaei et al., 2019).

Although these rising paradigms are very promising, these abundant possibilities have imposed additional challenges by creating a cumbersome system configuration process to adjust to all different stakeholders, users, and services. Therefore, lately, there is a need to simplify the management and configuration of networks in an autonomic way. Intent-Based Networking (IBN) is such a paradigm that realizes simplified, flexible, and agile network management and configuration with minimal external intervention (i.e., from the CP's perspective) (Leivadeas et al., 2022).

Although today's networks are programmable (Lopes et al., 2015). (Trois et al., 2016), (Parashar et al., 2004), (Comer et al., 2018), and (Woo et al., 2014), they need programmers. Current Network programming approaches allow the definition

of policies (conditional actions) to be executed in pre-defined scenarios. IT experts need to define when and what the network functions should do in low-level details, at multiple resource and technology layers, and on massive scales. However, the current network entities do not have a full representation of the overall goals (Szilágyi, 2021). Therefore, these raw network programmability approaches are not sufficient to achieve a high level of automation, that is, where automation goals are beyond configuration parameters that can be tuned and committed (Szilágyi, 2021).

Conversely, Intent-Based networks allow the definition of much higher level and, sometimes abstract objectives, without requiring instructions on how to reach them. Accordingly, the comprehension of the intent and its mapping to lower-level technology-dependent actions that make sense at any moment is needed for bridging the abstraction and automation gap between the intent and current systems (i.e., policy-based or workflow automation). This is a novel requirement compared to these non-Intent-Based network solutions. However, existing Intent-Based proposals have usually a narrow domain-specific and use-case-driven automation scope, which leaves the gap between intent abstraction and automation open (Szilágyi, 2021).

However, generalizing an intent expression could miss capturing some domain-specific requirements, especially with the different levels of possible intent users (i.e., technical and non-technical). Therefore, it is important to allow the customization of this generic expression and extend it with some additional specific requirements with respect to the problem domain and users.

Therefore, this surely requires additional capabilities within the IBN itself (i.e., a level of intelligence relevant to a certain context), knowing the potential actions available to manipulate the scenario and help with reaching the goal, and being aware of the expected and resulting outcome of each action in the given context (Szilágyi, 2021).

Since CDNs are being utilized in multiple other domains beside content delivery, this imposes an additional burden on the CDN operators due to the wider variety of use cases, service requirements, end-user requirements, etc. Hence, CDNs need to consider solutions that are able to handle these variable cases and requirements of these diverse domain problems that should reflect on the utilization and management of the underlying technologies. Therefore, a communication scheme is needed to enable multiple users from these different domains with different levels of technical knowledge to interact with the CDN in a sufficient and dynamic manner with correspondence to their domain-specific requirements.

Accordingly, a new CDN solution should be able to embrace current technologies and adopt new ones as well. Therefore, technology independence is key. Seeking a modular solution that separates the system services from the underlying technology allows stakeholders to agree on behavioral policies and even compare and alternate between technologies. For instance, CDN microservices (e.g. caches, load balancers,

DNS servers, etc.) could be easily upgraded or scaled without affecting the other functionalities. Moreover, this facilitates the realization of the CCDN functionalities in different ways depending on the underlying microservice implementation, which enables stakeholders to compare and alternate between different microservices that provide the same functionality but with different costs, performance, etc.

Hence, these CDNs could leverage some decision support systems which are defined as a Multiple-Criteria Decision-Making (MCDM) to establish alternatives ranking and selection based on multiple criteria. Realized via different approaches, this can help to negotiate requirements among stakeholders and analyze architectural and services trade-offs based on some evaluation criteria (Junior et al., 2018)

3.1.2 Different CDN stakeholders' collaboration

The collaboration between multiple stakeholders in the field of CDNs has attracted a lot of attention both in academia and industry. These stakeholders could be Internet Service Providers (ISPs), Cloud providers, CDN operators, content providers, Peer-to-peer (P2P), and several academic research and industrial groups. This survey (Jia et al., 2017) discussed the current state of different collaborations in content delivery and discussed thoroughly the current related works. This collaboration facilitates making full use of the management, infrastructure, and effective information provided by these stakeholders to improve the efficiency of content distribution and optimize the overall performance of the network. This is due to the diverse levels of control, exposure, and role with respect to CDN management and infrastructure. For instance, in order to optimize the content distribution performance, CDN providers/operators who usually have less underlying network information, can leverage the ISP's control and exposure of the underlying network topology, link load information, and user location as shown in Figure 3.1. Therefore, exploring these new efficient content distribution schemes could reduce the time delay for request and response, and improve user quality of experience (QoE).

Furthermore, different CDN providers may have access to different content at different costs, and dynamic demand. So, different business relationships can be built between CDN providers. There has been an ongoing effort in the CDN Interconnection domain (Bertrand et al., 2012) that tries to specify the communications and roles between different CDNs and ISPs to reduce CDN traffic and decrease latency where several use cases have been discussed. However, these different facets of collaborations between these stakeholders, their corresponding underlying microservice mapping, etc., are beyond the scope of this thesis.

Accordingly, a communication scheme is needed to enable these different stakeholders to express their high-level requirements via intents (as intent consumers) that should be handled by other stakeholders that have the required level of control (as

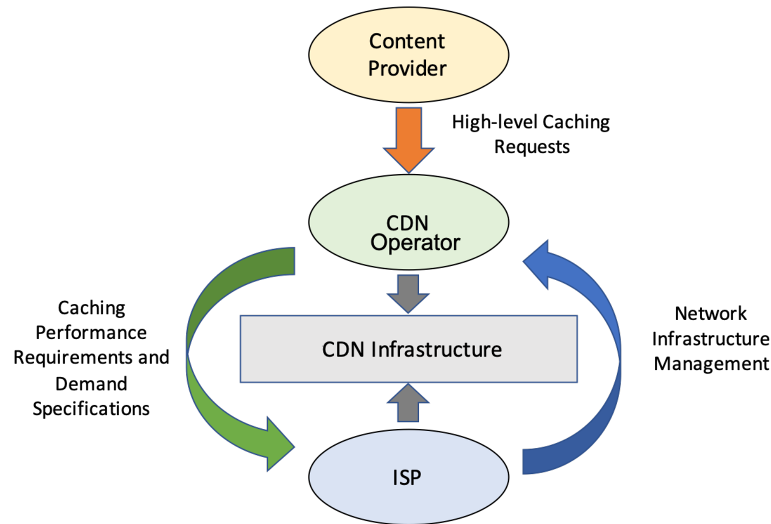

Figure 3.1: CDN Operator and ISP Collaboration.

intent developers). This communication also needs to keep the consumers involved throughout the whole life cycle of the intent by sharing feedback and getting their potential input. A stakeholder could play both roles; *intent consumer* and *intent developer*. For example, a CDN provider/operator could be a *consumer* of an intent developed by an ISP, and on the other hand, a CDN provider/operator could be the *intent developer* that creates intents for the content provider's consumption.

Therefore, an Intent-Based solution needs to provide multiple levels of intent expressions; *declarative* and *prescriptive*. The former is targeted for non-technical consumers to state their high-level target sufficiently, whereas the latter is meant to be expressed by technical users with the additional required technical specifications and details.

3.1.3 Bi-directional interaction between intent consumers and the CDN

CDNs are not confined to static content delivery as they are used in dynamic content delivery which requires mutual interaction with the user. Their application is not limited to Business-to-Business (B2B) interactions between telecommunications operators and CDN operators but also extends to Business-to-Consumer (B2C) business models as CDN operators and content providers are more prevalently adopting this technology (Zolfaghari et al., 2020).

Knowing these potential CDN actions leads to providing the necessary bi-directional Intent-Based APIs that involve both parties in the process (each with their

corresponding level of involvement) for resource and domain management, Quality of Service (QoS) resource provisioning and changing, slicing, and more. A full closed loop of intent-based interaction is depicted in Figure 3.2. This loop starts from the intent consumer who issues the intent request via different input methods (e.g., commands, selected choices from a list, natural language text, voice commands, etc.), and then this input has to be analyzed and processed (i.e., translated) for validation. If the intent input is invalid, then the intent consumer has to re-input it until a valid form is entered. Once the intent translation has been completed, it gets built and deployed based on the underlying infrastructure technology. Finally, feedback is shared with the intent consumer.

Accordingly, new Intent-Based APIs should allow incremental development by reusing the existing APIs (e.g., policy-based) that are currently exposed to human experts. However, in order to maintain the intent target, APIs need to derive insights and actions through measurement/data collection, configuration settings, and bi-directional communication with the intent consumers who can provide their insights, additional information, attributes, and elaborated specifications of their intent goal.

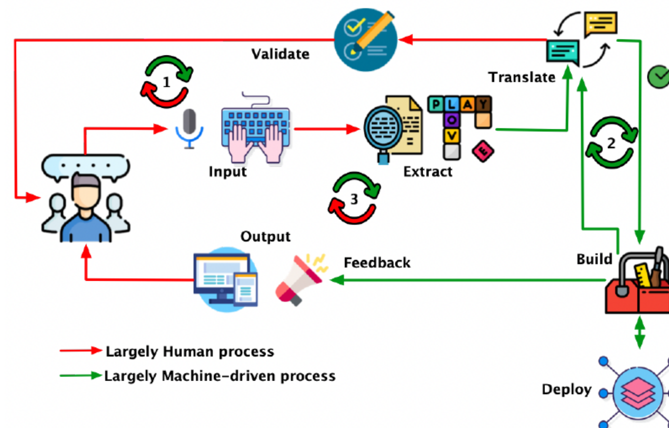

Figure 3.2: Bi-directional Interaction between users and the Intent-Based system (Bezahaf et al., 2021)

3.1.4 Summary of Motivating Factors

In this section, we have identified the motivation and a number of features to be considered in the design and implementation of an Intent-Based CCDN. Below, we collect and provide a summary of each of these motivating factors and their aimed design attributes.

1. **Leveraging CDNs by different domains**

- Enable multiple CDN users from different domains, even beyond content delivery, to interact more efficiently and easily with the CDN
- Extend the programmability approaches of the current networks via IBN
- Build a CDN that could embrace current technologies and adopt new ones as well by considering modularity, extensibility and technology independence
- Allow stakeholders to plan behavioral policies at an abstract level, but also compare and alternate between the underlying technology realizations at the lower levels, via some MCDM approaches

2. **Different CDN stakeholders' collaboration**

- Facilitate stakeholders' collaboration via intents
- Introduce different levels of intents/policies for different stakeholders according to their level of exposure, management, expertise, etc. Each with the suitable corresponding expression (i.e., declarative or prescriptive)
- Foster the hierarchical relationships between stakeholders via different intent-related roles (i.e., intent developer and intent consumer)

3. **Bi-directional interaction between intent consumers and the CDN**

- Provide a bi-directional interaction between different CDN players (e.g., stakeholder-to-stakeholder, CDN-to-CP, etc.)
- Integrate the bi-directional interaction along with other factors (i.e., metrics/data collection, service status, ML/AI, system configurations, etc.) to allow intent users to provide their insights, additional information, specifications of their intent goal updates, etc. for a potentially more efficient and interactive Intent-Based communication scheme

3.2 Intent Expressions, Syntax, and Descriptors

Service-oriented intents are high-level descriptions, that are targeted towards service consumers without assuming them to have deep knowledge or technical details about it. We devise a well-defined declarative expression that can represent user targets in light of the expression proposed in (Chao et al., 2018). The intent syntax is in the form of:

```
<SERVICE><RESOURCES><CONJUNCTION><TARGET>
```

<SERVICE> identifies the required service (e.g. caching, placement, etc.). <RESOURCES> identify the abstract resources the service would operate on (e.g. caches, switches, etc.). <CONJUNCTION> expresses the conjunction between the service and the target (e.g. 'with', 'to', 'that can handle', etc) to make intents more readable. <TARGET> expresses the objective of the required service that users want to achieve.

A service-oriented intent <TARGET> is decomposed into a set of prescriptive policies that work jointly to achieve the target as:

```
<SERVICE><RESOURCES>
    <POLICY1><OPERATOR><POLICY2><OPERATOR>...
    {<POLICYk1><OPERATOR><POLICYk2><OPERATOR>...<POLICYkm>}
    <OPERATOR>...<POLICYn>
```

Each <POLICY i> is expressed as:

```
<CONDITIONS><ACTIONS><CONSTRAINTS>[<PRIORITY>]
```

where <CONDITIONS> define the circumstances under which the service actions will apply. <ACTIONS> represent the type of service actions to be executed. <CONSTRAINTS> show the constraints on the actions to execute, while [<PRIORITY>] is an optional indicator of policy priority. <OPERATOR> is the logical operator that identifies how any two policies interact with each other, either in a sequential or parallel manner. The curly brackets ({ }) express a block of grouped policies.

There are two levels of policies (i.e. abstract and technical). The abstract policies are listed in the **Intent and Policies Descriptors** Component in Figure 3.4. Meanwhile, the technical policies are instantiated policies with certain calculated values in place of the abstract ones before the translation process. They both follow the same syntax but the technical policies determine the microservice alternatives or technologies used in <ACTIONS> and the corresponding technical requirements and operational parameters in <CONSTRAINTS>. An example of these policies will be discussed in the following Section.

3.2.1 Intent-to-Policy Mappings

CCDNs manage several operations to deliver their caching services (Frangoudis et al., 2017), broadly categorized as:

1. **Allocate resources** to cache servers that can be deployed via underlying virtualization or containerization technologies.

2. **Redirect end-user content requests** to local caches via SDN-based approaches or traditional DNS routing.

3. **Resize the caching service** based on demand, user target, available resources, etc.

Tag	Basic Expression
<SERVICE>	Caching
<RESOURCES>	contents to be cached
<CONJUNCTION>	that can handle, that can meet, etc.
<TARGET>	<WORKLOAD>
<WORKLOAD>	<NUMBER> <UNIT> or <ADJECTIVE> <UNIT> or <ADJECTIVE> "workload"
<NUMBER>	numeric values that can represent the workload
<UNIT>	GB/min, requests/sec, etc.
<ADJECTIVE>	max, min, dynamic, high, medium, etc.
<CONDITIONS>	new caching request, max threshold exceeded, ...
<ACTIONS>	allocate cache servers(), scale out(), ...
<CONSTRAINTS>	with max storage, with least latency, ...
<PRIORITY>	optional indicator of policy priority
<OPERATOR>	Policy will be executed in parallel / sequential way

Table 3.1: Basic expression syntax.

These operations could form up the intent's policies as illustrated in Fig. 3.3. Here, the *intent user* is the CP and the *intent developer* is the CCDN Operator. When an intent developer wants to create a new intent, he has to decide how users can express their high-level targets using the declarative intent expression. So, he has to determine the possible targets that users would want that specific intent service to achieve. Moreover, the resources that this service will use, have to be specified, and the equivalent conjunction between the resources and the target. For example, a **caching** intent would be important for CPs. Therefore, the intent developer can provide several targets that can be achieved via the equivalent policies and implementation, based on his knowledge of the CDN and the user requirements, which could be assisted

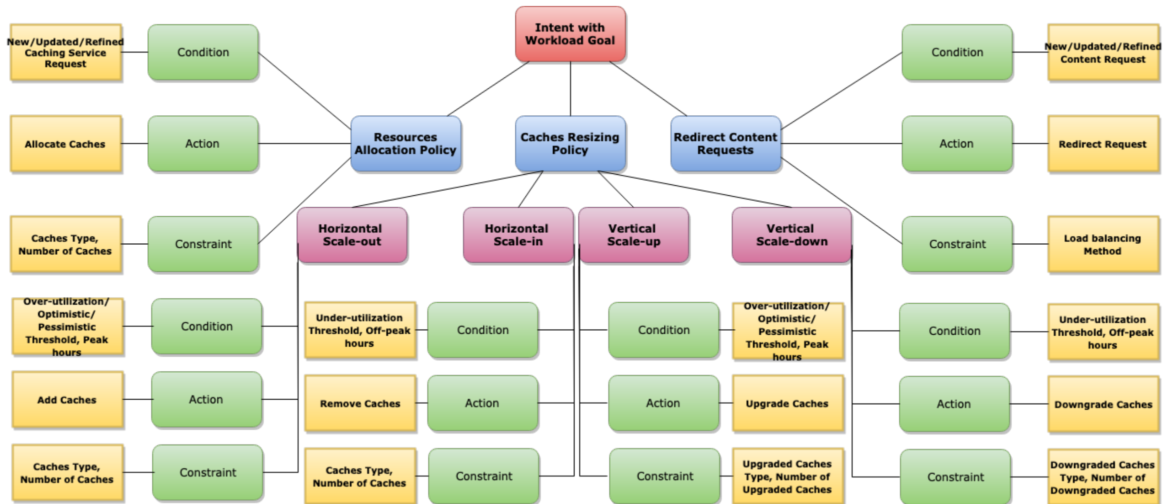

Figure 3.3: Abstract Policies Language Snapshot.

by some operational and technical requirements calculator modules. We discuss a target that a CP would possibly want to achieve at different levels via the caching intent which is `<WORKLOAD>`. The intent developer (i.e., CCDN Operator) has to determine how this could be expressed. For example, the CP can express his required workload that should be met by the caching service intent either numerically such as *"I want caching for content X to handle 20 GB/minute"* or describe it with an adjective such as *"I want caching for content X with the minimum number of requests/region"*. Accordingly, the intent developer must determine the equivalent mapping for each tag in the service-oriented intent expression as shown in Table 3.1. The CCDN Operator determines how to decompose the declarative intent expressed by the CP to a set of abstract policies that jointly achieve the requested operational goal. This is based on two perspectives, The first is *system-wise*, which requires a good understanding of the system: its microservice functionalities, resources availability, etc. The second is *service consumer-wise* which is based on understanding his requirements and what type of microservices and technical considerations are needed to achieve them. For example, the CP's declarative caching intent with a workload target (i.e. service consumer's requirement) can be decomposed into two policies, each maps to a microservice (i.e. allocate cache servers and resize the cache service). This is shown below:

```
If a new caching request is received:
then allocate cache servers with the selected cache server size.
<sequential>
{   If maximum threshold is exceeded in all current caches:
        then add more caches with number of caches to add.
    <parallel>
    If caches are underutilized:
        then remove some caches with number of caches to remove.
}
```

In abstract policies, conditions, actions, and constraints for each policy should be expressed as in Table 3.1, as a set of possible terms or expressions that should be evaluated (e.g. new caching service request). Then after translating them to their corresponding microservice API calls, they get injected into the Service Orchestrator for their execution. The abstract and technical policies can be expressed in JSON format as in Listings 3.1–3.2.

3.3 Architecture and Design

In this section, we discuss how the aforementioned aims will be met with an overall design. This design will also become the basis for our implementation, discussed later in Chapter 4.

```
{  "Resources-Allocation":{
          "Conditions": "NewCachingServiceRequest",
          "Actions": "AllocateCacheServers",
          "Constraints": "CacheServerSize"},
   "Cache-Service-Resizing":{
          "Scale-up":{
                "Conditions": "MaxThresholdExceeded",
                "Actions": "AddMoreCaches",
                "Constraints": "NumberOfCachesToAdd"},
          "Scale-down":{
                "Conditions": "UnderutilizedThreshold",
                "Actions": "RemoveSomeCaches",
                "Constraints": "Time"}
  }
}
```

Listing 3.1: Abstract Policies.

```
{  "Resources-Allocation": {
          "Conditions": "NewCachingServiceRequest",
          "Actions": "SpinUpCacheVMs",
          "Constraints": "CacheSize=medium"},
   "Cache-Service-Resizing":{
          "Scale-up":{
                "Conditions": "CPUUtilization>80%",
            "Actions": "SpinUpCacheVMs",
            "Constraints": "NumberOfVMs=1"        },
        "Scale-down":{
                "Conditions": "CPUUtilization<20%",
            "Actions": "SpinDownCacheVMs",
            "Constraints": "02:42"}
     }
}
```

Listing 3.2: Technical Policies.

The design of our proposed Intent-Based CCDN is based on the following principles:

- **Different Intent Representation Levels:**

  in a CCDN scenario, there are two players: CP (intent consumer) and CCDN operator (intent developer). Intent representation differs for each player as follows:

  - *CP (Intent Consumer):* Intents are represented in a *declarative* way that expresses the CP target at a higher-level
  - *CCDN Operator (Intent Developer):* The intents creation, development, and management steps are maintained by the CCDN operator who regulates the behavioral policies of the intents. Thus, a *prescriptive* expression is needed to represent these policies. However, to allow collaboration between different stakeholders (i.e., different CCDN operators) these prescriptive policy expressions have to be abstract which would be translated differently according to the underlying technology and infrastructure.

- **Technology-Independence:** It is hard to keep up with technology advancements when the CCDN functionalities are tightly coupled to the underlying technologies and implementations. Therefore, it is important to separate the CCDN behavioral policies from their actual technical realization. This enables CCDN operators to think at a higher level and articulate abstract behavioral policies without worrying about the implementation complexities. Moreover, this provides a more flexible, cost-effective, and portable solution by allowing the same policy to be applied across different underlying technologies without the need to define multiple variants. Different policy agents can be plugged into the different underlying systems to translate abstract policies into a technology-specific representation. This also enables comparing and alternating between different realizations for the same abstract policies, which facilitates the collaboration between different stakeholders in the policies definition.

- **Global System and Intent Status Awareness:** The Intent-Based framework needs to control and adjust the CCDN configurations to meet the CP intent goals based on the system and intent state. This could be achieved by means of telemetry and machine learning, as well as intelligent decision-making. This awareness has to include the *pre-*, and *post- CCDN deployment phases* and throughout the overall lifecycle of the intent.

- **Closed-loop Intent Refinement:** According to the feedback of the global system and intent status, the gap between the current CCDN configuration and the intent's desired target CCDN state is inferred. The closed-loop (full-lifecycle management)

CCDN adjustment and intent refinement are continuously performed to enhance the learning mode and adjust the CCDN configuration and finally satisfy the user intent.

- **Extensibility:** Is important to facilitate extending the current intents space by enabling the creation of new intents, and the composition of existing policies to create new more complex intents. This could be achieved by leveraging the current technology advancements in Microservices Architecture's modularity and functionality isolation where the composition of multiple microservice alternatives could jointly create different higher-level overall CCDN deployments that could achieve the intent goal. For example, if an intent goal is to achieve high performance, then this could be translated to the corresponding microservices that help with increasing performance (i.e., cache servers with faster processing, load balancers, CDN auto-scaler, etc.). This translation would be extended when new microservices (or upgraded old ones) that help with achieving better performance can be included. So, the modularity and continuously evolving nature of microservices could highly facilitate the intent translation process and the overall intent space extension. Therefore, it is important to evaluate the overhead of the intent translation time while extending the intent problem space that consists of different microservice alternatives and their evaluation criteria.

Our proposed Intent-Based CCDN framework is depicted in Figure 3.4, the components of which consist of three layers as discussed below. The blue arrows with green numbers denote the Pre-deployment phase (intent translation) and the purple arrows with magenta letters denote the post-deployment phase (intent refinement).

3.3.1 Translation Layer

This layer is the enabler for the communication between CPs and CCDN operators with the CCDN ecosystem, where the actual intent calculation and refinement processes take place. It consists of the following components:

- **Content Provider Intent API:** An interface allowing CPs to express (i.e., add, update, remove) declarative intents. A CP can express intents from available provided intent expressions (e.g., Caching with Low-Cost) (Step ①) as shown in Figures 3.5, 3.6 and 3.7. In these examples, a CP could make his decision regarding the suggested intent translation based on several criteria (i.e., cache server size, zone, and performance improvement level). The CP could accept, reject, or update the suggested CCDN deployment that corresponds to his intent input.

- **Multiple Criteria Decision Making Module for CCDN Deployment:**

The high-level intent then needs to be translated, so the MCDM module calculates the equivalent AHP graph (described in detail in Section 3.4.1) corresponding to the intent input (Step (2)). The AHP output would be a score ranking for each microservice that could be used in a CCDN deployment, which is done with correspondence to the evaluation criteria defined in the AHP graph, where their degree of contribution to the CCDN solution depends on the CP intent (Step (3)). For instance, if there are 2 main evaluation criteria defined in the AHP graph: *scalability* and *cost*. The cost in our context refers to what the CP has to pay for the CCDN service, so when they are interested in cost reduction, then the lower the cost value, the better it is, and vice versa. When the CP intent is to get the *best available performance*, then this intent target requires assigning a much greater weight of importance to the *scalability* criterion as opposed to the *cost* criterion (e.g., *scalability weight = 0.9* and *cost weight = 0.1*). Accordingly, the overall score calculation and priority for each microservice would differ.

- **CCDN Deployments Enumeration and Clustering Module:**

  After the AHP scores have been calculated for each microservice, the CCDN deployments enumeration and clustering take place. This process enumerates all possible CCDN deployments by selecting a microservice instance from each different

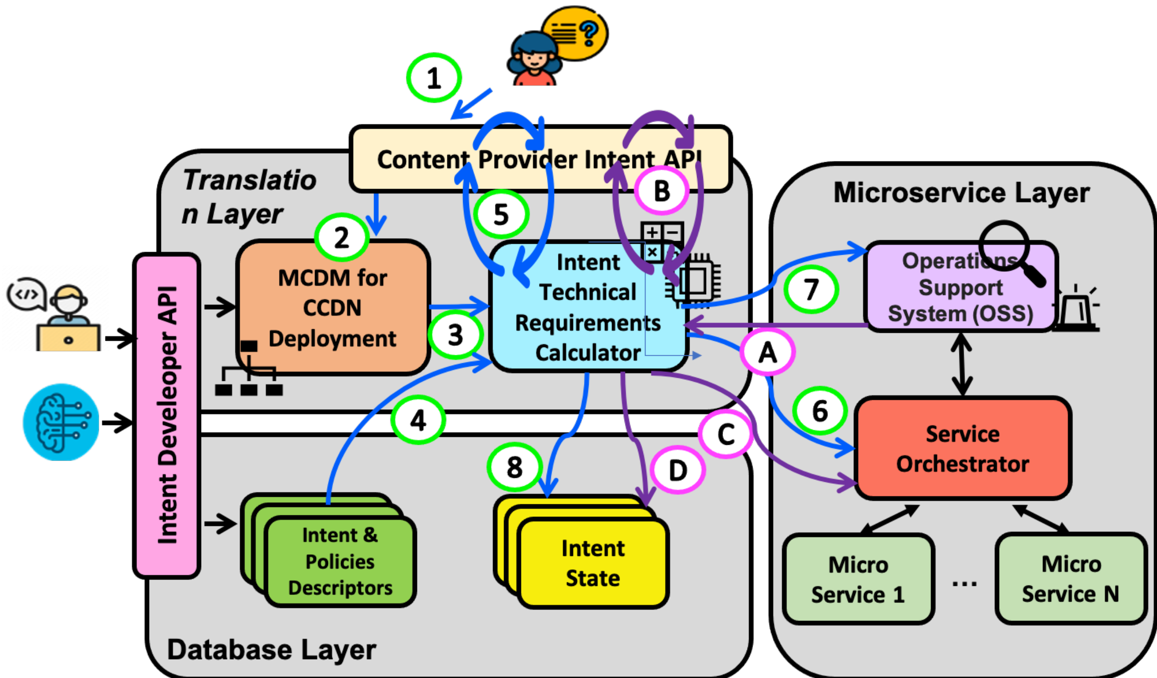

Figure 3.4: Intent-Based Framework.

type that could collectively constitute a CCDN deployment. For example, if we have 2 different types of microservices categorization: *cache size* and *cache placement zone*. Let us assume that each category has 3 different instances (*cache size*: big, medium and small) and (*cache placement zone*: local, nearby and further), then we could enumerate 9 (3×3) different CCDN deployments out of these microservice instances. Since each microservice in the enumerated CCDN deployments has a score, the overall combined score of the deployment can be calculated by summing

```
Please enter your request:
 {'service': 'caching', 'resources' : 'London', 'conjunction' : 'with',
'target' : 'low cost'}


Your suggested CDN deployment is:
Cache Server Size:  Small
Zone: London
Performance improvement Level: Medium


Do you wish to continue? Yes / No / Update
 Yes

Your CDN has been deployed and it is  ready!
```

Figure 3.5: CP's Approval on Suggested CCDN Deployment.

```
Please enter your request:
 {'service': 'caching', 'resources' : 'London', 'conjunction' : 'with',
'target' : 'low cost'}


Your suggested CDN deployment is:
Cache Server Size:  Small
Zone: London
Performance improvement Level: Medium


Do you wish to continue? Yes / No / Update
 No

Would you like to update your request? Yes / No
 No

Your CDN request has been cancelled!
```

Figure 3.6: CP's Rejection of Suggested CCDN Deployment.

up the scores of all microservices included in that deployment.

Once the total overall ranking for each possible CCDN deployment is achieved, the clustering process takes place which groups CCDN deployments with close or similar scores into a cluster.

Eventually, the generated output (Step ④) contains 3 levels of clusters: *high, medium, and low*. The CCDN deployments that belong to the high-scored cluster, indicate that they have a high capability in achieving the intent goal, whereas the medium ones have an intermediate capability, and the low-scored deployments have possibly the lowest capability to meet the intent goal.

The CCDN deployment selection is based on its priority score and resource availability, so the solution selected would be the highest-scoring available CCDN.

The benefit of having these CCDN clusters is to offer more flexibility and availability where multiple CCDN alternatives could be selected from the same cluster (level of capability). Whereas, when all resources in the high-scored CCDN deployments are not available, then the medium and low clusters could be beneficial if the CP would compromise his intent goal to some degree in return for running the service rather than not starting it at all.

```
Please enter your request:
 {'service': 'caching', 'resources' : 'London', 'conjunction' : 'with',
'target' : 'low cost'}


Your suggested CDN deployment is:
Cache Server Size:  Small
Zone: London
Performance improvement Level: Medium


Do you wish to continue? Yes / No / Update
 Update

Please enter your updated request:
 {'performanceImprovement': 'Low'}


Your suggested CDN deployment is:
Cache Server Size:  Small
Zone: London
Performance improvement Level: Low
```

Figure 3.7: CP's Update on Suggested CCDN Deployment.

- **Intent Technical Requirements Calculator (Intent Handler):**

  Starting with the CCDN **pre-deployment** phase, once a suggested CCDN deployment has been selected from the high-scored cluster depending on the resource's availability. Then it has yet to be technically defined by this module which calculates the corresponding operational parameters for its technical requirements (e.g., Max and Min No. Caches, Cache container image, etc.). This is done after obtaining the corresponding necessary information from the **Intent and Policies Descriptors** (Step (5)) located in the *Database Layer* that indicates the abstract regulatory behavior of the CP intent, which include the conditions, actions, constraints, attributes, etc. Then, the availability of this selected CCDN has to be checked by the **Operations Support System (OSS)** module to report back its availability to the **Intent Technical Requirements Calculator** (Step (6)). Following that, it reports back the available suggested CCDN to the CP for their input (Step (7)), who responds to the initial suggested and available CCDN that would help to achieve his/her intent (i.e., approve, decline). Once approved, the corresponding microservice API calls are formulated and sent to the **Service Orchestrator** (i.e., Kubernetes Controller) for service deployment (Step (8)). Moreover, the intent's details and status get stored in the **Intent State** in the *Database Layer* (Step (9)).

  Moreover, in the **post-deployment** phase, the intent refinement takes place, which aims at improving the CCDN and adapting it based on some captured and analyzed service and demand behavior. This module determines and sends the intent refinement metrics (e.g., peak hours, traffic bursts, etc.) to be monitored by the **OSS** module, and gets alerted when any metric has been hit to start the intent refinement process (Step (A)). Then the **Intent Technical Requirements Calculator** identifies and calculates the suggested CCDN refinement and reports this back to the CP and waits for their response to the suggested refinement (i.e., approve/decline/update) in (Step (B)). If the suggested refined CCDN was approved or updated (e.g., updated peak hours by the CP) as shown in Figure 3.8, then this module formulates the corresponding microservice API calls and sends them to the **Service Orchestrator** in (Step (C)). Finally, it updates the **Intent State** in the *Database Layer* in (Step (D)).

- **Intent Developer API:** Allows intent developers (CCDN Operators) to create intents and corresponding policies based on their expertise, or a Knowledge Base could be used to derive new intents and policies from previous experience. In specific, an intent developer has to define the MCDM approach (AHP in our solution) along with its components (i.e., evaluation criteria, microservice alternatives, and their comparative weights). We will elaborate on this in Section 3.4.1. The **Intent and Policies Descriptors** in the *Database Layer*, have to be

specified by the intent developer in order to state the translation behavior for the **Intent Technical Requirements Calculator module**.

```
Your suggested performance improvement is:
Improvement action: Cache size upgrade during peak hours
Upgraded cache  size: Medium
Peak hours start: 17:00
Peak hours end: 20:00

Do you wish to proceed with this performance improvement? Yes / No / Update
 Update



Please enter your performance improvement:
 {'peakStart': '18:30', 'peakEnd': '21:00'}



Your suggested performance improvement is:
Improvement action: Cache size upgrade during peak hours
Upgraded cache  size: Medium
Peak hours start: 18:30
Peak hours end: 21:00
```

Figure 3.8: CP's Update on Suggested Performance Improvement.

3.3.2 Microservice Layer

This layer denotes the Microservices Architecture, consisting of a set of microservices along with the following components:

- **Operations Support System (OSS) Module:**

  We assume that this module has direct access to the resource repository. Therefore, in the CCDN **pre-deployment** phase, the **OSS** module is responsible for checking the availability of the selected CCDN deployment and reporting that back to the **Intent Technical Requirements Calculator** (Step ⑥). As for the **post-deployment** phase, the **OSS** module receives the intent refinement metrics (e.g., peak hours, traffic bursts, etc.) to be monitored and continuously watches out for the refinement metrics if any of them has hit the defined limit by the **Intent Technical Requirements Calculator**, then alerts it (Step Ⓐ).

- **Service Orchestrator:** Coordinates the service behavior (e.g., Kubernetes Orchestrator) by regulating the interactions and configurations of the corresponding microservices while considering microservices API calls that correspond to the translated technical policies as guidelines (from Step (8) and Step (C)).

3.3.3 Database Layer

This layer consists of the internal Knowledge Base for the intent translation, refinement, and status storage. It contains the following components:

- **Intent and Policies Descriptors:** Stores the intent and policies descriptors that regulate the CCDN behavior, and breaks them down into their corresponding *conditions*, *actions*, *constraints*, *attributes*, etc. This contains the basis of the intent translation and refinement according to the regulated behavior. More details will be discussed in the following Section.

- **Intent State:** A database that keeps track of the intent state and details throughout the intent lifetime.

3.4 Multiple Criteria Decision Making for CCDN Deployment

This decomposition process of an intent aims to find the best CCDN deployment options for a requested CP intent target.

3.4.1 Analytical Hierarchy Process (AHP)

A similar Intent-Based work to ours (Scheid et al., 2017) used Softgoal Interdependency Graph (SIG) (Yrjönen et al., 2009) as their MCDM approach. However, we choose AHP instead which is one of the most widely used techniques for solving problems related to MCDM when the number of choices is known beforehand (Thomas L Saaty, 1994a). This choice has been influenced by the comparison between AHP and SIG (Kassab, 2013) which is summarized and shown in Table 3.2. AHP was developed in the 1970s by Thomas L. Saaty (Thomas L Saaty et al., 1979) and has been extensively studied and refined ever since (Thomas L Saaty et al., 2012), (Whitaker, 1987), (Thomas L Saaty, 1985),(Thomas L. Saaty, 1994b), (Thomas L Saaty et al., 1991), and (Thomas L Saaty, 1990). AHP has been used and refined to a wide variety of decision areas (e.g., resource allocation, product selection, transport planning, etc.).

	AHP	SIG (NFR Framework)
MCDM Category	Mathematical based	Model (graphical) based
Output	Ranking all solutions with regards to certain criteria (i.e., maximization, minimization, or optimal)	Finding adequate solutions that attempt to meet criteria but without necessarily producing an optimal solution
Handled Comparison Elements	Quantitative and qualitative	Mostly qualitative
Comparison Scales	More scales	Limited scales
Problem Size	Large amounts of data and variables	Not suited for large amounts of data as expressing and viewing it in a model can be cumbersome
Comparison Accuracy	More accurate due to pair-wise comparison	Could be error-prone due to the direct score assignment with absolute judgment
Purpose	Structured decision making with better analysis, repeatability, and consistency	Structured decision support material
Application	Bigger problems with the need for high accuracy and consistency	Small problems with the focus on the graphical modeling for easier presentation and information sharing

Table 3.2: Comparison Between AHP and NFR Frameworks.

It is a structured decision-making technique that can represent the human decision-making process and help to achieve better judgments based on mathematics and psychology. It provides several advantages as follows:

- **Modeling complex problems in the form of a hierarchy:** AHP facilitates prioritizing alternatives when multiple conflicting criteria and sub-criteria must be considered. This technique allows decision-makers and stakeholders to structure and model complex problems in the form of a hierarchy, or a set of integrated levels. It shows the relationships of the main goal, its satisfaction criteria and sub-criteria, and the alternative solutions. It was evaluated to be the most promising method for software requirements prioritization (Karlsson et al., 1998).
- **Using pair-wise comparison for more accurate judgments:** Psycholo-

gists argue that pair-wise comparison is easier and more accurate compared to simultaneously doing this on all the alternatives. (Ishizaka et al., 2011). This also helps different stakeholders and decision-makers to compare and check their possibly different pair-wise comparisons. Therefore, these comparisons are adopted in the AHP decision-making process. When the AHP consists of a large number of elements, decision-makers should attempt to arrange these elements in clusters so that they would not differ in extreme ways. The pair-wise comparison argument given by Saaty (Thomas L Saaty et al., 2012) against directly assigning scores with absolute judgment is that the latter is subjective, thus it could be error-prone. Using pair-wise comparisons in AHP for decision-making is seen as a means to aid these shortcomings. So instead of deciding *"how good is solution A at fulfilling criterion X?"*, the decision becomes *"how much better or worse is solution A at fulfilling criterion X compared to solution B?"*.

Intensity of Importance	**Definition**	**Explanation**
1	Same importance	Two elements contribute equally to the objective
2	Weak or slight	
3	Moderate importance of one over another	Experience and judgment slightly favor one element over another
4	Moderate plus	
5	Essential or high importance	Experience and judgment highly favor one over another
6	Strong plus	
7	Very strong importance	One element is very strongly favored over another, its dominance is demonstrated in practice
8	Very, very strong	
9	Extreme importance	The evidence favoring one element over another is of the highest possible order of affirmation
Reciprocals	Whenever element i compared to j is assigned one of the above numbers, the element j compared to i is assigned its reciprocal	A reasonable assumption

Table 3.3: The AHP Saaty's Original Scale.

- **The ability to use different judgment scales for more accuracy:** Pair-wise judgments could be quantified using many scales. Saaty proposes the scale [1...9] (i.e., linear scale) to rate the relative importance of one criterion (or solution alternative) over another, listed in Table 3.3, and construct a matrix of the pair-wise comparison ratings. Even though several other numerical scales in AHP have been proposed (Franek et al., 2014), the original linear scale proposed by Saaty has been used by far the most often in different applications. However, the authors in (Franek et al., 2014), have analyzed the impact of different judgment scales and found that there is no profound impact on the criteria ranking even though they could have a profound impact on the numerical criteria priority. The original 9-unit scale is based on psychological theories and experiments that consider the use of 9-unit scales as a reasonable set that allows humans to perform discrimination between preferences for two items (Nydick et al., 1992), where 1 means both are equally important, 3 means one is moderately more important than the other, and 9 means that one requirement is extremely more important than another. Conversely, if a component is less important than the other, then the **inverse** preference is rated in a scale 1, 1/2, 1/3... 1/9. For example, if decision-makers believe that "Security" is moderately less important than "Performance", then this judgment is expressed as 1/3. These pair-wise comparisons are required for all the criteria, sub-criteria, and solution alternatives for each criterion to assess their fulfillment of it. These judgments are usually provided by the stakeholders and decision-makers.

- **The ability to evaluate quantitative and qualitative criteria and alternatives on the same preference scale:** For performing a pair-wise comparison, AHP uses a ratio scale, which opposed to an interval scale, does not require units in the comparison. In this case, the comparison judgment is a relative value or a quotient x/y of two quantities x and y that both have the same units (cost, performance, etc). Therefore, one of the most prominent advantages of the AHP technique is its ability to evaluate quantitative as well as qualitative criteria and alternatives on the same preference scale.

- **Measuring decisions' consistency degree:** Another important advantage of using the AHP decision-making technique is that it can measure the degree to which decision-makers' judgments are consistent. In the real world, some inconsistency is acceptable and even natural. For example, in a marketing contest, if product A is usually more popular than product B, and if product B is more popular than product C, this does not imply that product A is usually more popular than product C. A slight inconsistency may result because of the way the products are evaluated overall. Hence, it is important to make sure that inconsistency remains within a reasonable limit (i.e., $< 10\%$), and if exceeded then some revision and modifications of judgments may be required. AHP technique provides a method to compute

the consistency of the pair-wise comparisons as discussed in (Thomas L Saaty et al., 2012), (Whitaker, 1987), (Thomas L Saaty, 1985),(Thomas L. Saaty, 1994b), (Thomas L Saaty et al., 1991), and (Thomas L Saaty, 1990).

- **Prioritizing all solution alternatives based on the degree of meeting criteria:** The AHP decision-making technique offers a methodology to rank solution alternatives based on the importance of the criteria, and the extent to which they are met by each alternative. Hence, AHP is a **mathematical optimization** that outputs the highest-scoring solution along with all other solutions' scores that could be then ordered based on their priority score.

3.4.2 Analytical Hierarchy Process Computation

In order to decide and generate priorities in an organized way for the AHP graph as shown in Figure 3.9, we need the following steps:

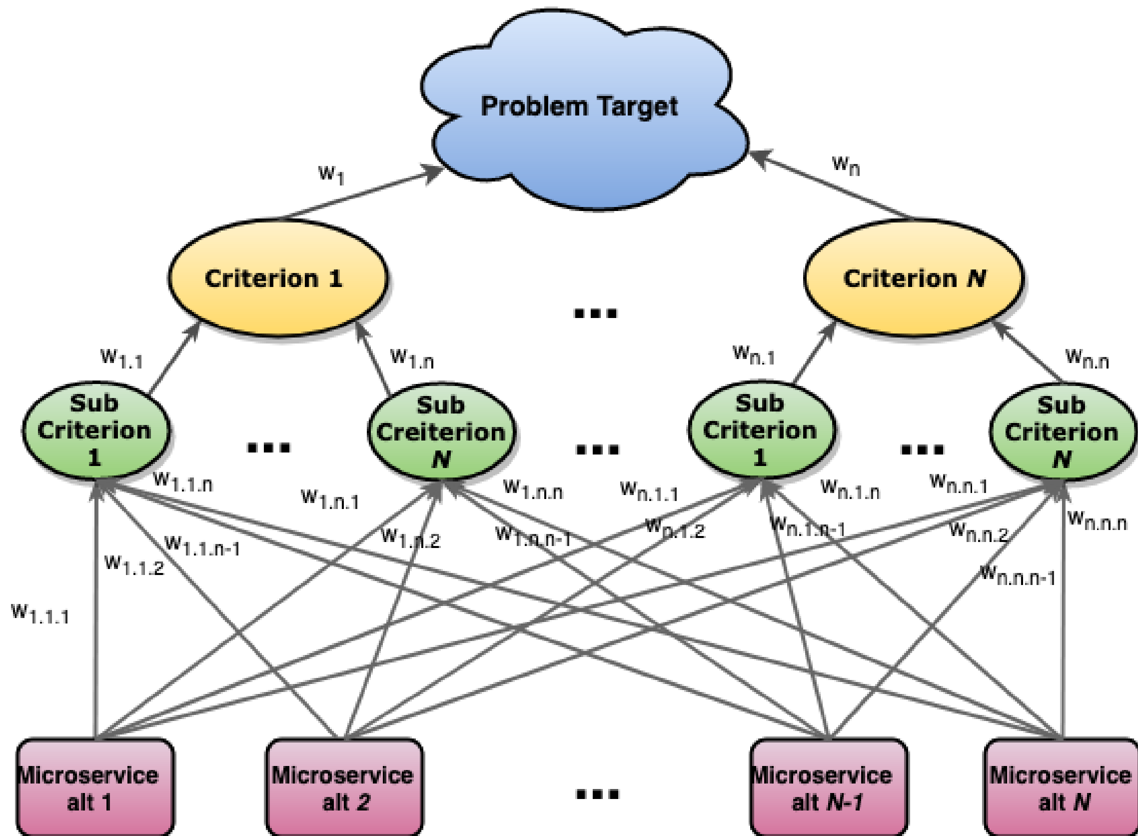

Figure 3.9: Generic AHP Graph.

1. **The AHP Decision Hierarchy Construction:** The first step in AHP is to define the problem and construct the decision hierarchy by doing the following:

    - state the problem objective (i.e., intent target)
    - define the criteria and sub-criteria (i.e., evaluation criteria)
    - choose the solution alternatives (i.e., microservice alternatives)

    The top of the hierarchy is the problem objective, whereas the criteria and sub-criteria are placed through the intermediate levels. Finally, the solution alternatives are at the lowest level. In this decision hierarchy, the objective is to prioritize and rank the alternative solutions aiming at satisfying the evaluation criteria while considering their interdependencies. The analysis of the problem hierarchy is based on the impact of a certain level on the elements in the level immediately below with respect to it.

    This process starts by determining how the criteria are relatively important in meeting the problem objective. Next, it measures the extent to which the solution alternatives achieve each of the criteria.

    Finally, the results of these two analyses are synthesized (multiplied) to compute the relative importance of the solution alternatives with regard to the problem objective. This process is done when we have only three hierarchy levels, otherwise, this process is done level by level. Most of the cases require dividing the criteria into sub-criteria which adds more levels.

2. **Make Pair-wise Judgments at each Level of The AHP Hierarchy** of the components on that certain level with respect to their impact on the overall goal. Each component is compared to all others and assigned a number to show its relative importance. These comparisons estimate the ratio of the relative importance of any two criteria in terms of the overall problem goal and also in terms of the criteria.

    The pair-wise comparison information for each component of the problem hierarchy is represented by a positive reciprocal $(n \times n)$ comparison matrix. If there are n elements to be compared for a given matrix, then a total of $n(n-1)/2$ judgments are needed. This apparent saving in the number of judgments is due to two reasons: *first*, any element is equally preferred to itself, so, 1's are placed along the diagonal of the matrix. *Second*, the corresponding positions below the diagonal are the reciprocals of the judgment values already entered in the matrix (i.e., above the diagonal).

3. **Aggregate Pair-wise Scores:** The final step is to aggregate these pair-wise scores to arrive at an overall ranking of the solution alternatives. Multiplied by

the established weights of the criteria. Then the overall scoring of all alternatives can then be determined and ranked based on their priority scores.

AHP calculation: We walk through the element's priority calculations process in more detail. In the pair-wise comparison matrix, Saaty (Thomas L Saaty et al., 2012), (Whitaker, 1987), (Thomas L Saaty, 1985),(Thomas L. Saaty, 1994b), (Thomas L Saaty et al., 1991), and (Thomas L Saaty, 1990) proposed using the *principal eigenvector method*, where he justifies this approach for slightly inconsistent matrices when inconsistency remains within a reasonable limit (i.e., $< 10\%$). These priorities could be obtained in either an approximate or exact form. The former is calculated by adding each row of the matrix and dividing by their total, whereas the latter is an iterative process that could be explained as follows:

1. raise the pairwise matrix to powers (i.e., power method (Ishizaka et al., 2006)). The matrix is squared (i.e., multiply it by itself)

2. The row sums are then calculated and normalized by dividing each by the total sum of all the rows. This is the first approximation of the eigenvector (priority vector)

3. Using the resulting matrix. Steps 1 and 2 are repeated

4. Step 3 is repeated until the difference between these sums in two consecutive priority eigenvector calculations is smaller than the stop criterion

These priority calculations are done for the alternatives with respect to the criteria or subcriteria in terms of which they need to be evaluated. Likewise, the criteria priorities are calculated in terms of a higher goal.

3.4.3 The AHP Graph for CP's Targeted Workload Intent

We propose a CP intent that aims to achieve a certain CCDN workload level in different possible ways. This could possibly be achieved with different CCDN deployment combinations based on the available microservice alternatives. Therefore, a CCDN operator is responsible for building the AHP hierarchy, where he has to define the AHP goal (intent goal) in the first level of the hierarchy, the main evaluation criteria in the second level, the sub-criteria in the third (or more) level, and the microservice alternatives in the last level of the hierarchy as discussed in Section 3.4.2. In the context of a `<WORKLOAD>` CP's intent target for the CCDN, the corresponding AHP graph is shown in Figure 3.10. We choose 2 main evaluation criteria defined in the AHP graph: *scalability* and *cost*. When the CP intent is to get the best available performance, then this would require assigning a much greater weight of importance to the scalability criterion as opposed to the cost criterion (e.g., *scalability weight*

= *0.9* and *cost weight* = *0.1*). Conversely, when a CP's intent is to get a Low-Cost CCDN then the corresponding *Scalability and Cost* weights are opposite to the best performance CCDN intent. Accordingly, the overall score calculation for each microservice alternative would differ.

As for the subcriteria level, each criterion has been decomposed into 3 different subcriteria. That is because we have focussed on 3 different microservice type

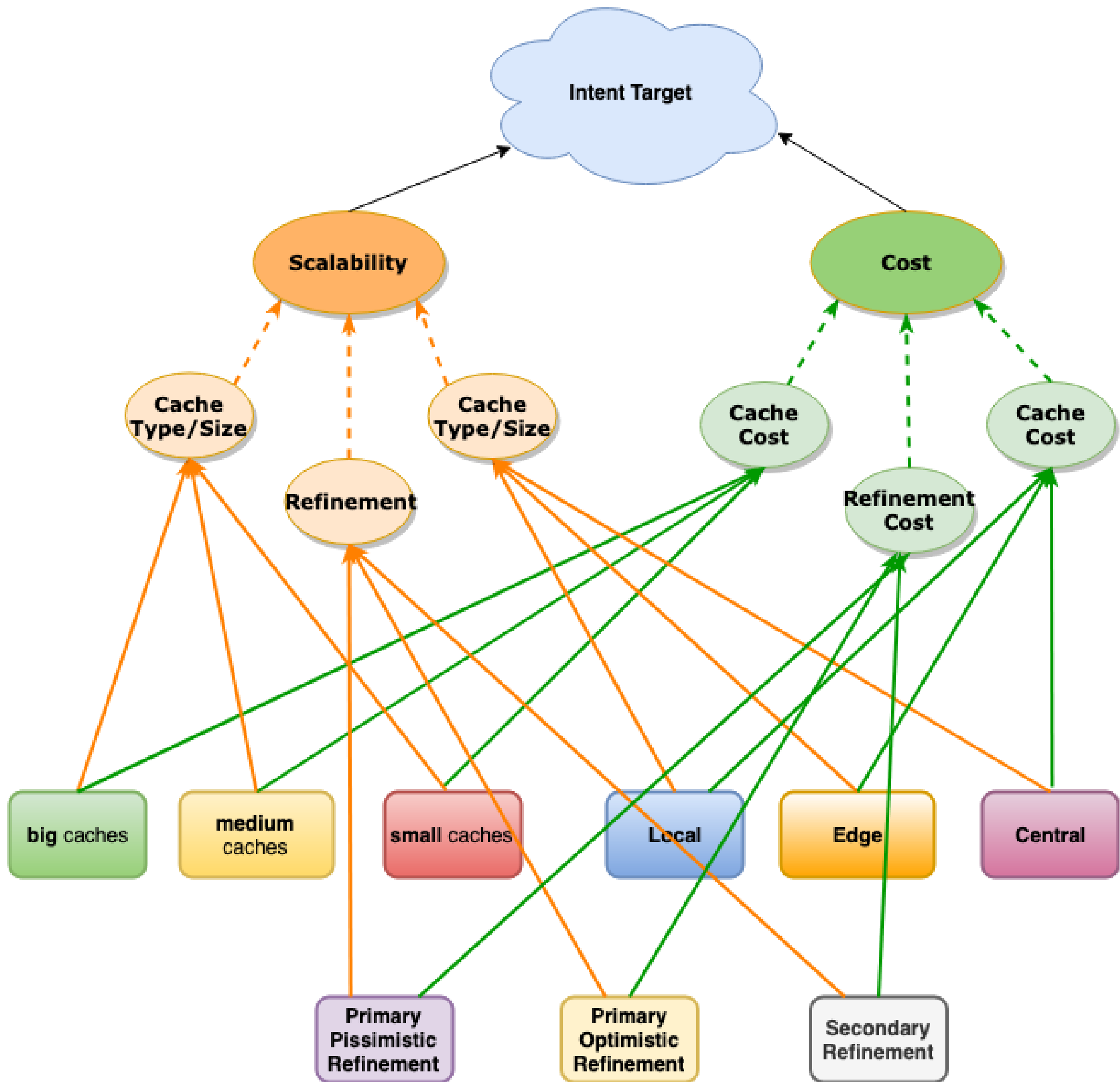

Figure 3.10: CP Workload Intent Target's AHP Graph (weights on arrows are omitted for figure clarity).

categories: *Cache Size*, *Cache Placement*, and *Refinement Technique*. We assume having the following alternatives for each microservice category:

- **Cache Size:** 3 different cache sizes (small, medium, and big) where each size has double the resources of the previous smaller size, and correspondingly, double the cost. For example, a Medium Cache has twice the resources, capabilities, and cost of a small cache. The same applies to Big and Medium comparisons. So in terms of Scalability, the bigger the cache size the better, but the smaller the cache the less costly it is.

- **Cache Placement:** Caches could be placed in different zones, we assume having 3 zones, namely, local, nearby (i.e., edge), and further (i.e., central), and accordingly, caches in each zone differ in the startup delay due to the location's impact on latency (Charyyev et al., 2020) and cost (Google, 2023g). The caches placed in closer zones have less startup delay which improves scalability, whereas the cache cost per location could vary depending on several factors (e.g., CAPEX and OPEX of data centers in each location), in our case, we assume that caches in closer locations are more costly. Thus, closer cache zones improve scalability, but at a higher cost.

- **CCDN Refinement Technique:** Aims at improving the CCDN service in the post-deployment phase after running the service which depends on the CP Intetn target and corresponding preferred service refinements. We assume having 3 levels of refinements (Pessimistic Primary, Optimistic Primary, and Secondary). The Pessimistic Primary has the highest potential degree of refinement and service improvement but at the highest expense due to relying on more inclusive refinement triggers compared to the Optimistic Primary. The Secondary refinement has the lowest ability to improve the service but at the lowest cost.

To demonstrate the AHP computation with respect to the **Low-Cost CP intent**, we start with the evaluation criteria level where we compare *Scalability* and *Cost* weights in a pair-wise comparison matrix as shown in Table 3.4. Each pair-wise comparison between items a and b is shown in Equation 3.1, where a is the item in the matrix **row** and b is the item in the **column**, and 1s are placed along the matrix diagonal, it is easy to fill up the lower triangular part of the matrix by the inverse of the related pair in the upper half of the matrix. So, the pair-wise comparison between b and a follows Equation 3.2. The **Global Priority** represents the *Priority vector* and is calculated as described in Section 3.4.2.

$$Score(a, b) = \frac{weight(a)}{weight(b)} \tag{3.1}$$

	Scalability	Cost	Global Priority
Scalability	1	1/9	0.1
Cost	9	1	0.9

Table 3.4: AHP Criteria

	Cache Size	Cache Placement	Refinement	Local Priority	Global Priority
Cache Size	1	2	4	0.57	0.057
Cache Placement	1/2	1	2	0.29	0.029
Refinement	1/4	1/2	1	0.14	0.014

Table 3.5: AHP Scalability Sub-Criteria

$$Score(b,a) = \frac{1}{\frac{weight(a)}{weight(b)}} = \frac{weight(b)}{weight(a)} \tag{3.2}$$

Due to having 3 microservice categories, each has its own contribution to the overall Scalability. Therefore, we break it down into 3 subcriteria (*Cache Size*, *Cache Placement Startup Delay* and *Refinement*) with respect to the different microservice types. Their corresponding pair-wise comparison scores are calculated in the same way explained with the criteria comparison, and it is also the same as the microservice alternatives comparison in the last level of the AHP. The corresponding comparison matrix is shown in Table 3.5 where all weights should be defined and tuned by the CCDN operator. The local and global scores are calculated as discussed in Section 3.4.2.

The pair-wise comparison scores in the comparison matrices mentioned above have been assigned based on some lightweight benchmarking to identify the degree of impact for a criterion/alternative with respect to the upper-level criteria (e.g., how much important is Cache Size to Scalability compared to Cache Placement?). However, these scores have to be carefully tuned based on the problem requirements and the CCDN operator's expertise.

Samewise, the Cost criterion is broken down into 3 subcriteria (*Cache Size Cost*, *Cache Placement Cost*, and *Refinement Cost*). Their corresponding comparison matrix is shown in Table 3.6.

At the last level of the AHP graph, we compute the pair-wise comparisons between the microservice alternatives within each category with respect to its equivalent Scalability evaluation subcriteria as shown in Tables 3.7, 3.8 and 3.9.

	Cache Size Cost	Cache Placement Cost	Refinement Cost	Local Priority	Global Priority
Cache Size Cost	1	2	4	0.57	0.513
Cache Placement Cost	1/2	1	2	0.29	0.261
Refinement Cost	1/4	1/2	1	0.14	0.126

Table 3.6: AHP Cost Sub-Criteria

	Big Cache	Medium Cache	Small Cache	Local Priority	Global Priority
Big Cache	1	6	9	0.64	0.036
Medium Cache	1/6	1	6	0.31	0.018
Small Cache	1/9	1/6	1	0.05	0.003

Table 3.7: AHP Scalability of Microservices under Cache Size Category

	Local	Nearby	Further	Local Priority	Global Priority
Local	1	4	7	0.64	0.019
Nearby	1/4	1	4	0.28	0.008
Further	1/7	1/4	1	0.07	0.002

Table 3.8: AHP Scalability of Microservices under Cache Placement Startup Delay Category

	Primary Pessimistic	Primary Optimistic	Secondary	Local Priority	Global Priority
Primary Pessimistic	1	4	7	0.64	0.009
Primary Optimistic	1/4	1	4	0.28	0.004
Secondary	1/7	1/4	1	0.07	0.001

Table 3.9: AHP Scalability of Microservices under Refinement Category

	Big Cache	Medium Cache	Small Cache	Local Priority	Global Priority
Big Cache	1	1/2	1/6	0.11	0.056
Medium Cache	2	1	1/3	0.22	0.113
Small Cache	6	3	1	0.67	0.344

Table 3.10: AHP Cost of Microservices under Cache Size Cost Category

	Local	Nearby	Further	Local Priority	Global Priority
Local	1	1/2	1/6	0.11	0.029
Nearby	2	1	1/3	0.22	0.057
Further	6	3	1	0.67	0.175

Table 3.11: AHP Cost of Microservices under Cache Placement Cost Category

Similarly, the microservice alternative comparisons concerning the Cost subcriteria are shown in Table 3.10, 3.11, and 3.12.

Finally, for each microservice alternative, both global scores corresponding to Scalability and Cost are summed up to produce the final score ranking as shown in Table 3.13, 3.14, and 3.15.

3.4.4 The Corresponding CCDN Deployment to the CP's Targeted Workload Intent

After all microservice alternatives got assigned with their total score indicating their level of contribution towards the CP's intent target, the following step is to enumerate

	Primary Pessimistic	Primary Optimistic	Secondary	Local Priority	Global Priority
Primary Pessimistic	1	1/2	1/6	0.11	0.014
Primary Optimistic	2	1	1/3	0.22	0.028
Secondary	6	3	1	0.67	0.084

Table 3.12: AHP Cost of Microservices under Refinements Cost Category

	Big Cache	Medium Cache	Small Cache
Scalability Global Score	0.036	0.018	0.003
Cost Global Score	0.056	0.113	0.344
Total Global Score	0.092	0.131	0.347

Table 3.13: Cache Size Microservice Alternatives Global Score

all possible CCDN deployments such that a deployment consists of a microservice instance from every different microservice category. In this case, a CCDN deployment is constituted of an instance from *Cache Size*, *Cache Placement*, and *Refinement Technique* (e.g., *Medium Cache, Nearby Zone, and Optimistic Refinement*). Since we have 3 alternatives from each microservice category, the total number of possible CCDNs to enumerate is 27 ($3\times3\times3$), where each deployment's score is the summation of all of its constituent microservices. For example, to demonstrate the total score calculation of a CCDN deployment, let us take a random CCDN deployment that has been enumerated according to our defined AHP graph. The CCDN deployment is: (*Medium Cache, Nearby Zone, and Optimistic Refinement*). By obtaining the total score for each microservice instance within the CCDN deployment from Tables 3.13, 3.14, and 3.15, the overall deployment score would be ($0.131 + 0.065 + 0.023$) = 0.219.

Once all CCDN deployment scores get calculated, they get clustered into groups of deployments with similar or close scores. These clusters have 3 levels of score classification: *High, Medium, and Low*. More details about this process will be discussed in Chapter 4. The CCDN deployments that belong to the high-scored cluster, indicate that they have a high capability of achieving the intent goal, whereas the medium-scored ones have an intermediate capability and the low-scored deployments have possibly the lowest capability to meet the intent goal. The benefit of having these CCDN clusters is to offer more flexibility and availability where multiple CCDN alternatives could be selected from the same cluster (level of capability). Whereas, when all resources in the high-scored CCDN deployments are not available, then the medium and low clusters could be beneficial if the CP would compromise his intent goal to some degree in return for running the service rather than not starting it at all. These scores are dependent on the CP's intent target which is translated as its corresponding weights for the evaluation criteria. Therefore, the AHP calculation is reactive to the intent's target.

	Local	Nearby	Further
Scalability Global Score	0.019	0.008	0.002
Cost Global Score	0.029	0.057	0.175
Total Global Score	0.048	0.065	0.177

Table 3.14: Cache Placement Microservice Alternatives Global Score

	Optimistic Ref.	Pessimistic Ref.	Secondary Ref.
Scalability Global Score	0.009	0.004	0.001
Cost Global Score	0.014	0.028	0.084
Total Global Score	0.023	0.032	0.085

Table 3.15: Refinement Microservices Alternatives Global Score

3.5 Different Intent Targets

Based on the presented AHP graph in Figure 3.10 which is related to our CCDN use case, several intent targets could possibly be achieved as shown in Figure 3.11 by adjusting the evaluation criteria weights accordingly, and correspondingly, the whole AHP computation would rank and prioritize the microservices differently. Thus, the whole CCDN scores would differ for each intent target. For instance, when the CP intent is to get a "High Performance" CCDN, then this translates to a

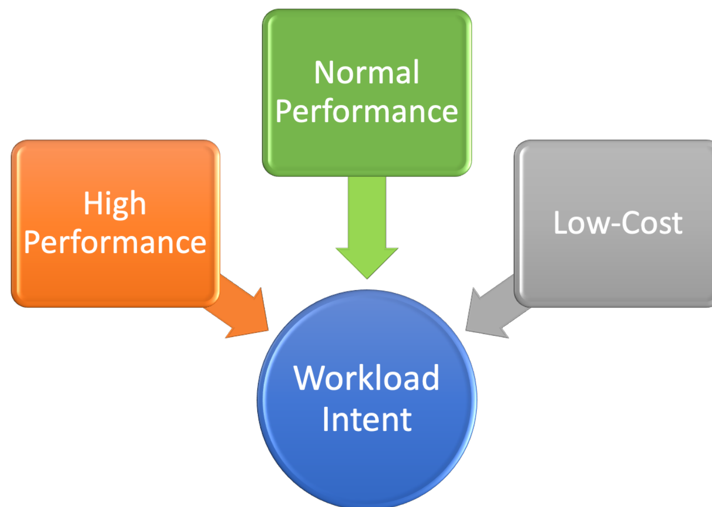

Figure 3.11: Different Possible CP Intent Targets.

considerably higher Scalability weight than the Cost weight. Whereas, if the intent is to get a "Normal Performance" CCDN, then both Criteria get the same weight. Conversely, when the CP is interested in cost reduction, then the "Low-Cost" CCDN intent translates to a considerably higher Cost weight compared to the Scalability weight. However, these weights are up to the CCDN to decide and tune based on the service requirements, system requirements, tradeoffs, etc. In this Ph.D., we focus on investigating several alternatives of Low-Cost intents due to the importance of offering less-costly CCDNs to CPs who may not be as interested in performance optimization as much as cost reduction. These Low-Cost intent alternatives are discussed along with their refinements after running the service, in Chapter 4, Section 4.3.3.

3.6 Intent Refinement

The dynamic nature of the CCDN intent refinement that adopt microservice ecosystems calls for adaptation support to be added into the architecture similar to that provided by autonomic systems (Cheng et al., 2009). It is not always possible or even desirable to have a fully autonomic approach, in fact, in most cases, manual human interventions or directives (i.e., IT personnel) are complemented by autonomic capabilities. Several mechanisms could lead towards autonomicity (Cheng et al., 2009), (Garlan et al., 2004). Their fundamental ideas are generally underlain by **feedback loops** (Brun et al., 2009), which include the activities of (MAPE-K): Monitor (M), Analyze (A), Plan (P) and Execute (E). These actions acquire sufficient knowledge to perform actions that are to be automated. The knowledge (K) that is maintained in a Knowledge Base, is needed to recognize the need for adaptation, and automatically decide and perform the required actions correspondingly. The approach above is known as the **MAPE-K loop** (IBM, 2006).

Consider a CP intent that operates in a microservice-based CCDN ecosystem which requires it to be scaled to meet the demand of a load increment. However, this increment needs to be analyzed and planned to govern the scaling behavior (e.g., cache size, cache placement, refinement technique, earlier scaling, etc). Therefore, in many cases, a service orchestrator (i.e., Kubernetes) is needed to steer reconfigurations in the correct direction. However, some additional problem-specific decision-making is needed with correspondence to the CCDN intents to regulate and manage the intent refinement with respect to the intent target.

Referring to our proposed Intent-Based CCDN, the MAPE-K Loop is mapped to our Framework in Figure 3.12. Its steps are achieved as follows:

- **Monitoring** is performed by continuously collecting metrics from the deployed microservice containers (i.e., cache containers). In specific, the refinement-related

triggering metrics and other alerts are being watched throughout the service lifetime. This is realized via the **OSS** module in our Framework.

- **Analysis** is to extract statistics from the monitored data and metrics. This involves filtering, data normalization, transformation, etc. For example, calculating demand throughput, demand bursts, and demand peak hours. This maps to the **Intent Technical Requirements calculator** module in our Framework.
- **Planning** is performed according to the initial customization provided by the responsible IT personnel who plans for the required initial configurations and set the policies of the system and infrastructure behavior that could be managed by the infrastructure environment which is maintained in a Knowledge Base (**Intent and Policies Descriptors** in our Framework). The plans are obtained by our **Intent Technical Requirements calculator** module.
- **Execution** is performed by Kubernetes, for the planned actions. Based on our Framework, the **Service Orchestrator** is responsible for this step after it receives the corresponding microservice API calls from the **Intent Technical**

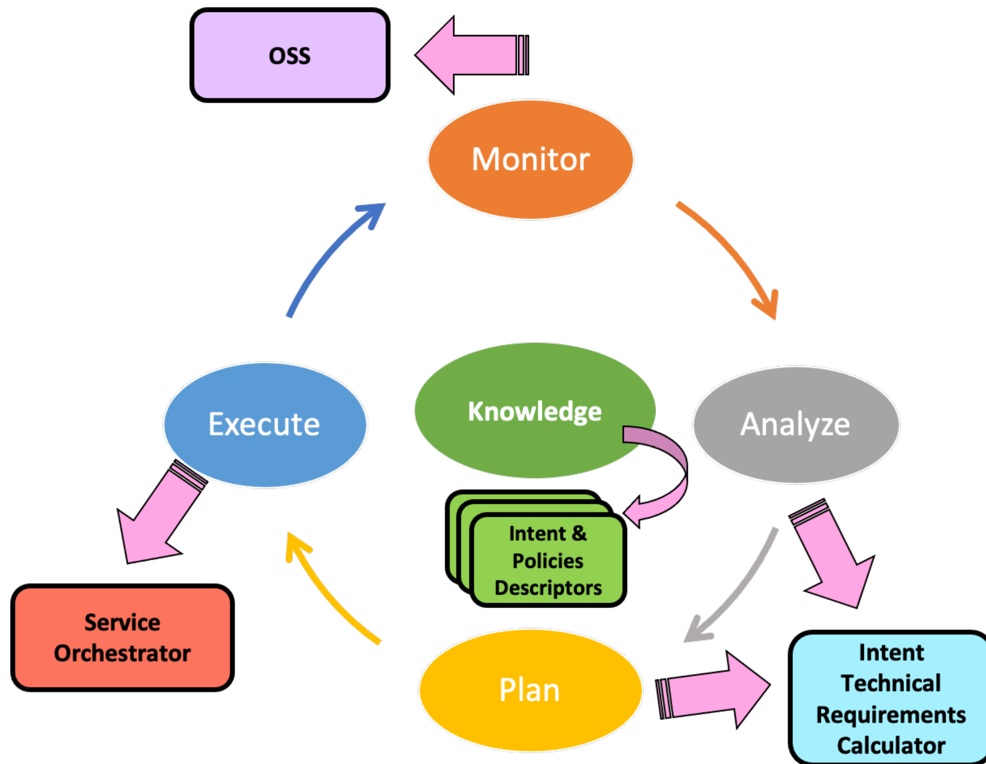

Figure 3.12: MAPE-K Loop Mapping to Our Framework Components.

Requirements calculator.

3.7 Discussion

The Intent-Based CCDN architecture described in this chapter is designed to meet the aforementioned goals and aims. Opposed to the current CCDNs that do not enable the CPs to express their high-level intents or include that in their decision-making, we proposed an Intent-Based communication architecture and mechanism between the CP and the CCDN that enables CPs to express their high-level intents and get involved in the feedback loop with the CCDN, and even optionally participate in the decision-making (i.e., intent refinement decisions). Even though the MSA is the current trend in systems paradigms, we still need to investigate its adoption in the CCDN domain. The MSA advantages discussed in this chapter introduce a promising dynamic and continuously evolving ecosystem that could be leveraged in next deployment phase of Intent-Based CCDNs and facilitate the realization of a wide variety of intent targets. Since microservices could offer several alternatives for the same functionality, and hence, many different CCDN deployments could be enumerated, it is important to have an MCDM tool that helps with the reasoning in terms of microservices selection which involves finding at runtime the best architectural microservice alternatives to achieve the desired intent goals. We chose the popular widely-used AHP technique and discussed the advantages that led us to choose it, and we described our AHP example corresponding to a proposed Low-Cost Intent which aims at cost reduction when deploying the CCDN. Furthermore, we clustered all possible enumerated CCDN deployments into 3 clusters depending on their overall capability to achieve the intent target. This expands the number of available CCDN solutions rather than selecting only the deployment with the highest score, instead, all CCDN deployments are prioritized and could be selected based on their availability. Additionally, we discussed how our architecture components map to the MAPE-K Loop approach for self-adaptation which leads to autonomicity to manage the intent dynamic and reactive refinement automatically with respect to the intent target. Accordingly, we tackled the intent creation on a broader scope from a CCDN operator point of view. In Chapter 4, we will discuss our proposed Low-Cost intents and their refinements in detail.

Chapter 4

Implementation

In the previous chapter, we described the components that make up our proposed CCDN framework. We also discussed the utilization of microservices and their selection based on AHP. Once these are taken into consideration, there is a number of elements that must be implemented to facilitate CP intents. Firstly, the overall communication flow between the CP and the CCDN is described in Section 4.1 which shows how the CP intent passes through the CCDN framework modules to get translated eventually into its corresponding CCDN deployment. This shall be refined later to improve the CCDN performance while keeping the CP in the feedback loop throughout the intent lifetime.

Accordingly, we approach our framework implementation from a CCDN operator perspective (as an intent developer/creator) who has to design, implement, and manage the intent translation which aims at selecting the best available CCDN deployment that meets the intent target. Moreover, the CCDN operator should consider intent refinement which tries to improve the CCDN performance. Therefore, we describe the implementation of our proposed solution in Section 4.2, then we elaborate on the intent translation process in Section 4.3, which is broken down into two CCDN deployment phases: pre- and post- deployment. The former corresponds to the initial intent translation which results in a CCDN deployment selection, and the latter corresponds to the intent and CCDN deployment refinement after running the service.

In specific, we focus on proposing and implementing a Low-Cost intent for CPs that considers cost reduction while provisioning a CCDN. This intent would be translated into its corresponding less-costly CCDN deployment. However, to demonstrate the flexibility, and scalability provided by adopting microservices, we implement several Low-Cost intent realizations which are composed of different microservices alternatives that collectively form up the whole CCDN. We also implement these Low-Cost intent refinements which differ according to different traffic scenarios.

Finally, in Section 4.4, we present our implemented CCDN deployment via Google Kubernetes Engine (GKE).

4.1 Communication Flow between Content Provider and the CCDN

First, we start with the sequence diagram shown in Figure 4.1 to break down the communication flow between the CP and the CCDN system with correspondence to our proposed Intent-Based CCDN framework components. In this diagram, we discuss how the CP high-level intent passes through the CCDN framework modules to get translated eventually into its corresponding CCDN deployment. Each deployment is composed of microservice alternatives that collectively form the highest-scoring CCDN deployment that could achieve the intent target. This initial translation is the first step that takes place during the pre-deployment phase. Later, this CCDN deployment would be refined to improve the CCDN performance, while keeping the CP in the feedback loop throughout the intent lifetime. The communication flow is broken down as follows:

- **Pre-Deployment Phase (black arrows in the diagram)**

  1. The flow starts with the **intent request** from the CP, who can express intents from available provided intent expressions (e.g., *Caching with Low-Cost*)
  2. The high-level intent then needs to be translated, so the **MCDM module** calculates the equivalent AHP graph corresponding to the intent input. The output would be a score ranking for each microservice that could be used in the CCDN deployment. This score calculation and ranking for each microservice is done with correspondence to the evaluation criteria defined in the AHP graph, where their degree of contribution to the CCDN solution depends on the CP intent goal. We elaborate on this in Section 4.3.
  3. After the AHP scores have been calculated for each microservice, the **CCDN Deployments Enumeration and Clustering** processes take place. This module enumerates all possible CCDN deployments by selecting a microservice instance from each different type that collectively could constitute a CCDN deployment. We further elaborate on this in Section 4.3.2.
  4. Once a suggested CCDN deployment has been selected from the high-scored cluster depending on the resource's availability, then it has yet to be technically defined by the **Intent Technical Requirements Calculator** module which calculates the corresponding operational parameters for its technical requirements (e.g., Max and Min No. Caches, Cache container image, etc.)

5. The availability of this selected CCDN requirements has to be checked by the **Monitoring** module which reports back its availability to the **Intent Technical Requirements Calculator**
6. The availability of the suggested CCDN would be then reported back to the CP for their input even if the initial CP's intent target could not be met (e.g., due to unavailable resources) where some downgraded service alternative could be suggested and the CP could then decide (i.e., approve, decline)
7. The CP responds to the initial suggested and available CCDN that would help achieving his/her intent (i.e., approve, decline)
8. If the suggested CCDN was approved by the CP, the **Intent Technical Requirements Calculator** formulates the corresponding microservice API calls and sends them to the **Service Orchestrator** (i.e., Kubernetes Controller) for service deployment
9. **The Intent Technical Requirements Calculator** identifies and sends the intent refinement metrics (e.g., peak hours, traffic bursts, etc.) to be monitored to the **Monitoring** module, to enable reactive intent refinement during runtime

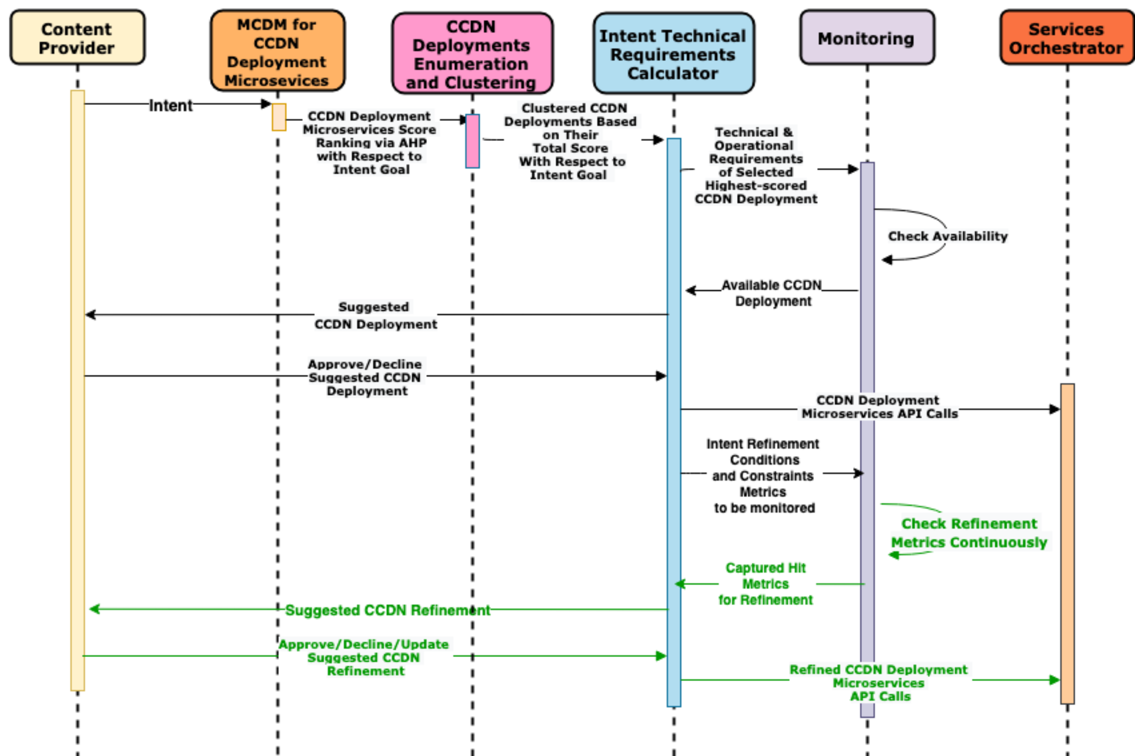

Figure 4.1: Communication Flow Between Content Provider and the CCDN.

- **Post-Deployment Phase (green arrows in the diagram)**

1. The **Monitor** module continuously watches out for the refinement metrics, if any of them has hit the set limit by the **Intent Technical Requirements Calculator**, then it alerts it
2. The **Intent Technical Requirements Calculator** identifies and calculates the suggested CCDN refinement and repots this back to the CP
3. The CP responds to the suggested refinement (i.e., approve/decline/update)
4. If the suggested refined CCDN was approved or updated (e.g., updated peak hours by the CP), then the **Intent Technical Requirements Calculator** formulates the corresponding microservice API calls and sends them to the **Service Orchestrator**

4.2 Implementing the Intent-Based CCDN

This section details the implementation of the Intent-Based CCDN based on the design shown in Section 3.3 and in Figure 4.1. All modules have been implemented in Python as follows:

- **Multi-Criteria Decision-Making Module:** has been implemented with the help of the ***AHPy*** Python library (Griffith, 2023) which has been used to easily allow defining AHP elements and pairwise comparative score ratios in the form of Python dictionary elements, and perform the computations of the AHP decision-making process to help assign the total scores for each microservice instance with respect to the overall intent target

- **CCDN Deployments Enumeration and Clustering Module:** this component has been implemented to enumerate all possible CCDN deployments and then leverage the assistance of the **KMeans** clustering algorithm, which is a very popular clustering method (Jin et al., 2017). The `sklearn` Python library has been used, which provides the KMeans method computation (Pedregosa et al., 2011) to facilitate the CCDN deployments clustering process

- **Intent Technical Requirements Calculator Module (Intent Handler):** is considered as the hub. It has been implemented to serve as the main intent manager and communicator between the CP and the CCDN system

- **Monitoring Module:** this component has been implemented while leveraging the direct accessibility to the Service Orchestrator's (i.e., Kubernetes Controller) overall service and resources status

4.3 Intent Translation

In this section, we elaborate on the CP intent translation process from a CCDN operator (i.e., intent developer and provider) point of view, who is responsible for defining the AHP graph and has to determine its components and scores based on his knowledge, user requirements, resource availability, etc. Moreover, an intent developer needs to address several main steps to develop an intent translation from a high-level declarative syntax all the way down to low-level technical commands (microservice API calls), that would be carried out by the underlying technology.

4.3.1 Multi-Criteria Decision-Making Module

AHP is our chosen MCDM method for the reasons discussed in Chapter 3 Section 3.4.1. A CCDN operator is responsible for building the AHP hierarchy, where he has to define the AHP goal (intent goal) in the first level of the hierarchy, the main evaluation criteria in the second level, the sub-criteria in the third (or more) level, and the microservice alternatives in the last level of the hierarchy. In the context of CCDN, our JSON expression for the AHP graph is shown in Listing 4.1. We choose JSON because of its simplicity, clarity, and popularity. For instance, if there have been 2 main evaluation criteria defined in the AHP graph: *scalability* and *cost*. When the CP intent is to get the best available performance, then this would require assigning a much greater weight of importance to the scalability criterion as opposed to the cost criterion (e.g., *scalability weight = 0.9* and *cost weight = 0.1*). Accordingly, the overall score calculation for each microservice alternative would differ. An output example of the AHP calculation is shown in Listing 4.2 which ranks all microservices based on their total score of contribution towards the evaluation criteria related to the intent goal.

To carry on with the AHP whole calculations, ***AHPy*** Python library has been used (Griffith, 2023). This easily allows defining AHP elements and pairwise comparative score ratios in the form of Python dictionary elements. Each dictionary *key* is a comparison pair (i.e., (*a, b*)), and its *value* is their weight ratio (i.e., *weight(a)/weight(b)*), then ***AHPy*** manages all the corresponding computations in order to rank all microservices based on their global scores that denote their level of contribution to the AHP problem goal. The main ***AHPy*** API elements provided: *compare*, *add_children()*, *report()*.

- **Compare Class**: This class computes the weights and consistency ratio of a positive reciprocal matrix that represents the pairwise comparison. An object of this class is created using an input of a Python dictionary which holds the pairwise comparison values. The Compare objects could also be linked together to form a

```
self.AHP = {
    "name": "CDN Deployment",
    "criteria": [
        {"Scalability": self.scaclability_weight},
        {"Cost": self.cost_weight}
     ],
    "subCriteria": {
      "Scalability": [
          {"Cache Size": self.cache_size_weight},
          {"Cache Placement Startup Delay": self.startup_weight},
          {"Refinement": self.refinement_weight}
       ],
      "Cost": [
          {"Cache Size Cost": self.cache_cost_weight},
          {"Cache Placement Cost": self.placement_cost_weight},
          {"Refinement Cost": self.refinement_cost}
       ]
    },
    "alternatives":{
        "Cache Sizes": ["Big Cache", "Medium Cache", "Small Cache"],
        "Cache Placement": ["Local", "Nearby", "Further"],
        "Refinements": ["Primary Pessimistic", "Primary Optimistic", "Secondary"]
    },
    "alternatives weights": {
        "Cache Size": [
            {"Big Cache": self.big_cache_size_weight},
            {"Medium Cache": self.medium_cache_size_weight},
            {"Small Cache": self.small_cache_size_weight}
        ],
        "Cache Size Cost":[
            {"Big Cache": self.big_cache_cost},
            {"Medium Cache": self.medium_cache_cost},
            {"Small Cache": self.small_cache_cost}
        ],
        "Cache Placement Startup Delay": [
            {"Local": self.zone20_startup_weight},
            {"Nearby": self.zone40_startup_weight},
            {"Further": self.zone60_startup_weight}
        ],
        "Cache Placement Cost": [
            {"Local": self.zone20_startup_cost},
            {"Nearby": self.zone40_startup_cost},
            {"Further": self.zone60_startup_cost}
        ],
        "Refinement": [
           {"Primary Pessimistic": self.primary_pess_ref_scal},
           {"Primary Optimistic": self.primary_opt_ref_scal},
           {"Secondary": self.secondary_ref_scal}
           ],
        "Refinement Cost": [
           {"Primary Pessimistic": self.primary_pess_ref_cost},
           {"Primary Optimistic": self.primary_opt_ref_cost},
           {"Secondary": self.secondary_ref_cost}
           ]
    }
}
```

Listing 4.1: The JSON Representation of the AHP Graph Used in CP Intents Translation.

```
"Microservice Scores:"
{"Small Cache": 0.3511, "Further": 0.1746, "Medium Cache": 0.1283, "Big Cache": 0.0921,
"Secondary": 0.0874, "Nearby": 0.0635, "Local": 0.0476, "Primary Optimistic": 0.0318,
"Primary Pessimistic": 0.0238}
```

Listing 4.2: AHP Output for Ranked Microservices Scores.

hierarchy representing the AHP decision problem. The target weights (i.e., scores) of the AHP elements are then derived by synthesizing all levels of the hierarchy. The Compare object syntax and parameters are as follows:

`Compare(name, comparisons, precision=4, random_index='dd', iterations=100, tolerance=0.0001, cr=True)`

The most important parameters are:

`name: str (required)`, the Compare object's name which is used to link a child object to its parent and must be unique.

`comparisons: dict (required)`, pairwise comparison dictionary

`random_index`: 'dd' or 'saaty', denotes the set of random index estimates used to compute the consistency ratio for the Compare object. The default random index is 'dd'

'dd' (Donegan et al., 1991) supports the calculation of the consistency ratios for matrices less than or equal to 100×100 in size

'saaty' (Thomas L Saaty et al., 2012) supports the calculation of the consistency ratios for matrices less than or equal to 15×15 in size

- **Compare.add_children()**: The Compare objects can be linked together to form the AHP hierarchy that represents the decision problem. This method is called on the Compare object that forms the *upper* level (the parent) and includes an argument as a list or tuple of one or more Compare objects that are intended to form its *lower* level (the children).
- **Compare.report()**: By calling this method on the Compare object, it returns a dictionary that contains a standard report on the details of a Compare object.

4.3.2 CCDN Deployments Enumeration and Clustering Module

In this module, all possible CCDN deployments are enumerated, which consist of a microservice instance of every different type. Since each microservice in

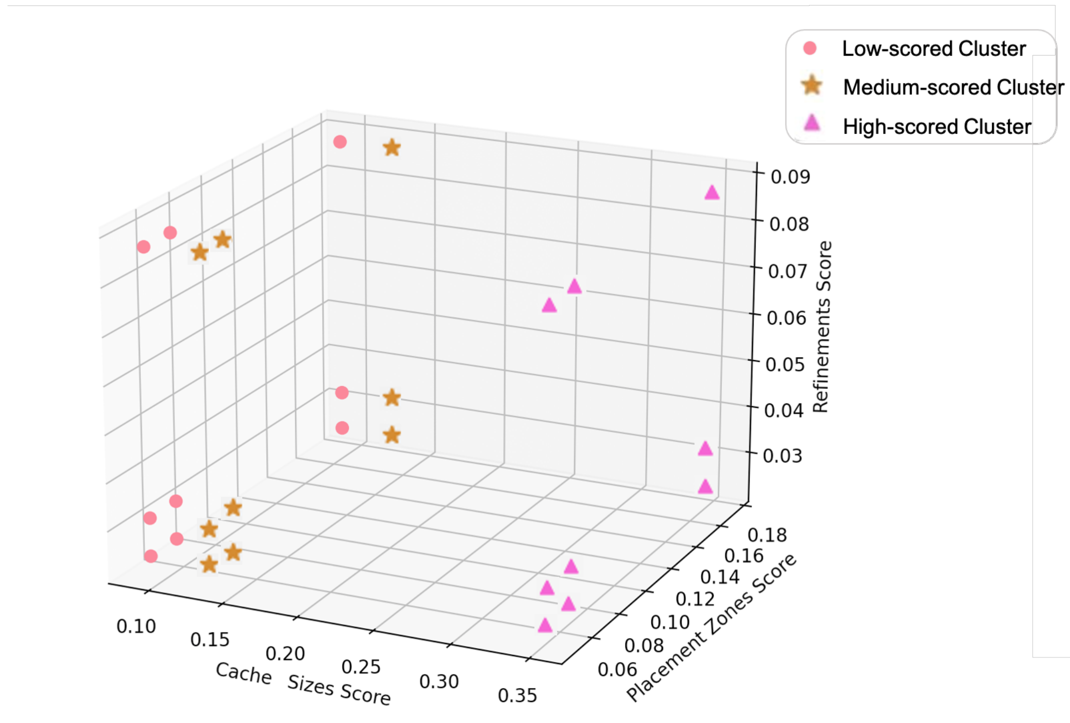

Figure 4.2: Clustered CCDN Deployments Based on Their Scores Towards the Intent Goal.

the enumerated CCDN deployments has a score, the overall combined score of the deployment can be calculated by summing up the scores of all microservices included in that deployment. Once the total overall ranking for each possible CCDN deployment is achieved, the clustering process takes place which groups CCDN deployments with close or similar scores into a cluster as shown in Figure 4.2. An output example is shown in Figure 4.3

We use the **KMeans** clustering algorithm, which is a very popular clustering method (Jin et al., 2017). We use `sklearn` Python library that provides the KMeans method computation (Pedregosa et al., 2011). Since each CCDN deployment is constituted of microservices from different types, we use a multi-dimensional KMeans clustering of the deployments where each microservice type is a dimension. For instance, if a CCDN deployment consists of *cache size*, *cache placement*, and *refinement,* then we perform a 3-dimensional clustering based on the scores of each microservice instance in the deployments. We choose 3 clusters: high, medium, and low, which group similar-scored deployments based on their level of capability to contribute to the intent (AHP goal). Although this clustering was unsupervised, we still need to rank these clusters as high, medium, and low. So, we label each cluster

```
low
('Big Cache', 'Local', 'Primary Pessimistic')
('Big Cache', 'Local', 'Primary Optimistic')
('Big Cache', 'Local', 'Secondary')
('Big Cache', 'Nearby', 'Primary Pessimistic')
('Big Cache', 'Nearby', 'Primary Optimistic')
('Big Cache', 'Nearby', 'Secondary')
('Medium Cache', 'Local', 'Primary Pessimistic')
('Medium Cache', 'Local', 'Primary Optimistic')
('Medium Cache', 'Local', 'Secondary')
('Medium Cache', 'Nearby', 'Primary Pessimistic')
('Medium Cache', 'Nearby', 'Primary Optimistic')
('Medium Cache', 'Nearby', 'Secondary')
medium
('Big Cache', 'Further', 'Primary Pessimistic')
('Big Cache', 'Further', 'Primary Optimistic')
('Big Cache', 'Further', 'Secondary')
('Medium Cache', 'Further', 'Primary Pessimistic')
('Medium Cache', 'Further', 'Primary Optimistic')
('Medium Cache', 'Further', 'Secondary')
high
('Small Cache', 'Local', 'Primary Pessimistic')
('Small Cache', 'Local', 'Primary Optimistic')
('Small Cache', 'Local', 'Secondary')
('Small Cache', 'Nearby', 'Primary Pessimistic')
('Small Cache', 'Nearby', 'Primary Optimistic')
('Small Cache', 'Nearby', 'Secondary')
('Small Cache', 'Further', 'Primary Pessimistic')
('Small Cache', 'Further', 'Primary Optimistic')
('Small Cache', 'Further', 'Secondary')
Number of Deployments with (High) ranking: 9
Number of Deployments with (Medium) ranking: 6
Number of Deployments with (Low) ranking: 12
Total number of deployments: 27
```

Figure 4.3: Total CCDN Deployments Clusters Based on Their Scores Towards the Intent Goal.

based on the comparison of the maximum value for each. Accordingly, the cluster with the highest maximum value was labeled as "high", then the cluster with the second highest value was labeled as "medium" and the cluster with the lowest maximum value was labeled as "low".

4.3.3 Intent Technical Requirements Calculator Module

This module is responsible for calculating the operational parameters and technical requirements needed for deploying, running, and refining the CCDN. These parameters are dependent on both the regulatory policies stated by the CCDN operator and the underlying technology. They are calculated during runtime according to the requested intent target. Some examples of these parameters: *max/min number of caches, cache container image, cache CPU/RAM resources, and load balancing scaling threshold (i.e., CPU utilization, requests rate, etc.).*

This module is also responsible for triggering and managing the intent refinement process according to the continuous tracking of the related parameters by the

Monitoring module. The refinement metrics and functionality have to be defined in the regulatory policies as part of the intent translation breakdown, and calculated and implemented reactively once certain thresholds have been exceeded.

In specific, we propose 3 **Low-Cost Intent realizations (LCIs)** that aim to provide the CP with a lower-cost CCDN, compared to the default non-Intent-Based CCDNs. We assume that the CCDN has different cache microservices with different sizes (i.e., *small, medium, big*) and costs that depend on their location within the CCDN region (i.e., *local zone, nearby zone, further zone cache cost*). Our proposed LCIs corresponding CCDNs are:

- **Low-Cost Intent 1 (LCI 1)**: **Medium** cheaper caches that are placed in a **nearby zone** rather than the local and more costly ones. This intent trades cost reduction for increased *startup delay* since the cheaper caches are not located locally
- **Low-Cost Intent 2 (LCI 2)**: **Small** caches **locally** placed in the same end-users zone. The small cache's cost is less than (i.e., half) the medium (default) cache's cost. This intent trades cost reduction for a *smaller cache size*
- **Low-Cost Intent 3 (LCI 3)**: **Small** cheaper caches placed in a **nearby zone**. This intent trades cost reduction for both more *startup delay* and smaller *cache size*

In LCI 1, The tradeoff of deploying cheaper caches that are placed in a nearby zone is the increased startup delay that could affect the performance negatively by increasing the possibility of request drops.

When small caches in LCI 2 are deployed in this CCDN (as opposed to medium default caches), this leads to more frequent scale-outs due to having smaller cache size (cheaper than bigger ones) that results in more drops and eventually performs significantly worse than the default CCDN.

As in LCI 3, combining both cost reduction factors in the previous intents, small cheaper caches are deployed in a nearby zone. Naturally, this CCDN would suffer from the highest droppings compared to the former ones in return for the highest cost reduction.

We propose two intent refinements: *optimistic* and *pessimistic*. The former, performs the refinement within a smaller time window, whereas the latter aims at improving the performance more than the former. Thus, it has a bigger time window for the refinement process (e.g., local cache placement, vertical upgrade, or both).

We consider two types of traffic patterns: **gradual increase** and **traffic bursts**. The former resembles a normal traffic pattern whereas the latter resembles the abnormal sudden traffic surges that need a faster resolution compared to the former. Due to the different nature of each scenario, we shall handle their refinement differently as follows:

4.3.3.1 Usual traffic behavior with gradual traffic increase

In normal cases, traffic would follow a similar pattern which starts at a very low request rate during non-active hours (i.e., late at night or too early in the morning) and then increases gradually throughout the active hours of the day until it reaches its peak during the peak hours, then drops gradually until it reaches the lowest requests rate again, and so on. Usually, the peak hours require the highest number of cache scaleouts to accommodate the traffic of the intense request, which could lead to a higher number of dropped requests during the scaling process until the new caches are up and running. Hence, that is specifically when we are interested in the LCIs refinement that takes place to alleviate the compromised performance of the original intents. Each intent correspondent CCDN would be **refined differently** as follows during peak hours:

- **LCI 1** CCDN deploys **local** caches during peak hours where the traffic load is the most intense and needs faster processing, rather than the original nearby ones. This is depicted in Algorithm 1.
- **LCI 2** CCDN vertically upgrades small caches to **medium** ones during peak hours to reduce the need to scale out. Hence, reduce the droppings during the scaling process. This is shown in Algorithm 2
- **LCI 3** CCDN vertically upgrades small nearby caches to **local medium** ones to reduce scale-outs and startup delay. The detailed process is shown in Algorithm 3

Accordingly, one important refinement metric is the ***"peak hours"***, this could be calculated in different ways as the CCDN operator sees fit, and based on the traffic behavior and history observation. However, we suggest either setting a proactive value by defining specific start and end times or defining a reactive value that corresponds to the traffic rate (e.g., when the traffic rate is 30% greater than the average traffic rate). Both can be determined based on the analysis of the traffic pattern history and the service nature.

4.3.3.2 Bursty Traffic behavior with sudden increase

In some cases, there could be some bursts of traffic due to the popularity of the content, seasonal popularity, etc. In our definition, traffic bursts are sudden surges in incoming requests traffic that significantly exceeds the average handled traffic rate at that time and last for some certain period. However, to differentiate between traffic bursts and random traffic spikes, we have set a time window (e.g., 3 continuous minutes) in which the traffic surge exceeds, then it is considered to be a traffic burst rather than just a random spike, and thus, it has to be handled differently during the

Algorithm 1 LCI 1 CCDN Refinement During Peak Hours

```
S ← CurrentActiveCacheSize
Min_C ← MinNo.CurrentCaches
Max_C ← MaxNo.CurrentCaches
Zone_C ← CurrentCachesNearbyZone
Zone_R ← RefinedCachesLocalZone
A_D ← Dep(S, Min_C, Max_C, Zone_C)          ▷ Current Active CCDN Deployment
CCDNrefined ← FALSE
while CCDN.Service.Running is TRUE do
    if PeakHours is true then
        update(A_D) ← Dep(S, Min_C, Max_C, Zone_R)
        CCDNrefined ← TRUE
    else   ▷ when peak hours are over, re-locate caches in the previous non-refined nearby cache zone
        if CCDNrefined is true then
            update(A_D) ← Dep(S, Min_C, Max_C, Zone_C)
            CCDNrefined ← FALSE
        end if
    end if
end while
```

Algorithm 2 LCI 2 CCDN Refinement During Peak Hours

$S \leftarrow CurrentActiveCacheSize$
$Min_{\mathrm{C}} \leftarrow MinNo.CurrentCaches$
$Max_{\mathrm{C}} \leftarrow MaxNo.CurrentCaches$
$Max_{\mathrm{R}} \leftarrow MaxNo.RefinedUpgradedCaches$
$A_{\mathrm{D}} \leftarrow Dep(S, Min_{\mathrm{C}}, Max_{\mathrm{C}})$ ▷ Current Active CCDN Deployment
$CCDNrefined \leftarrow FALSE$
while $CCDN.Service.Running$ is TRUE **do**
  if $PeakHours$ is true **then**
    update(A_{D}) $\leftarrow Dep(S \times 2, \lceil \frac{Min_{\mathrm{C}}}{2} \rceil$, Max $_{\mathrm{R}})$ CCDNrefined $\leftarrow TRUE$

  else
    ▷ when peak hours are over, downgrade CCDN to the previous non-refined deployment, with the suitable number of caches
    if $CCDNrefined$ is true **then**
      update(A_{D}) $\leftarrow Dep(\lceil \frac{S}{2} \rceil$, Min $_{\mathrm{C}} \times 2, Max_{\mathrm{C}})$
      $CCDNrefined \leftarrow FALSE$
    end if
  end if
end while

Algorithm 3 LCI 3 CCDN Refinement During Peak Hours

$S \leftarrow CurrentActiveCacheSize$
$Min_{\mathrm{C}} \leftarrow MinNo.CurrentCaches$
$Max_{\mathrm{C}} \leftarrow MaxNo.CurrentCaches$
$Max_{\mathrm{R}} \leftarrow MaxNo.RefinedUpgradedCaches$
$Zone_{\mathrm{C}} \leftarrow CurrentCachesNearbyZone$
$Zone_{\mathrm{R}} \leftarrow RefinedCachesLocalZone$
$A_{\mathrm{D}} \leftarrow Dep(S, Min_{\mathrm{C}}, Max_{\mathrm{C}}, Zone_{\mathrm{C}})$ ▷ Current Active CCDN Deployment
$CCDNrefined \leftarrow FALSE$
while $CCDN.Service.Running$ is TRUE **do**
  if $PeakHours$ is true **then**
    update(A_{D}) $\leftarrow Dep(S \times 2, \lceil \frac{Min_{\mathrm{C}}}{2} \rceil$, Max $_{\mathrm{R}}$, Zone $_{\mathrm{R}}$) CCDNrefined $\leftarrow$ $TRUE$
  else
    ▷ when peak hours are over, downgrade CCDN to the previous non-refined deployment, with the suitable number of caches and re-locate them in the previous nearby zone
    if $CCDNrefined$ is true **then**
      update(A_{D}) $\leftarrow Dep(\lceil \frac{S}{2} \rceil$, Min $_{\mathrm{C}} \times 2, Max_{\mathrm{C}}, Zone_{\mathrm{C}})$
      $CCDNrefined \leftarrow FALSE$
    end if
  end if
end while

low-cost intent refinement due to its sudden occurrence that has to be handled in a timely manner.

We focus on LCI 2 refinement in this case since it depends on the more impactful factor (i.e., smaller cache size) on the performance degradation in the normal traffic behavior, and thus we want to investigate the scale of this degradation in a bursty traffic situation. We argue that the previous LCI 2 refinement proposed in the normal traffic scenario would not be that effective with traffic bursts. Therefore, we propose a different LCI 2 refinement in bursty traffic cases. Opposed to the previously proposed LCI 2 refinement which vertically upgrades small caches to medium in the normal traffic scenario, it is important to opt for a faster refinement solution that could accommodate the sudden traffic burst quickly, since the vertical upgrade delay might lead to increased droppings during the burst. In this case, the refined LCI 2 aims at speeding up the scaling process of the existing small caches without the need to vertically upgrade them. Hence, once a traffic burst is detected, then the horizontal auto-scaling threshold is lowered to allow a faster scale-out process. For example, the default horizontal scaling threshold is 80% of the average CPU utilization of all existing caches. When this threshold gets exceeded then a new cache container is added. But in order to speed up the scaling process during traffic bursts, we lower this threshold to 65% of average CPU utilization. This refinement is shown in Algorithm4.

Accordingly, one important refinement trigger is the ***"traffic burst"*** metric. A CCDN operator has to define a traffic burst (i.e., *at least N continuous minutes of traffic rate above average with a certain percentage X%*). After some stabilization time window, the horizontal scaling threshold goes back to the default value.

4.4 CCDN Deployment via Google Kubernetes Engine (GKE)

We create CCDN deployments using Google Kubernetes Engine (GKE). This widely used powerful platform runs on Google Cloud Infrastructure spreading across 37 regions and 112 zones that are available in over 200 countries around the world (Google, 2023e), it provides a managed environment to deploy, manage, and scale containerized applications using Google infrastructure. The GKE environment usually consists of multiple worker machines (called nodes) grouped together to form a GKE cluster powered by Kubernetes; an open-source cluster management system (Kubernetes, 2023c). These nodes run containerized applications such that they host the Pods that are the components of the application workload. A Pod is the smallest deployable unit to create and manage in Kubernetes. It is a group of one or more containers, with shared resources, and contains specifications for how to run the containers.

Algorithm 4 LCI 2 CCDN Refinement During Traffic Bursts

```
S ← CurrentActiveCacheSize
Min_C ← MinNo.CurrentCaches
Max_C ← MaxNo.CurrentCaches
Min_R ← MinNo.RefinedCaches
Max_R ← MaxNo.RefinedCaches
ScaleThresh_C ← CurrentHigherScalingThreshold
ScaleThresh_R ← RefinedLowerScalingThreshold
W ← StabilizationWindowTime
A_D ← Dep(S, Min_C, Max_C, ScaleThresh_C)   ▷ Current Active CCDN Deployment
CCDNrefined ← FALSE
while CCDN.Service.Running is TRUE do
    if TrafficBurst is true then
        update(A_D) ← Dep(S, Min_R, Max_R, ScaleThresh_R)
        CCDNrefined ← TRUE
    else        ▷ when traffic burst is over, restore scaling threshold to the previous non-refined scaling
        if CCDNrefined is true & W is exceeded then
            update(A_D) ← Dep(S, Min_C, Max_C, ScaleThresh_C)
            CCDNrefined ← FALSE
        end if
    end if
end while
```

4.4.1 Kubernetes Objects

Kubernetes can be managed via specifying data in a YAML file, typically to define a Kubernetes object. The YAML configuration file is called a "manifest", and when it is applied to a Kubernetes cluster, the corresponding Kubernetes object is created based on the configuration.

4.4.1.1 Kubernetes Deployment

The Deployment object creates the pods, ensures their correct number is always running in the cluster, handles scalability, and takes care of the pod updates continuously (Kubernetes, 2023a). All these activities can be configured through their corresponding fields in the Deployment YAML manifest.

When a Pod is specified, you can optionally specify how much of each resource a container needs. The most common resources to specify are CPU and memory (RAM). In this context, we focused on these two resources and excluded the disk resources due to the nature of our request processing (compute- and memory- bound nature). You can set `request`, and `limit` for these resources.

CPU resource represents compute processing specified in units of Kubernetes CPUs, where 1 CPU unit is equivalent to 1 physical CPU core, or 1 virtual core, depending on whether the node is a physical or virtual machine. Whereas, Memory is specified in units of bytes.

Resource `request` specifies the amount of requested resources for containers in a Pod. The kube-scheduler uses this information to decide which node to place the Pod on. Resource `limit` specifies the maximum limit of resources for a container, which is not allowed to use more of that resource than the specified limit.

A probe is a periodic check that monitors the health of an application. One of these control probes is `initialDelaySeconds` which determines the waiting time after the container starts and before beginning the probe.

In our implementation of the CCDN cache containers, we use the popular *Nginx* server as our cache container image (NGINX, 2023). Since we have different cache sizes (i.e., small and medium), we set different resources (`request` and `limit`) such that the *medium* caches have `twice` CPU and RAM resources as the *small* caches. Each cache size is specified in its own deployment YAML manifest since each differs in specifications. As for the cache placement in different zones that could impose different startup delays based on their proximity to the end-users, we set the `initialDelaySeconds` metric to emulate the different cache startup times which represent starting and running caches in different zones. A cache in a Local zone translates to a lower `initialDelaySeconds` than in nearby or further zones. A snippet of the corresponding parts of the deployment manifest file is shown in Listing 4.3. Having different deployment manifests for different cache sizes and their

specifications gives us the advantage of running them separately and in parallel when needed, without requiring to update the currently active deployment which would temporarily take it down until the updated one is deployed. In specific, this feature is helpful in the LCI refinements with vertical upgrades during peak hours. As the default small caches run throughout the day, only during the peak hours do we upgrade to medium caches, but to reduce the possible droppings in the process, we deploy medium caches in parallel with the small ones just right before the peak hours to ensure that the refined medium caches are up and running and then can be used, so no downtime occurs.

```
spec:
 containers:
  -name: nginx
     image: nginx
     resources:
       limits:
         cpu: 200m
         memory: 4000Mi
       requests:
         cpu: 100m
         memory: 2000Mi
     readinessProbe:
       initialDelaySeconds: 40
     ports:
      -containerPort: 80
```

Listing 4.3: Container Resources and Image Specification Segment.

4.4.1.2 Horizontal Pod Autoscaler

A Horizontal Pod Autoscaler (HPA) automatically deploys more pods with the aim of automatically scaling the workload to match the increasing demand (Kubernetes, 2023b). It also scales pods back down automatically when the load decreases. HPA refers to a scaling threshold, if it got exceeded then the scaling takes place. The most popular threshold is `targetAverageUtilization` which is computed by taking the average of CPU utilization across all Pods.

The `minReplicas` and `maxReplicas` number of replicas could be determined, which sets the range of replicas that could be scaled by the HPA.

In our implementation, we define different HPAs for each deployment as the they differ in their specifications (i.e., cache size). The `maxReplicas` and `minReplicas`

differ according to the deployment. For example, with *small* caches, we need **twice** the number of *medium* caches since the former has **half** the resources of the latter.

In our proposed LCI 2 refinement during traffic bursts, we can update the scaling threshold `targetCPUUtilizationPercentage`, by lowering it (e.g., from 80% to 65%), and then restore it again after the bursts are over. Luckily, updating the HPA wouldn't take the CCDN cluster down as opposed to updating the deployment manifests. A snippet of the corresponding parts of the HPA manifest file is shown in Listing 4.4.

```
spec:
  maxReplicas: 20
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deployment-small_caches
  targetCPUUtilizationPercentage: 80
```

Listing 4.4: Horizontal Pod Auto Scaling Segment.

4.5 Summary

In this chapter, we presented the implementation of our proposed Intent-Based CCDN. In particular, we explored how the core components of the prior design are realized in working code. Together, these components facilitate the intent translation and refinement to meet its aims. We also described some of the main Kubernetes manifest segments corresponding to our proposed intents, which denote the final level of translation that creates and applies the microservices API calls via the Kubernetes orchestrator. In the following section, we take this implementation and evaluate it in a number of scenarios along with different LCI alternatives and their refinements, each of which exercises a particular functionality or design requirement.

Chapter 5

Evaluation

In this chapter, we aim to evaluate our implemented intent translation overhead alongside comparing the performance, cost, and performance-to-cost score for several implemented Low-Cost intent alternatives that lead to cost reduction. We also evaluate these intents' refinements which aim at improving their performance. The overall experimental methodology is devised in Section 5.1 At first, we evaluate the overhead of the intent translation process in terms of the time needed to translate the high-level intent into its corresponding CCDN deployment, which would be converted later to microservice API calls. This is presented in Section 5.2. Further to this, we examine the performance (in terms of dropped requests), cost, and a combined performance-to-cost score for our proposed Low-Cost intents that focus on CCDN cost reduction and the level of performance improvement after refining these intents in Section 5.3. Thereupon, these evaluations of the intent alternatives and their refinements have been done during different traffic pattern scenarios. These results reflect the direct benefits to the CPs.

5.1 Experimental Methodology

We split our tests into 2 phases: the *Intent Translation* phase (CCDN Pre-deployment phase), and the *Intent Execution and Refinement* phase (CCDN Post-deployment phase).

5.1.1 Intent Translation Evaluation

We explain the tests we made in our CCDN pre-deployment phase to evaluate the intents translation overhead.

- **Dimensions:** Intents are translated via calculating the corresponding AHP graph which is constituted of evaluation *criteria* and *microservice* alternatives. These

form the dimensions for our tests to investigate the impact of each on the overall intent translation overhead.

- **Scale:** We vary each dimension (AHP criteria and microservices) while fixing the other's value to examine its overhead impact on the overall translation time.
  - First, we increase the number of simulated *criteria* in the AHP graph: 10, 20, 30, up to 100, with 2 microservice types, each has 10 alternatives to be evaluated correspondingly.
  - Then, we examine simulating the *microservice alternatives* scaling: 10, 20, 30, up to 100, for each microservice type (i.e., 2 microservice types) which would be evaluated against 10 criteria.

  The simulations were designed to stress this intent translation process via the AHP graph calculation in order to discover its overhead. Accordingly, this is followed by the score calculation and clustering for varying numbers of possible CCDN deployments that range from *hundreds* of possible deployments to bigger scenarios with a *10K* possible CCDN deployments that could be composed of the microservice alternatives. We obtain 3 clusters (high, medium, and low) of CCDN deployments based on their overall scores that represent their level of contribution towards achieving the intent target. This AHP graph setup in this simulation was pretty straightforward due to the use of the `AHPy` Python library (Griffith, 2023) which was responsible of the heavy lifting of the required AHP computations while offering an easy-to-use API.

- **Metrics:** We measure the *Execution Time* for the translation process to be completed which includes the AHP calculation and CCDN deployments enumeration and clustering based on their scores with respect to the intent goal.

- **Hardware:** The tests were performed in a MacBook Pro with a Quad-Core Intel Core i7 CPU at 2.7 GHz, Turbo Boost up to 3.6 GHz, with 8 MB shared L3 cache and 16 GB of RAM. The simulations were implemented in Python utilizing the `AHPy` Python library (Griffith, 2023) for the AHP calculations, and we use `sklearn` Python library (Pedregosa et al., 2011) for KMeans clustering.

5.1.2 Intent Execution and Refinement Evaluation

We explain the setup we used in our CCDN post-deployment phase to evaluate our proposed Low-Cost intents and their refinements.

- **Dataset:** We utilize real data from a major ISP that represents the measurement of the bitrate of data transferred between the CDN caches (in the ISP's facilities)

and the end-users. Each measure is average bits/second over the granularity of 1 minute. We transform the captured bitrate in the dataset into request rate under the assumption that the user request size is 500 KB which is equivalent to a video chunk size as in this work (Frangoudis et al., 2017). Accordingly, we get the (time stamp, request rate) pair/minute in order to facilitate generating a similar traffic behavior pattern. We test our solution as opposed to the baselines according to the dataset traffic from 13–19 Oct 2018.

- **Traffic Cases:** We test 2 traffic cases based on the used dataset as follows:

  Normal Traffic with Gradual Increase In normal cases, traffic would follow a similar pattern which starts at a very low requests rate during non-active hours (i.e., late at night or too early in the morning) and then increases gradually throughout the active hours of the day until it reaches its peak during the peak hours, then drops gradually until it reaches the lowest requests rate again, and so on. A sample from the dataset is shown in Figure 5.1 where the number of caches that handle the requests increase/decrease accordingly.

  Traffic Bursts In some cases, there could be some bursts of traffic due to the popularity of the content, seasonal popularity, etc. In our definition, traffic bursts are sudden surges in incoming requests traffic that significantly exceed the average handled traffic rate at that period and last for some certain time.

  In this tested scenario, we inject 3 synthetic traffic bursts into the tested traffic segments which did not contain bursts originally in the dataset as it resembled normal traffic pattern. We test them over a period of 1 hour as shown in Figure 5.2.

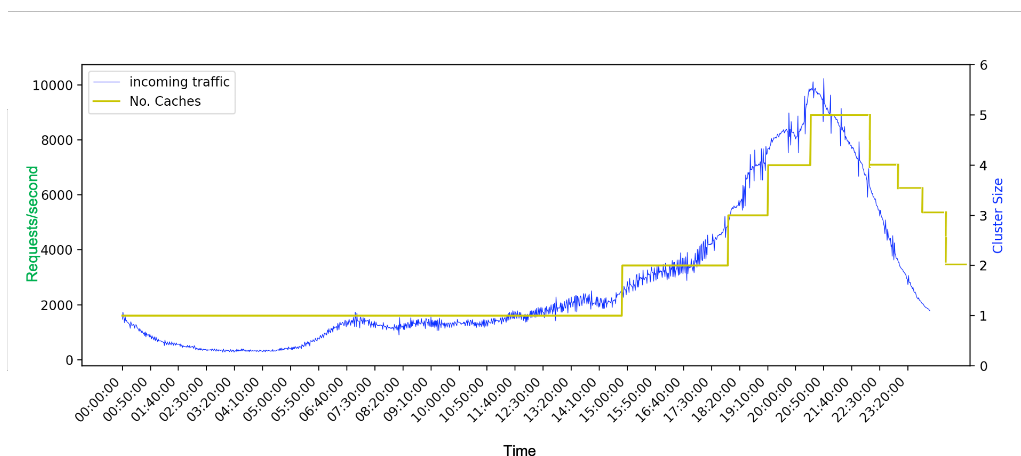

Figure 5.1: Normal Traffic with Gradual Increase.

- **Google Cloud Platform Setup:** Tests on several CCDN deployments have been done using Google Kubernetes Engine (GKE). The setup is shown in Figure 5.3. The Kubernetes cluster consisted of 2 Compute Engines (Nodes). The first node is mainly dedicated to the Ingress controller which manages the external HTTP traffic routing and load balancing to the Kubernetes cluster. The second node is dedicated to hosting all cache pod replicas that handle incoming requests. We dedicated a separate external Compute Engine (VM) located in the same zone to generate the end-users traffic. Each Compute Engine has 2 vCPU and 8 GB RAM.

- **Baselines:** A GKE cluster consists of a control plane and worker machines (nodes). Both make up the Kubernetes cluster orchestration system. GKE offers two operation modes depending on the required control level, these modes are **Autopilot** and **Standard** (Google, 2023d) (Google, 2023a). **Autopilot** mode provides a hands-off Kubernetes experience by providing optimized pre-configured clusters that are ready for production workloads. It manages the control plane, nodes, scaling, placement, and all system components. Therefore, users can only focus on their workloads without additional cluster management knowledge. With an Autopilot GKE cluster, only the region (not the zone) could be selected. On the other hand, the **Standard** mode provides more flexibility and full control over managing and configuring the cluster depending on the required workload, this

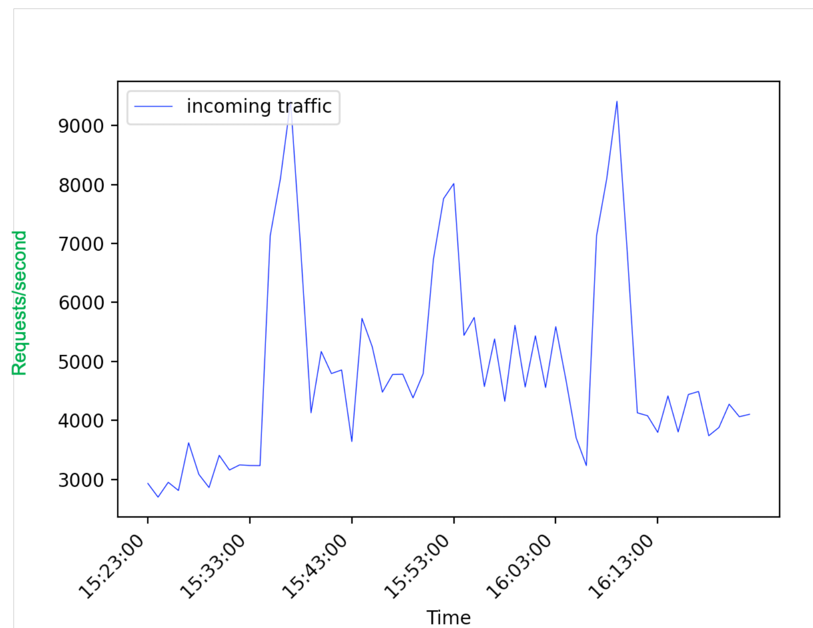

Figure 5.2: Bursty Traffic (the cluster re-sizing behavior was omitted due to its stability throughout the test since the traffic bursts occurred frequently)

mode manages the control plane and system components, while the user manages the nodes, their scaling, placement, etc. In the Standard mode, both the region and zone can be selected. Both GKE operation modes have been tested as baselines against our proposed Intent-Based CCDN solution.

Our tests of CCDN deployments have been carried out by creating a GKE cluster in `Europe_West` region for both baselines.

– **Baseline 1** is an **Autopilot** GKE regional cluster in `Europe_West` region

– **Baseline 2** is a **Standard** GKE zonal cluster in `London` zone within `Europe_West` region

In both baselines, the cache size considered is medium.

- **Traffic Generation:** The nature of the CDN requests rate is highly variable depending on several factors like the popularity of the content or seasonal popularity, network congestion, resources availability, etc. Therefore, we test the

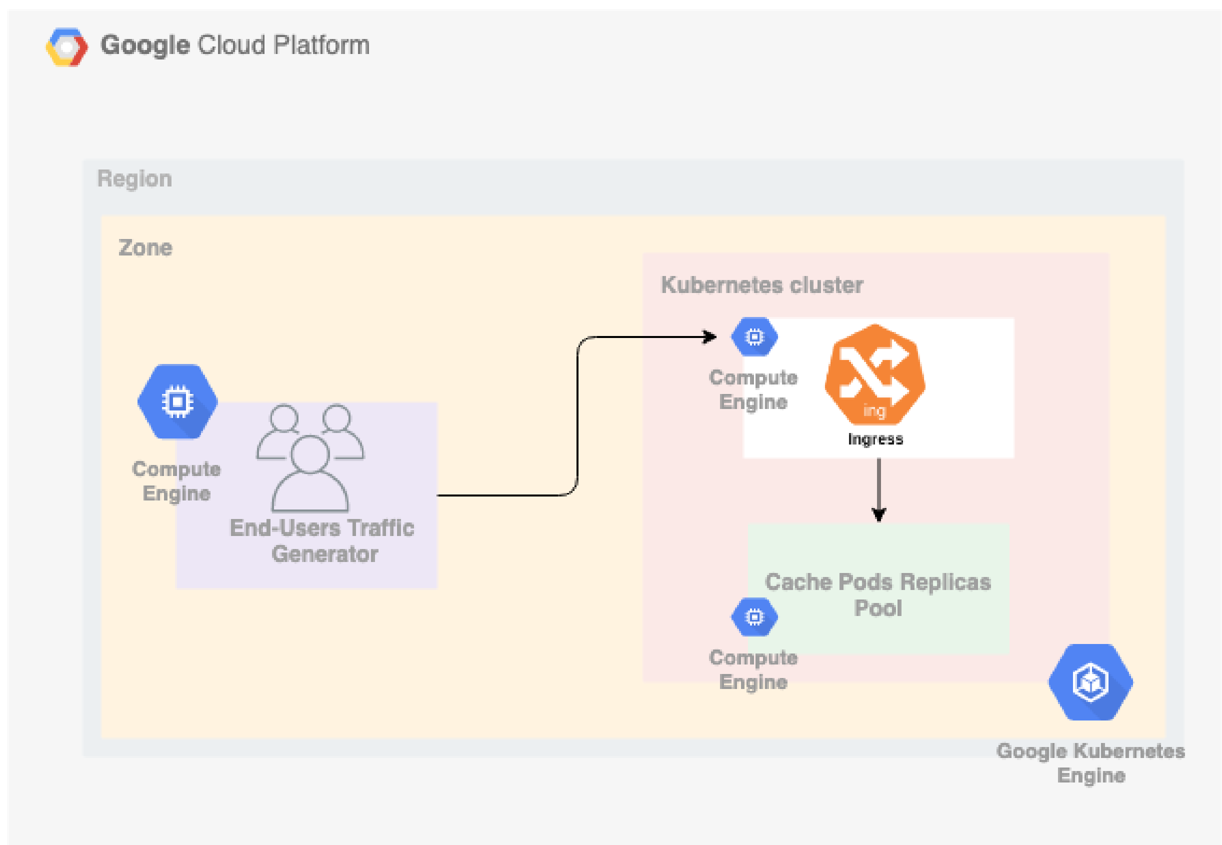

Figure 5.3: CCDN Experiment Setup on Google Cloud Platform.

traffic for *one week* in order to measure its *performance (in terms of dropped requests)* and *cost*. However, since generating and serving the whole traffic 24/7 via the GKE CCDN would be very costly, we look at the traffic in the dataset between 13:00 and 23:00 where the demand intensity was ranging from moderate to extreme. We focus on the *transitional traffic segments* which require scaling out the CCDN caches from *N* to *N+1* caches. This scale-out process leads to some possible dropped requests that would not be served due to the overhead on the current caches until the new cache is up and running. Hence, we run these specific segments of traffic with their corresponding number of caches that could meet the demand before the need to scale out, each test result is the *average of 10 different runs of the same traffic*. Additionally, we integrate *Automatic Horizontal Scaling (called the Kubernetes Horizontal Pod Autoscaling (HPA))*, which automatically scales out the caches by adding more of them when needed. So, we have been able to capture the dropped requests during that process. However, in our tests, we have not measured the possible increased delay due to cache overload since we mainly focused on capturing the dropped requests.

To generate end users' traffic requests, we create a VM that resides in the same region and zone to emulate the real traffic generated by actual CDN end users that are located in the same zone. For traffic generation, *Wrk2* (Tene, 2019), a popular HTTP requests generation tool has been used. Wrk2 allows sending HTTP requests to a specific URL at a certain request rate. This feature allows us to generate the same traffic pattern captured in the dataset used in these tests. We assumed that the user request size is 500 KB which is equivalent to a video chunk size as in this work (Frangoudis et al., 2017).

- **Metrics:** The main evaluation metrics that have been chosen are:

  Performance Ratio: measured in terms of a ratio (fraction) **inverse** for the number of dropped requests in a CCDN deployment over the baseline's drops as the fraction's denominator. Since the number of dropped requests affects performance negatively, we calculate the **inverse** of the ratio where the higher value, the better it is (meaning fewer drops).

  Cost Ratio: represents the ratio **inverse** for a CCDN's cost per test period over the baseline's cost. Since the cost increase is not favorable, we calculate the ratio **inverse** for the cost where the higher the value (i.e., less cost) the better.

  Performance-to-Cost Ratio: combines both previous metrics in a single ratio (i.e., Performance Ratio/Cost Ratio) which summarizes the overall gained performance for the money. The higher the value, the better it is.

5.1.3 CCDN Deployments Cost Calculation

To compare and calculate the corresponding cost per each CCDN deployment, we refer to the `E2 standard` machine cost in the `Europe-West` region provisioned by Google Cloud (Google, 2023g). The costs are listed in Table 5.1 and Table 5.2. Since the level of granularity for our tests was per minute, the cost/minute has been calculated to get the overall cost/day, which depends on the number and size of caches at each time instance. Both **baselines** provide medium-sized caches, which is equivalent to `e2-standard-4` machine instance cost. Nonetheless, **Baseline 1** is in Autopilot mode, and **Baseline 2** and other **LCIs** are in Standard mode, where both **Baselines** are placed in `(Europe-West)` region and **Baseline 2** is specifically placed in the local zone `(London Europe-West2)` and **LCI 1** is placed at a nearby zone `(Belgium Europe-West1)`. Contrarily, **LCI 2** and **LCI 3** CCDNs and their refinements provision small caches that are equivalent to the cost of `e2-standard-2` machine, with **LCI 2** being placed locally in `(London Europe-West2)`, whereas **LCI 3** is in `(Belgium Europe-West1)`.

	Cost ($) / hour	
E2 Standard Machine in local zone (London Europe-West2)	Autopilot mode	Standard mode
e2-standard-2 2 vCPU 8 GB RAM (Small cache)	0.12838	0.08633
e2-standard-4 4 vCPU 16 GB RAM (Medium cache)	0.25676	0.17267

Table 5.1: Compute Instances Costs in a Local Zone.

5.2 Intent Translation (CCDN Pre-Deployment Phase)

The main goal of this evaluation is to investigate the overhead of the intent translation process with respect to different numbers of microservice alternatives and evaluation criteria in the corresponding AHP graph. In specific, we measure the time needed to complete the intent translation, which includes the AHP calculation to prioritize all microservice alternatives, and the clustering process, which clusters all possible CCDN deployments that are enumerated by composing microservices of different types which

	Cost ($) / hour	
E2 Standard Machine in nearby zone (Belgium Europe-West1)	Autopilot mode	Standard mode
e2-standard-2 2 vCPU 8 GB RAM (Small cache)	0.12838	0.07371
e2-standard-4 4 vCPU 16 GB RAM (Medium cache)	0.25676	0.14743

Table 5.2: Compute Instances Costs in a Near-by Zone.

constitute the overall CCDN. Each CCDN deployment is composed of an instance of the following microservice types: *cache size, cache zone, and refinement* as based on our previous analysis in Section 3.4.3.

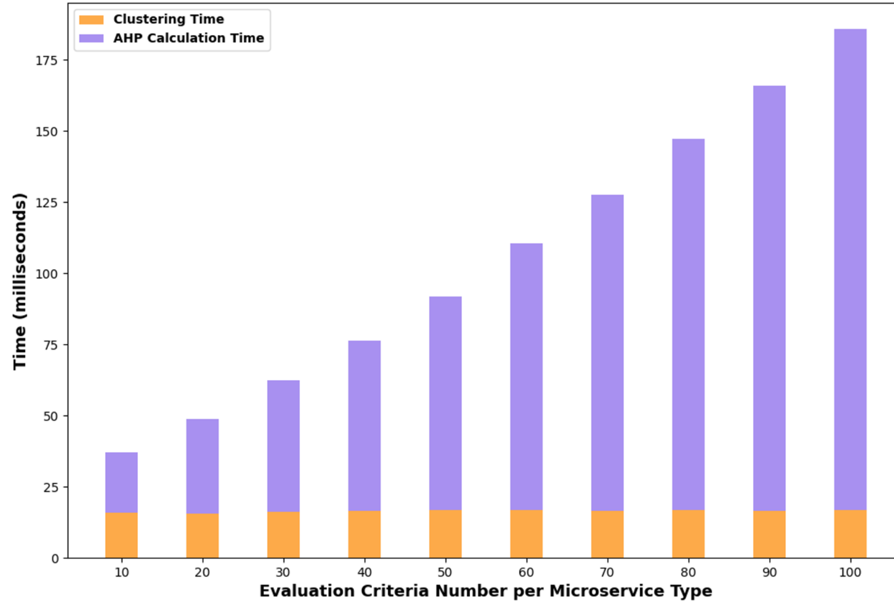

Figure 5.4: Intent Translation Time (AHP Calculation and Clustering) for Varying Number of Criteria.

5.2.1 Increasing Number of Criteria Results

First, we measure the time needed to complete the AHP calculation while increasing the number of evaluation criteria in the AHP graph that are used to evaluate microservices. We fix the number of microservice types to 2, each type has 10 alternatives. Therefore, we have a total of 100 (10×10) enumerated CCDN deployments to evaluate. We start with 10 evaluation criteria for each microservice type and increase that linearly all the way up to 100. Even though having these higher numbers of evaluation criteria is unlikely usual in practice, nonetheless we want to evaluate the translation process feasibility and scalability. In Figure 5.4, we notice 2 observations. The *first* is that even with 100 evaluation criteria, the time needed to complete the AHP calculation was approximately 175 milliseconds, which is very reasonable and the increase in the AHP calculation time is linear to the number of criteria increase. The *second* observation is that in all tests, the clustering time was the same which was under 20 milliseconds. The clustering process comes after the AHP calculation which assigns a score to each microservice based on the evaluation criteria. Therefore, the clustering operates based on these scores and enumerates all possible CCDN deployments, where each contains an instance of every different type of microservice that builds up the CCDN.

By combining the times of AHP calculation and clustering, at maximum, when we have 100 evaluation criteria and 10 different microservice alternatives per type to evaluate, the time needed is less than 190 milliseconds, which is notably better than the intent translation time achieved in a similar work (Scheid et al., 2017), where their MCDM process (i.e., SIG calculation) needed slightly more than 1 second to complete in the worst case, and around 3.5 seconds to complete the clustering process for a problem size similar to our biggest problem size.

5.2.2 Increasing Number of Microservices Results

On the other hand, we want to evaluate the translation time when increasing the number of alternatives for each microservice type to be evaluated and clustered. We set 10 evaluation criteria and vary the number of microservice alternatives to be evaluated from 10 to 100 linearly. Correspondingly, since we have 2 microservice types, the number of enumerated CCDN deployments to be evaluated ranged from 100 (10^2) all the way up to 10000 (100^2).

Opposed to varying criteria, microservices increase led to an exponential increase in the AHP calculation time. With 100 instances/microservice type to evaluate, AHP calculation needed approximately 260 milliseconds as shown in Figure 5.5. On the other hand, the clustering time showed a sublinear increase with the microservices number increase. With a smaller number of microservice alternatives, the clustering time is slightly more dominant as opposed to bigger numbers of

microservice alternatives (i.e., 40 and above) where the AHP calculation time becomes the dominant in time consumption. Therefore, for the biggest scenario with 100 alternatives/microservice type, 10 criteria, and 10K CCDN deployments to evaluate and cluster, the overall translation time was around 330 milliseconds to complete the AHP calculation and then cluster all CCDN deployments based on their overall microservice scores with respect to the intent target.

5.2.3 Discussion

First, by looking closely at the effect of increasing the number of AHP criteria with a fixed number of microservices to evaluate, in Figure 5.4, we justify the linear increase in AHP calculation time due to the $N \times N$ pair-wise comparison matrix computation, where N is the number of criteria. Each criterion would be compared against all other criteria in a matrix. (e.g., 100 criteria require 100×100 matrix for pair-wise comparison computation). On the other hand, the clustering time was the same which was under 20 milliseconds. This is due to the fixed number of microservice alternatives (i.e., 10 in our test) that would be used to enumerate all possible CCDN

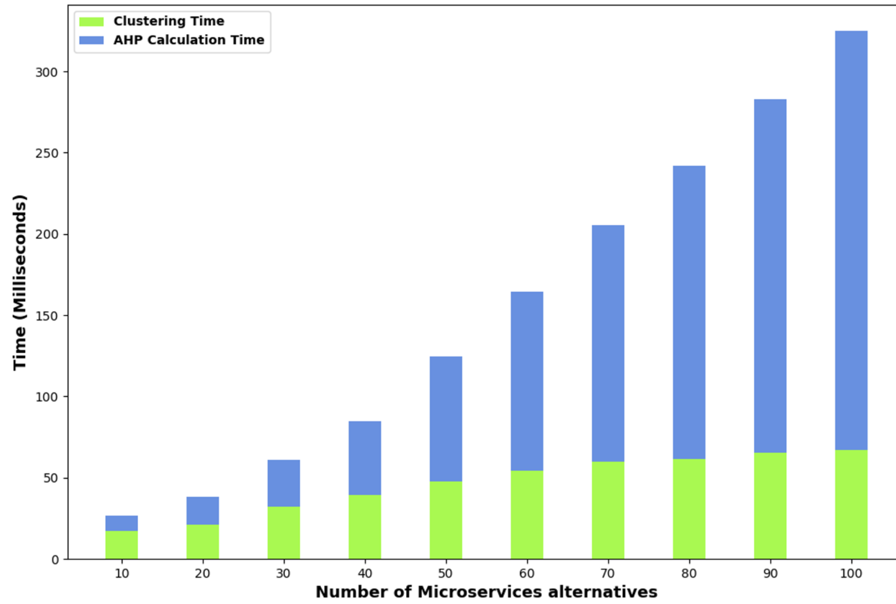

Figure 5.5: Intent Translation Time (AHP Calculation and Clustering) for Varying Number of Microservice Alternatives.

deployments from them. The total number of enumerated deployments that have been composed of 2 types of microservices, where each has 10 alternatives, was 100 deployments (10×10).

Secondly, moving to the effect of increasing the number of microservice alternatives with a fixed number of criteria to be evaluated, in Figure 5.5, we find that an exponential time increase is needed to compute AHP for the increasing number of microservices. This is due to the pair-wise comparison of all microservices against each other with respect to each evaluation criterion. Since the microservices are located at the last level of the AHP graph, their global score depends on all previous levels of criteria and sub-criteria score calculations in the graph. Unlike the pair-wise comparison of the criteria which are located in the middle of the AHP graph and their comparisons do not depend on the last level (i.e., microservices).

In conclusion, increasing the number of microservices affects the translation time more remarkably than increasing the number of criteria. However, in real-life scenarios, it is unfamiliar to have a large number of microservice alternatives of the same type. Nonetheless, in both previous evaluations, the overall translation time was significantly better than the closest work to ours (Scheid et al., 2017).

5.3 CCDN Post-Deployment Phase

In this section, we examine the performance, cost, and performance-to-cost ratios for our proposed Low-Cost Intents (LCIs) after executing them and refining them as well under different traffic cases.

5.3.1 Normal Traffic with Gradual Increase

As illustrated in Figure 5.1, the normal traffic pattern gradually increases until it reaches the maximum intensity during the peak hours. This was the dominant pattern in the used dataset that captured real CDN incoming requests. Usually, the peak hours require the highest number of cache scaleouts to accommodate the traffic of the intense requests, which could lead to a higher number of dropped requests during the scaling process due to the over-utilized current caches until the new caches are up and running. Hence, that is specifically when we are interested in the LCIs refinement that takes place to alleviate the potentially bad performance of the original LCIs which cared about cost reduction rather than performance. As discussed in Section 5.1.2, we capture the dropped requests during the scale-out process after running the generated traffic based on the real ISP CDN dataset, these traffic segments have been chosen based on the transitional periods that require scaling the caches. Therefore, within each day of the week, we run several traffic segments of which each requires a different scaling. We run these tests 10 times and record the number of dropped

requests, and calculate the average and standard deviation. Since we have several traffic segment tests throughout the day, we calculate the average standard deviation per day as shown in Equation 5.1 which is used to find the average standard deviation among k groups and each group has the same sample size (i.e., 10 repetitions) where $\mathbf{S_k}$**:** Standard deviation for k$^{\text{th}}$ group and **k:** Total number of groups.

$$AverageS.D. = \sqrt{\frac{S_1^2 + S_2^2 + ... + S_k^2}{k}} \tag{5.1}$$

5.3.1.1 Low-Cost Intents Performance Results

We proposed and discussed several LCIs and their refinements for the scenario of normal traffic in Chapter 4, Section 4.3.3. These refinements aim to improve the performance (in terms of reducing dropped requests) of the LCI. In other words, since the LCI would compromise the performance for cost reduction, hence, its refinement aims to reduce the number of dropped requests depending on the CCDN deployment. If the caches have not been placed locally within the requested zone, then it gets refined by spinning up local caches during the peak hours to reduce the effect of the increased start-up delay of the non-local caches. However, if the deployed cache size could be increased, then it is refined by vertically upgrading the cache size (e.g., from small to medium) during the peak hours to decrease the need for scaling out and hence, reduce the number of dropped requests in the process. Two types of intent refinements have been demonstrated, namely, **optimistic** and **pessimistic**. The latter aims at decreasing the drops even more than the former by performing the local cache placement and/or vertical upgrade *earlier* and for a *longer period*. For example, the optimistic refinement operates during the time window (18:00 – 22:00), whereas the pessimistic refinement time window (17:00 – 23:00).

$$PerformanceRatio = \frac{drops_{\text{baseline}}}{drops_{\text{dep}}} \tag{5.2}$$

In Figure 5.6, an overview of the dropped requests throughout a week is shown per day. The intent refinement results have been omitted from this figure for better result readability but they have been demonstrated fully in Figures 5.7 and 5.8. It is notable that the drops are significantly higher in the Autopilot GKE CCDN (Baseline1) compared to the Standard fully managed GKE CCDN (Baseline2), Even though both baselines contain the same cache size (medium) and number. We notice

that in Autopilot GKE, the new cache containers that get created during the scale-out process could crash or get a delayed scheduling more frequently as opposed to the Standard GKE, where we control the scale-out process within the same zone.

In **LCI 1**, The tradeoff for deploying less costly caches that are placed in a nearby zone (instead of the local zone) is the increased startup delay that could affect the performance negatively by increasing the possibility of request drops. When small and less costly caches in **LCI 2** are deployed in this CCDN (as opposed to medium caches in both baselines), this leads to more frequent scale-outs due to having smaller cache sizes (less costly than medium ones) that result in more drops and would eventually perform significantly worse than the baselines. As in **LCI 3**, combining both cost reduction factors in the previous intents, where small cheaper caches are deployed in a nearby zone. Naturally, this CCDN would suffer from the most droppings compared to the former ones in return for the highest cost reduction.

Therefore, the LCIs refinement takes place to alleviate the bad performance of the original intents. Each intent correspondent CCDN would be refined differently.

Figure 5.7 and Figure 5.8 compare the total dropped requests in a week for the CCDN that correspond to the LCIs, and their refinements that improve the performance by reducing the drops in contrast to the drops in both baseline CCDNs. Moreover, we also list the ***performance ratio*** score which denotes the performance ratio for each CCDN deployment compared to the baselines in Table 5.4. The performance ratio refers to the ratio of the dropped requests in our case, the fewer drops, the better. Therefore, we calculate the **inverse** of the ratio as follows:

Starting with the comparison against the Autopilot GKE CCDN deployment (**Baseline 1**), we can see that the performance has been improved 3.6 times with **LCI 1** CCDN. Even though **LCI 1** was expected to reduce the cost in return for the performance downgrade. However, in comparison to **Baseline 1**, it outperformed it considerably. Both **LCI 1 optimistic** and **pessimistic** refinements have improved the performance even further by 4.8 and 4.9 times respectively. This improvement has been achieved by reducing the cache startup delay during the peak hours, which is done by local cache placement within the local zone, as opposed to the nearby less costly caches that have been used throughout the day except for the peak hours. This refinement comes with an additional cost that will be discussed later in Section 5.3.1.3. The **pessimistic** refinement was slightly better than the **optimistic** as expected because it performs the local placement for a longer period compared to the latter, which helps reduce the droppings for more time.

By looking at **LCI 2**, in contrary to **LCI 1**, we can see that the performance has been reduced by 30% compared to **Baseline 1**. This performance downgrade is attributed to the use of small caches instead of medium ones, which leads to more frequent scale-outs, and accordingly losing more requests during this scaling process until the newly added caches are up and running. However, this performance

Performance Ratio to	Startup Delay			Cache Size			Startup Delay & Cache Size		
	LCI 1	Optimistic Refined LCI 1	Pessimistic Refined LCI 1	LCI 2	Optimistic Refined LCI 2	Pessimistic Refined LCI 2	LCI 3	Optimistic Refined LCI 3	Pessimistic Refined LCI 3
Autopilot GKE CCDN (Baseline 1)	3.6	4.8	4.9	0.7	3.3	3.8	0.6	2.4	2.8
Fully Managed GKE CCDN (Baseline 2)	0.7	0.98	0.98	0.1	0.7	0.8	0.1	0.5	0.6

Table 5.4: Performance Ratio to Baselines (higher is better)

decline has been reversed after the intent refinement which vertically upgrades the cache size from small to medium during the peak hours. The **optimistic** refinement outperformed **Baseline 1** by 3.3 times, whereas the **pessimistic** refinement that performs the vertical upgrade earlier and for a longer period performed even better with 3.8 times fewer drops.

In order to combine the cost reduction intent techniques used in **LCI 1** and **LCI 2**, we test **LCI 3**, which uses small and cheaper caches that are placed in a nearby zone. As expected, it has the lowest performance compared to the other LCIs. It reduced the performance of **Baseline 1** by 40%. Fortunately, this has been overcome with the intent refinements that use local and medium caches during the peak hours, with 2.4 and 2.8 times less droppings for **optimistic** and **pessimistic** refinements respectively.

Moving to the comparison against the Standard fully-managed GKE CCDN deployment (**Baseline 2**), It is observed in Figure 5.8 that the total request drops are considerably less than in **Baseline 1**, as it resulted in a total of 4K drops compared to 22K drops in **Baseline 1**. In fact, **Baseline 2** outperformed **Baseline 1** by 4.9 times more as shown in Figure 5.9.

The **LCI 1** trades off the performance with the cost reduction of cheaper caches that run in a nearby zone as opposed to the local caches in **Baseline 2**. As shown in Table 5.4, it performed worse compared to **Baseline 2** than to **Baseline 1**. The drops

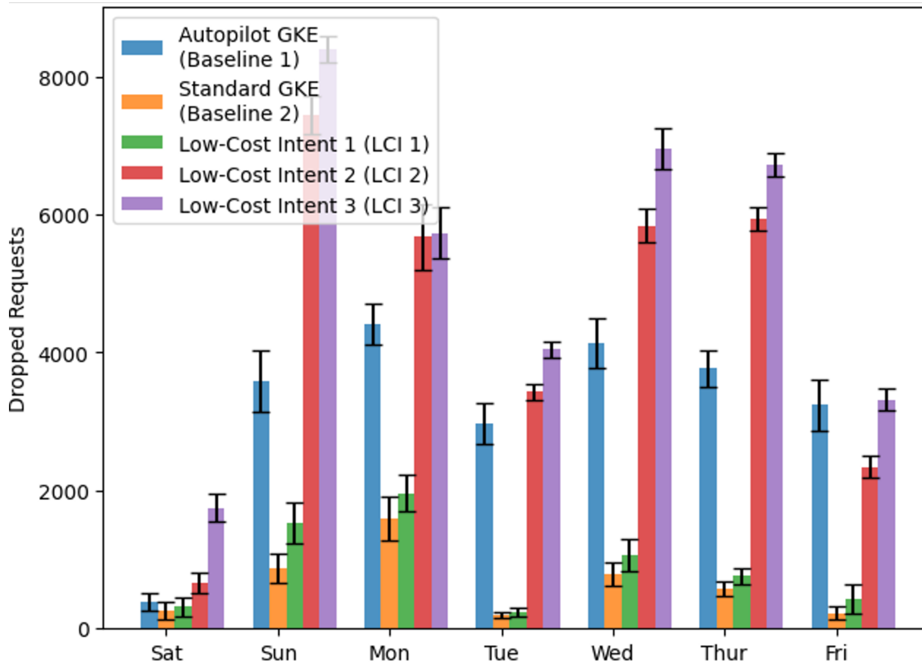

Figure 5.6: Average Number of Dropped Requests in a Week.

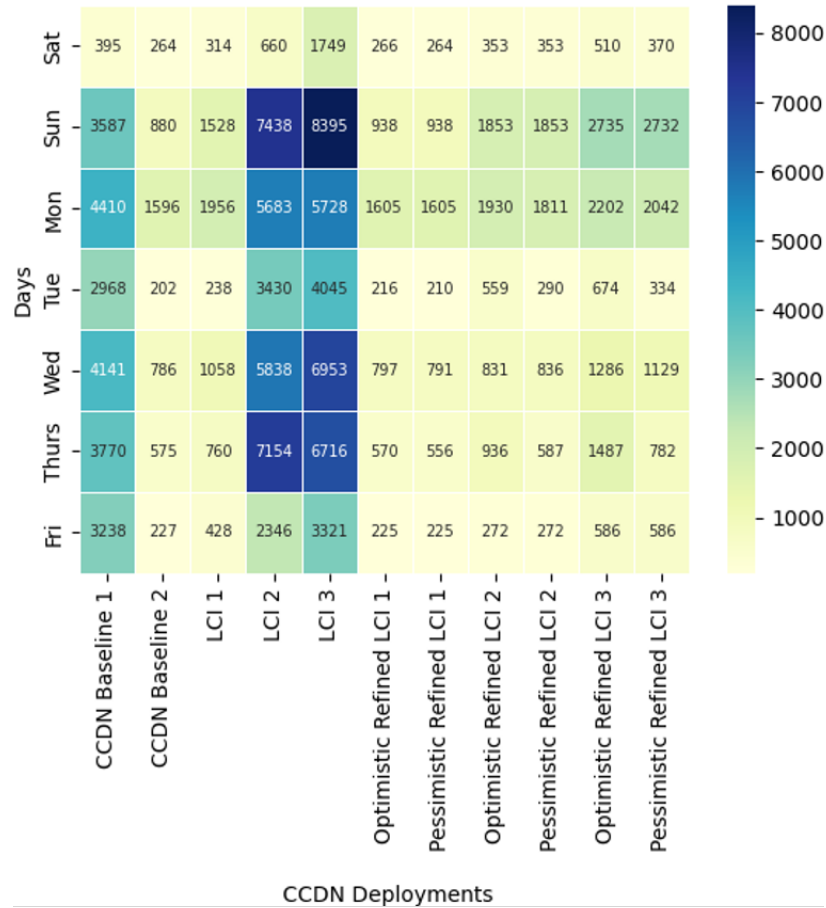

Figure 5.7: Dropped Requests Throughout a Week.

increased by 30% compared to **Baseline 2**. However, this performance degradation has been positively mitigated with intent refinement, to rise up to 98% of **Baseline 2**'s performance with both the **optimistic** and the **pessimistic** refinements.

Contrarily, **LCI 2** achieved only 10% of **Baseline 2**'s performance due to the significant increase in droppings that occurred during the frequent scale-outs of small caches. Advantageously, the intent refinements improved the performance up to 70% and 80% of **Baseline 2**'s performance, by vertically upgrading the small caches to medium during the peak hours.

On the other hand, **LCI 3**, had a similar performance to **LCI 2**, which reduces the performance of **Baseline 2** by 90%. However, the gained performance improvement by the intent refinement scored a lower performance ratio compared to **LCI 2**. It improved the performance to reach 50% and 60% of **Baseline 2**'s performance. The performance of **LCI 3** and its refinements resulted in the lowest performance ratio

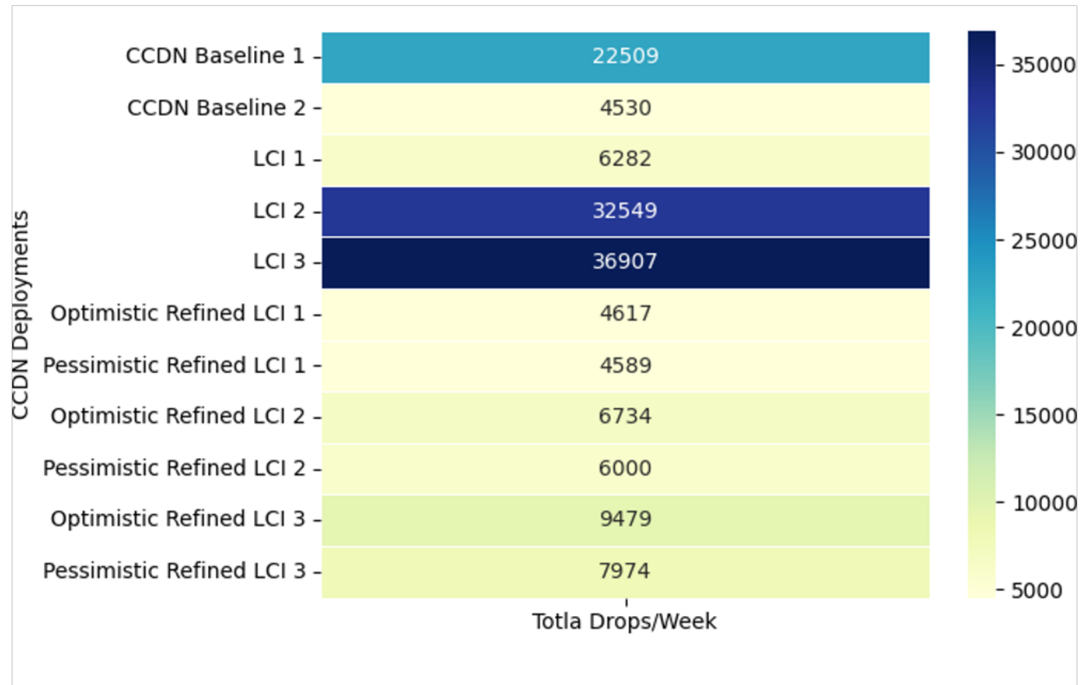

Figure 5.8: Total Dropped Requests in a Week.

scores compared to its counterparts **LCI 1** and **LCI 2**.

5.3.1.2 Discussion

By dissecting the dropped requests for each day in a selected week as shown in Figure 5.6, 5.7 and 5.8 we can observe that all CCDNS had almost consistent behavior regardless of the traffic intensity, which helped with evaluating the effectiveness of our proposed LCIs and their refinements.

We think that the reason behind the considerable difference in performance between **Baseline 1** and **Baseline 2**, is that **Baseline 1** runs in the GKE Autopilot mode that operates on a regional level. The region of the Kubernetes cluster has to be determined by the CCDN operator. This regional cluster runs multiple replicas of the control plane in multiple zones within a given region. By default, each node pool is replicated across three zones of the control plane's region. Hence, it is suited for high availability. Nonetheless, this comes with certain trade-offs. *First*, cluster configurations take longer because they must propagate across all control planes in a regional cluster. *Second*, users might not be able to create or upgrade regional clusters as often as zonal clusters. The nodes and containers can be placed within multiple zones inside that region. Thus, by default, the CCDN operator has no control over the exact placement of the cluster components since this is automatically managed

by Google (Google, 2023d). Hence, more cache container crashes and startup delays have been noticed during our test with **Baseline 1**.

On the other hand, **Baseline 2**, runs in the GKE Standard mode, which in our case, has been a zonal cluster. A CCDN operator has full control over all Kubernetes components and management within a specific zone (i.e., local to end users). Hence, the nodes and container placements are controlled on the level of zones rather than regions.

LCI 1 performed the best due to using medium caches compared to small caches in **LCI 2** and **LCI 3**. Therefore, we can infer that the ***cache size*** factor is more impactful on performance than the ***startup delay***. This is due to the frequent need to perform more scaling with smaller caches which leads to more dropping in the process until the new caches are up and running. Therefore, **LCI 2**, comes in second place in terms of performance which uses small local caches. Finally, **LCI 3** is the least performant due to combining both factors of cost reduction: ***smaller cache***

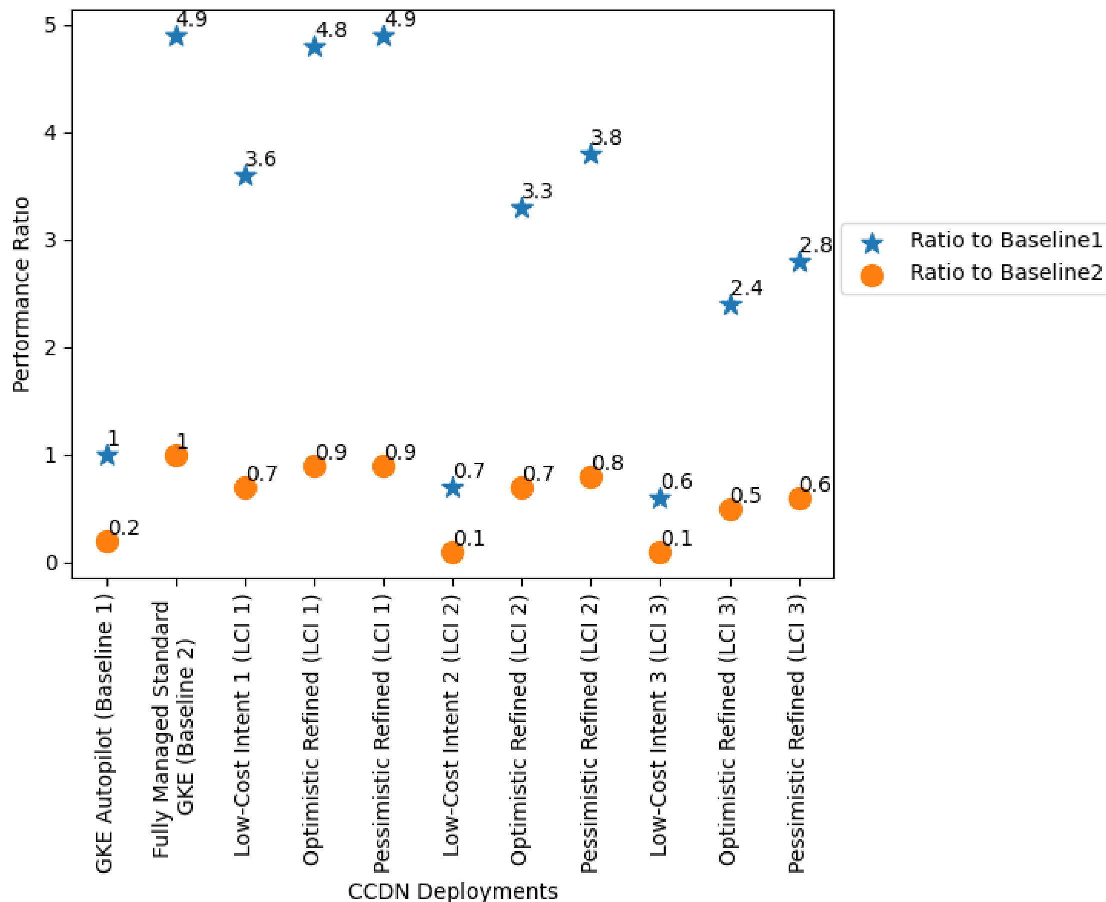

Figure 5.9: Performance Ratio Comparisons Against Baselines (higher is better).

size, and ***cheaper caches*** with more startup delay since they are located in a *nearby zone* rather than the local zone.

After intent refinements during peak hours, refined **LCI 1** with local cache placement (i.e., for startup delay reduction) was the best performing amongst the other refined intents. This is expected due to primarily using the more impactful factor that affects performance which is a bigger cache size compared to the others. Even though the refined **LCI 2** came in second place, interestingly, the vertical upgrade from small to medium caches caused a dramatic increase in the performance ratio compared to the original **LCI 2**. Again, this demonstrates the impact of bigger caches with less frequent scaling on decreasing the droppings. Finally, **LCI 3** came in last place. Although it utilized vertical upgrade for small caches, it still was affected by the additional startup delay that was the tradeoff for cheaper caches in a nearby zone. Therefore, the difference between refined **LCI 3** and **LCI 2** is less (which denotes the startup delay effect) compared to the difference between **LCI 2** and **LCI 1** (which denotes the cache size effect).

In comparison between **optimistic** and **pessimistic** refinements, the latter is expected to be better in performance due to the longer refinement period with a bigger time window that starts before and ends after the former. However, sometimes, this might not be the case. This mainly depends on the number of actions taken within the refinement time window in both refinements. In other words, if the refinement actions (i.e., caches scale out) needed to be taken only during the smaller time window for the **optimistic** refinement, then it would be no different from the **pessimistic** one since no additional actions were needed in that extra period.

In summary, amongst the proposed original LCIs, the one that used bigger-sized caches performed the best. And eventually, refined intents outperformed the original ones, and even dramatically increased the performance when the refinement depended on the vertical upgrade for the cache size. Moreover, the difference in the performance ratio score between **optimistic** and **pessimistic** refinements would be further improved by increasing the time window difference but with an additional cost. In general, these results emphasized the fact that the ***Cache Size*** factor has more impact on the performance in terms of reducing dropped requests compared to ***Start up Delay*** which is affected by the cache placement. Therefore, when designing solutions that aim at providing better performance in other problem domains, it would be useful to have this finding in consideration.

5.3.1.3 Low-Cost Intents Cost Results

Following our previous discussion of the CCDN deployments cost calculation in Section 5.1.3, in **LCI 1**, refinements calculation changes during the peak hours since local medium caches are deployed (`e2-standard-4` in `London Europe-West2`).

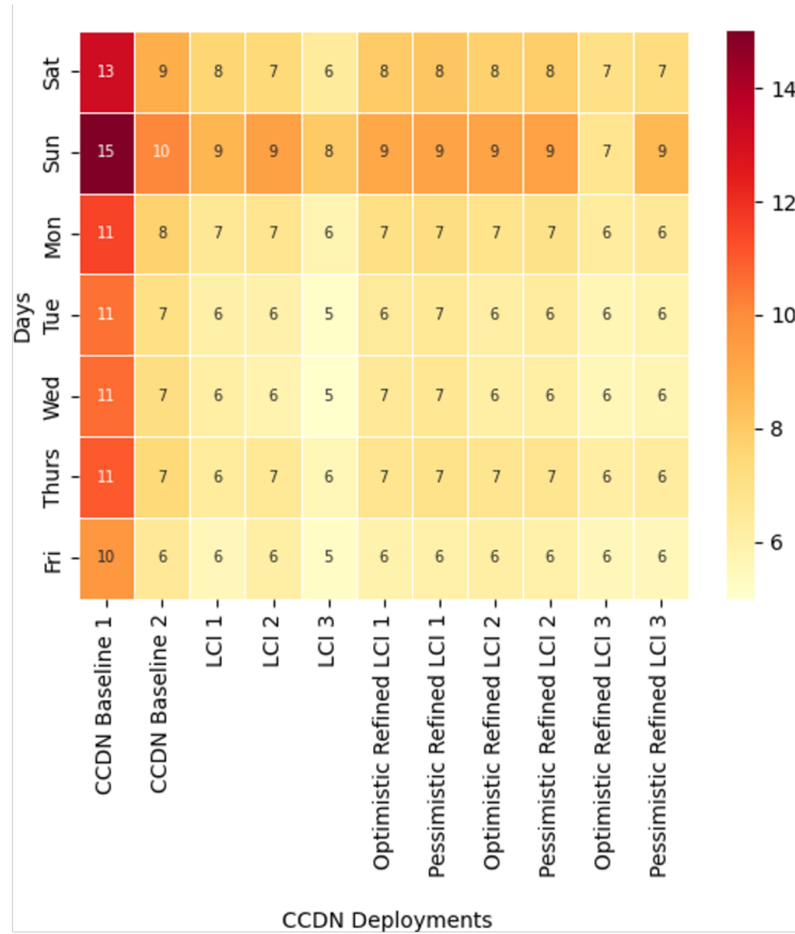

Figure 5.10: Cost($) Throughout a Week.

Whereas, in **LCI 2** refinement, local small caches are vertically upgraded to medium caches in the same local zone `(e2-standard-4 in London Europe-West2)`. Finally, in the refinement of **LCI 3**, vertical cache upgrade and local placement are considered in the cost calculation `(e2-standard-4` in `London Europe-West2)` during the peak hours. All intents and their refinements, and **Baseline 2** costs have been calculated based on the Standard GKE mode.

After calculating the total cost for a week of running the traffic extracted from the dataset, we can easily observe from Figure 5.10, Figure 5.11 and Table 5.6 that the **LCI 1**, **LCI 2**, and **LCI 3** costs are significantly reduced compared to **Baseline 1**, which is the main goal of these intents. In other words, we express this as a *cost ratio* that compares each CCDN deployment's cost against the baseline's cost. Surely, the less the cost is, the better. Therefore, we calculate the **inverse** of the ratio as follows:

$$CostRatio = \frac{cost_{\text{baseline}}}{cost_{\text{dep}}} \tag{5.3}$$

Accordingly, **LCI 1** has reduced the cost by 70% compared to **Baseline 1**. On the other hand, both its refinements (**optimistic** and **pessimistic**) required a small cost increase compared to the **LCI 1** original intent, which is considered to be very good compared to the considerable performance improvement (4.8 and 4.9 times better performance, respectively) discussed in the previous section. Both refinements cut the cost of **Baseline 1** to slightly more than half. **LCI 2** and its refinements all achieved 70% more cost reduction compared to **Baseline 1**. Whereas **LCI 3** was 2 times less costly than **Baseline 1**, and both its refinement reduced the cost 1.9 and 1.8 times respectively.

Since **Baseline 2** required noticeably less cost compared to **Baseline 1**, where it costs almost 50% less as expressed in the cost ratio Figure 5.12, it is expected to have less reduction degree between the LCIs and **Baseline 2** because all of the corresponding CCDN deployments have been fully managed (in contrast to **Baseline 1** Autopilot GKE CCDN). As listed in Table 5.6, **LCI 1**, provided 20% cost reduction, which is the result of selecting cheaper medium caches in a nearby zone. This has been decreased slightly to 10% with both intent refinements that shift to local more costly caches during the peak hours.

LCI 2, on the other side, cost slightly more than **LCI 1** by using small local caches, it reduced cost by 10% compared to **Baseline 2**. Both its refinements have not increased the cost and still remained 10% cost reduction.

Finally, **LCI 3** provided the highest cost reduction compared to **Baseline 2** with 30%. This is a result of combining the cost reduction solutions of the previous intents by selecting small caches in a nearby less costly zone. Its **optimistic** refinement provided almost the same cost reduction, as for the **pessimistic** refinement, slightly less cost reduction has been achieved due to the bigger time window of refinement, where local medium caches have been used.

Cost Ratio to	Startup Delay			Cache Size			Startup Delay & Cache Size		
	LCI 1	Optimistic Refined LCI 1	Pessimistic Refined LCI 1	LCI 2	Optimistic Refined LCI 2	Pessimistic Refined LCI 2	LCI 3	Optimistic Refined LCI 3	Pessimistic Refined LCI 3
Autopilot GKE CCDN (Baseline 1)	1.7	1.6	1.6	1.7	1.7	1.7	2.0	1.9	1.8
Fully Managed GKE CCDN (Baseline 2)	1.2	1.1	1.1	1.1	1.1	1.1	1.3	1.3	1.2

Table 5.6: Cost Ratio to Baselines (higher is better)

	GKE Autopilot (CCDN Baseline 1)	Fully Managed GKE (CCDN Baseline 2)	Low Cost Intent 1 (LCI 1)	Low Cost Intent 2 (LCI 2)	Low Cost Intent 3 (LCI 3)	Optimistic Refined LCI 1 (18:00 - 22:00)	Pessimistic Refined LCI 1 (17:00 - 23:00)	Optimistic Refined LCI 2 (18:00 - 22:00)	Pessimistic Refined LCI 2 (17:00 - 23:00)	Optimistic Refined LCI 3 (18:00 - 22:00)	Pessimistic Refined LCI 3 (17:00 - 23:00)
Ratio to Baseline 1	1	3.3	2.0	0.4	0.3	2.9	3.0	2.0	2.2	1.3	1.6
Ratio to Baseline 2	0.3	1	0.6	0.1	0.1	0.8	0.9	0.6	0.7	0.4	0.5

Table 5.7: Performance-to-Cost Ratio to Baselines (higher is better)

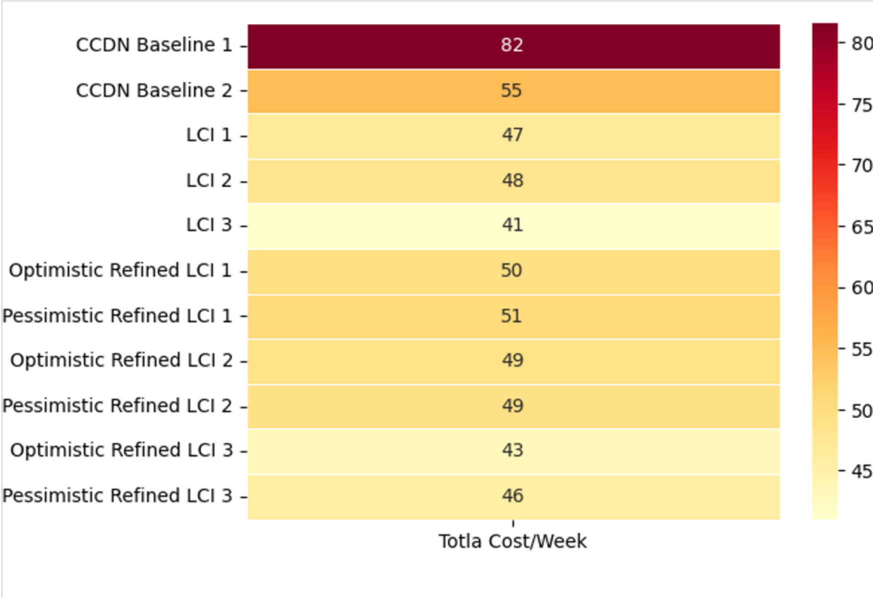

Figure 5.11: Cost($) in a Week.

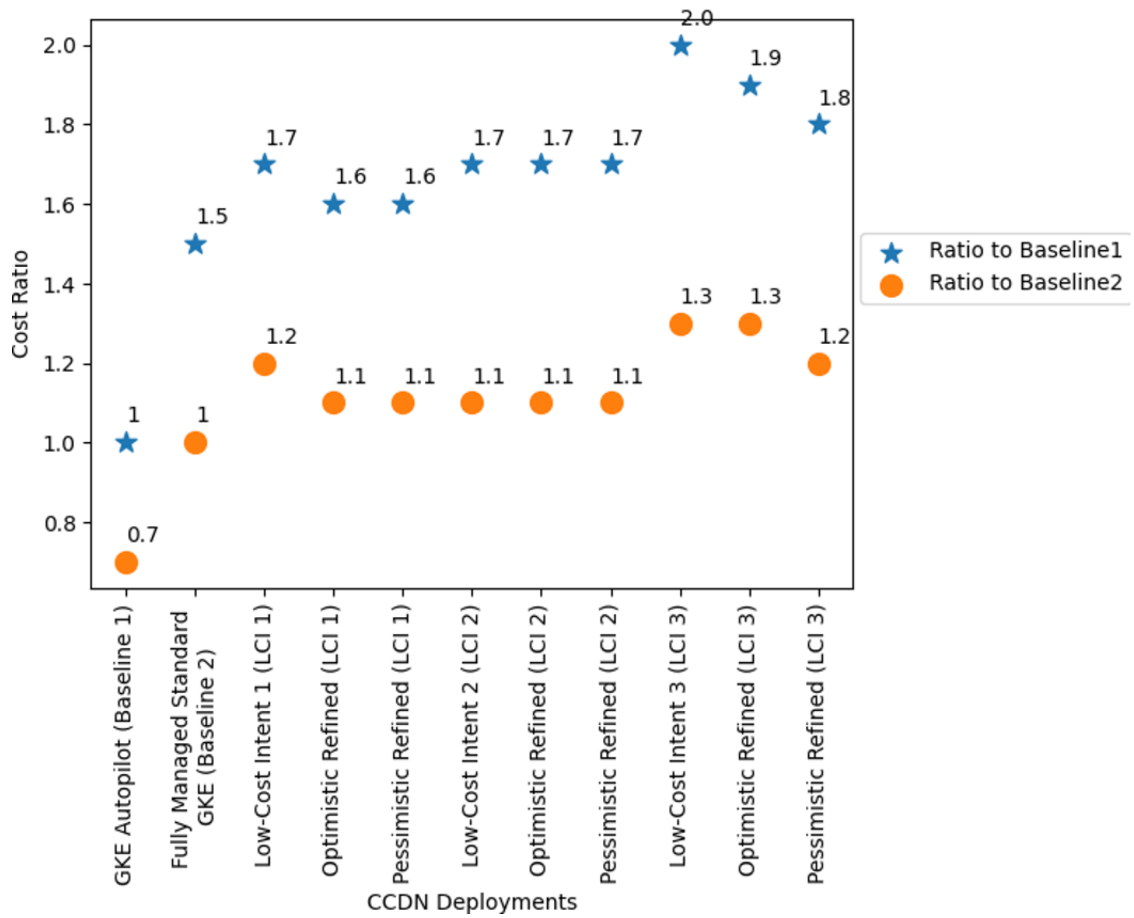

Figure 5.12: Cost Ratio Comparisons Against Baselines (higher is better).

5.3.1.4 Discussion

Baseline 1 was the costliest CCDN deployment according to Table 5.6 and Figure 5.11. Therefore, the cost reduction achieved by LCIs and their refinements was significant compared to **Baseline 1**. Ranging from 60% all the way up to 100% cost reductions. This is due to the Standard GKE cluster deployment in all of these cases which follows the standard compute instance pricing, as opposed to the Autopilot GKE cluster which follows the pricing based on the used resources like CPU, RAM, etc. The same exact resources have been used in all tested CCDN deployments. Therefore, the cost calculations have been made accordingly.

By narrowing down our focus on the cost comparison between **Baseline 2**, the LCIs, and their refinements, we can see that the costs are very close to each other with slight differences. They all followed the same pricing scheme based on the standard compute instance prices as listed in Table 5.1 and Table 5.2.

The reason behind the cost closeness or similarities between the LCI deployments and **Baseline 2** depends on the size and number of caches. For example, by referring to Table 5.1 and Table 5.2, ***medium*** caches (4 vCPU 16 GB RAM) have twice the capabilities and resources of ***small*** caches (2 vCPU 8 GB RAM). Hence at any point in time, to handle the traffic load, the required number of small caches is double the number of medium caches. Whereas the medium cache cost is double the small cache cost. Therefore, eventually, this leads to a very close overall cost. For instance, by comparing **optimistic** and **pessimistic** refinements, the bigger period of vertical upgrade (from small to medium caches) that the latter required, compared to the former, is met by almost twice the number of small caches which eventually summed up to a close cost value. Nonetheless, the proposed LCIs have reduced the cost by 10% to 30% depending on their type as discussed in the results. However, the dominant factor for cost variation is the cache placement since the caches in the nearby zone as shown in Table 5.2 are cheaper than the caches in the local zone as depicted in Table 5.1. In summary, these results emphasized the fact that the pricing scheme based on ***Cache Placement*** factor has more impact on the cost compared to smaller and less costly ***Cache Size*** where we need more of them which eventually compensates for the reduced cost. Therefore, when designing solutions that aim at providing less cost in other problem domains, it would be useful to have this finding in consideration.

5.3.1.5 Low-Cost Intents Performance-to-Cost Score Results

We discussed performance and cost results separately for all CCDN deployments. However, it is important to express an overall comparison that represents both as well. Therefore, we use the ***Performance-to-Cost ratio***, which summarizes the overall gained performance for the money where the higher score is better. It is calculated based on Equation 5.2 and Equation 5.3 as follows:

$$PerformanceToCostRatio = \frac{PerformanceRatio}{CostRatio} \tag{5.4}$$

Starting with **Baseline 1**, as shown in Table 5.7 we can outline that **LCI 1** and its refinements scored the highest ratio compared to the other intents. It achieved double the score of **Baseline 1** and even got increased by 2.9 and 3 times more with **optimistic** and **pessimistic** refinements respectively. **LCI 2** was the second scoring intent along with its refinements. However, **LCI 2** is drastically worse than **LCI 1** because it achieved only 40% of **Baseline 1**'s ratio score. Fortunately, this was notably improved by refinement to rise up to 2.0 and 2.2 times better score compared to **Baseline 1**. Finally, **LCI 3** and its refinements resulted in the worst overall scores.

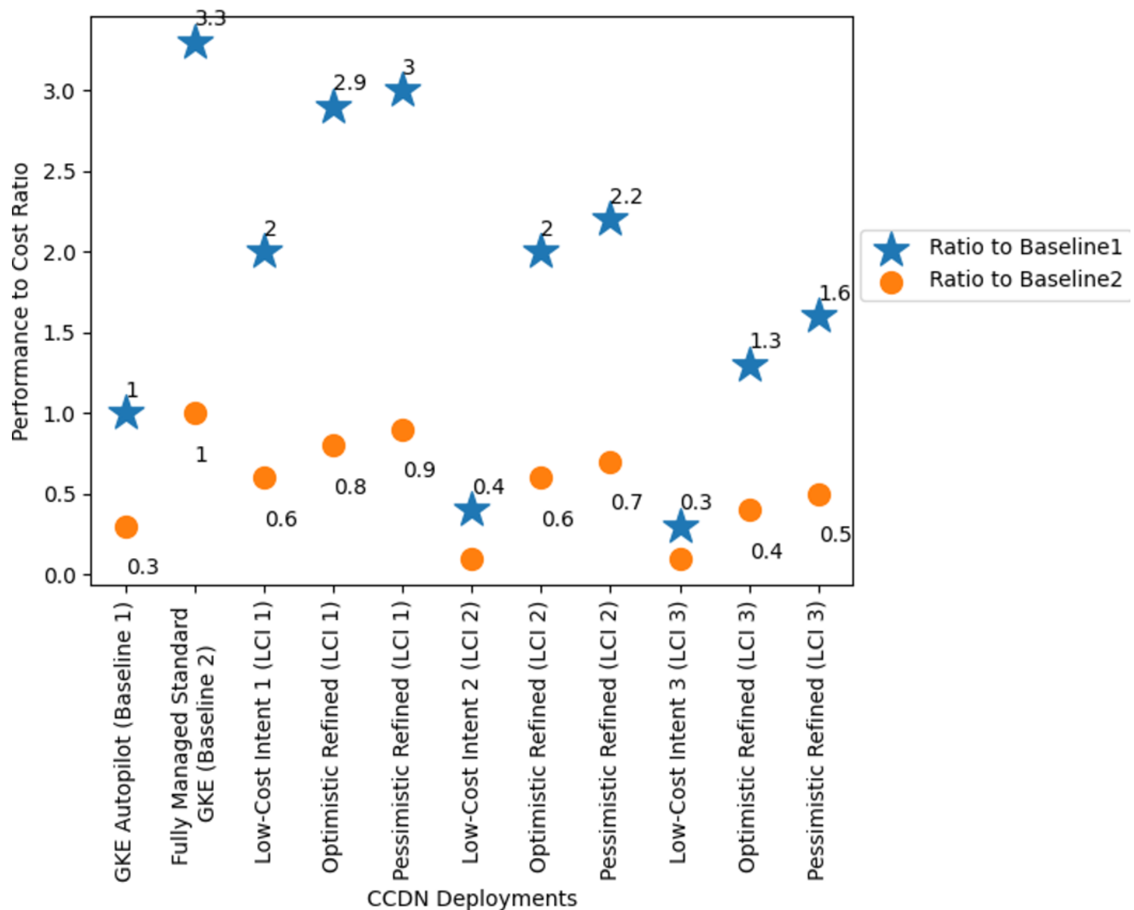

Figure 5.13: Performance-to-Cost Ratio Comparisons Against Baselines (higher is better).

LCI 3 only scored 30% compared to **Baseline 1**, which got improved after refinement to reach 30% and 60% better score than **Baseline 1**'s score.

Moving to **Baseline 2**, we can notice a similar reference pattern as shown in Table 5.7 and in Figure 5.13. **LCI 1** was the highest scoring intent that was able to reach 60% of **Baseline 2**'s score. However, this was further improved after refinement to reach 80% and 90% with **optimistic** and **pessimistic** refinements. **LCI 2** scored poorly, which was 90% less than **Baseline 2**. Fortunately, it got markedly improved to 60% and 70% after **optimistic** and **pessimistic** refinements. At last, **LCI 3** also scored poorly with a score of 90% less than **Baseline 2**. It was improved after refinement but to a smaller degree compared to **LCI 2**. After refining **LCI 3**, the corresponding scores are 40% and 50% compared to **Baseline 2**. The overall ratio score comparisons between all CCDN deployments are shown in Figure 5.13.

5.3.1.6 Discussion

The performance-to-cost ratio scores compared to **Baseline 1** are considerably higher than compared to **Baseline 2**. This is an expected result based on the performance and cost results individually and for the reasons discussed previously. However, the resulting general comparison behavior between the intents and their refinements against **Baseline 1**, is similar to the one against **Baseline 2**. **LCI 1**'s score even without refinement, is superior in comparison to **LCI 2** and **LCI 3**, and even pretty close to the **refined LCI 2**'s score. Moreover, after refining **LCI 1**, we got the best scores amongst all other CCDN deployments. This is due to the cache size factor that positively impacted the performance, where bigger caches have been used while on the other hand, requiring close or similar costs to other deployments with smaller caches. The original intents **LCI 2** and **LCI 3** scored really low, which necessitated the need for their refinements. Therefore, the refined **LCI 2** scores got improved to acceptable levels compared to **LCI 1**. Nonetheless, the refined **LCI 3** scores were the lowest, and are not preferred in high-intensity traffic scenarios.

5.3.2 Traffic with Bursts

In this tested scenario, we generate some traffic bursts and test them over a period of 1 hour, where we demonstrate the outcome of **LCI 2** compared to the **baselines**. We focus on LCI 2 refinement in this scenario since it depends on the more impactful factor (i.e., smaller cache size) on the performance degradation in the normal traffic behavior, and thus we wanted to investigate the scale of this degradation in a bursty traffic situation. Due to the sudden and fast nature of traffic bursts, the intent refinement should take this into consideration while performing the refinement. Therefore, we propose a different **LCI 2** refinement. Opposed to the previously proposed and tested **LCI 2** refinement which vertically upgrades small caches to medium in the

normal traffic scenario, it is important to opt for a faster refinement solution that could accommodate the sudden traffic burst quickly. So, we also compare this **burst-related LCI 2 refinement** to the **previous LCI 2 refinement** that we proposed and implemented in a normal traffic situation.

5.3.2.1 Low-Cost Intents Performance Results

Similar to the Performance calculation in the normal traffic case, it refers to the number of droppings and to calculate the performance ratio compared to the **baselines**, we use the same Equation 5.2. As depicted in Figure 5.14, we test **LCI 2** which aims at lowering the cost by using small local caches compared to the medium ones in both **baselines**. Both **baselines** are the same as previously discussed. Accordingly, **Baseline 1** is remarkably worse than **Baseline 2** as listed in Table 5.8. However, the **previous LCI 2** intent refinement which **vertically upgrades** the cache size from small to medium during the peak hours, would fall behind in the traffic bursts scenario as the vertical upgrade would not be able to keep up with the fast and rapid traffic increase during the burst. Therefore, we propose another **LCI 2** refinement during traffic bursts which aims at **speeding up the scaling** process of the existing small caches without the need to vertically upgrade them (which could possibly take a longer time and cause more drops). Hence, once a traffic burst is detected, then the Horizontal Pod Auto-scaling (HPA) threshold is lowered to allow a faster scale-out process. For example, the default HPA threshold is 80% average CPU utilization. When it gets exceeded then a new cache container is added. But in order to speed up the scaling process during traffic bursts, we lower this threshold to 65% CPU utilization.

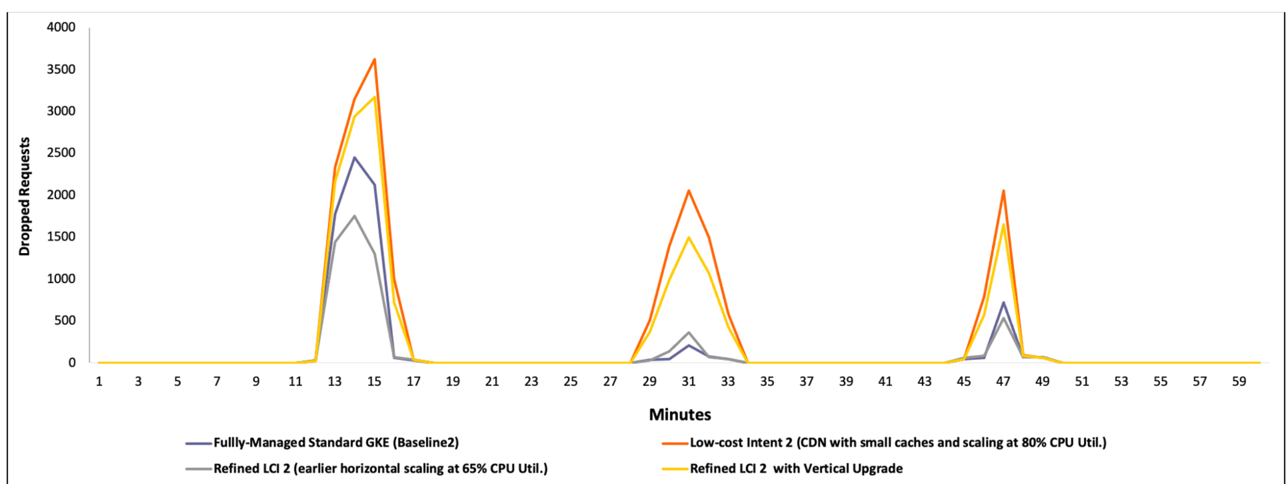

Figure 5.14: Dropped Requests During Traffic Bursts.

	CCDN Deployment	Baseline	LCI 2	Refined LCI 2 (Vertical Upgrade)	Refined LCI 2 (Earlier Scaling)
Autopilot GKE CCDN (Baseline 1)	Dropped Requests in an Hour	35634	19234	15814	6077
	Performance Ratio to Baseline1	1.0	1.9	2.3	5.9
Fully Managed GKE CCDN (Baseline 2)	Dropped Requests in an Hour	7835	19234	15814	6077
	Performance Ratio to Baseline2	1.0	0.4	0.5	1.3

Table 5.8: Performance Ratio to Baselines in a Traffic Bursts Scenario

Starting with **Baseline 1**, more than 35K of dropped requests occur during 1 hour with 3 traffic bursts within. As listed in Table 5.8. This was the highest number of droppings compared to all other CCDN deployments. On the other hand, the **LCI 2** caused more than 19K droppings which was the result of using small local caches. We can visibly see that the cache placement factor and potentially the synchronization between all replicas within the region in **Baseline 1** have a much bigger effect on the performance compared to the cache smaller size in **LCI 2**. In fact, **LCI 2** performed 1.9 times better than **Baseline 1**. By refining this intent as before with vertical upgrade once a burst is detected, we got an improvement with 2.3 times better performance. However, after testing our proposed new refinement for **LCI 2** in traffic bursts, it even outperformed the previous refinement with 5.9 times fewer drops compared to **Baseline 1**.

Moving to **Baseline 2** where medium caches have been deployed locally, as expected, the performance of **LCI 2** was significantly less by 60%. Interestingly, the previous refinement that vertically upgrades small caches to medium only improved the performance slightly by reaching 50% of the performance of **Baseline 2**. Therefore, our new proposed intent refinement in bursty traffic cases outperformed all other CCDN deployments. Since it scales out the small caches earlier than the other deployments, it was able to reach the lowest number of dropped requests. It even surpassed the performance of **Baseline 2** by 30%. The overall performance ratio comparisons between all CCDN deployments that handled the same bursty traffic are shown in Figure 5.15

5.3.2.2 Discussion

In a traffic burst situation, the number of dropped requests could increase dramatically compared to normal traffic. This is a natural result of the sudden surge of incoming requests that have not been handled in a timely manner since the CCDN deployment was not scaled enough to handle this amount of traffic yet. The huge difference between **Baseline 1** and **Baseline 2** is most likely due to the regional and zonal level of cache placement and other replication overhead reasons mentioned previously, but the difference is even augmented in the bursty traffic situation due to the rapid

need for scaling.

We have demonstrated the **LCI 2** performance in this case compared to the **baselines**. The reason for choosing this intent is to investigate how would the previously proposed **LCI 2** refinement perform in a bursty traffic situation. This refinement vertically upgrades the caches upon the burst detection.

We find that **LCI 2** was the worst performing which was expected due to the frequent scale-outs of small caches and especially with the traffic surge where a lot of requests have been dropped. Therefore, its refinement is very important. *First*, we find that the previously used refinement (with normal traffic) of vertical upgrade

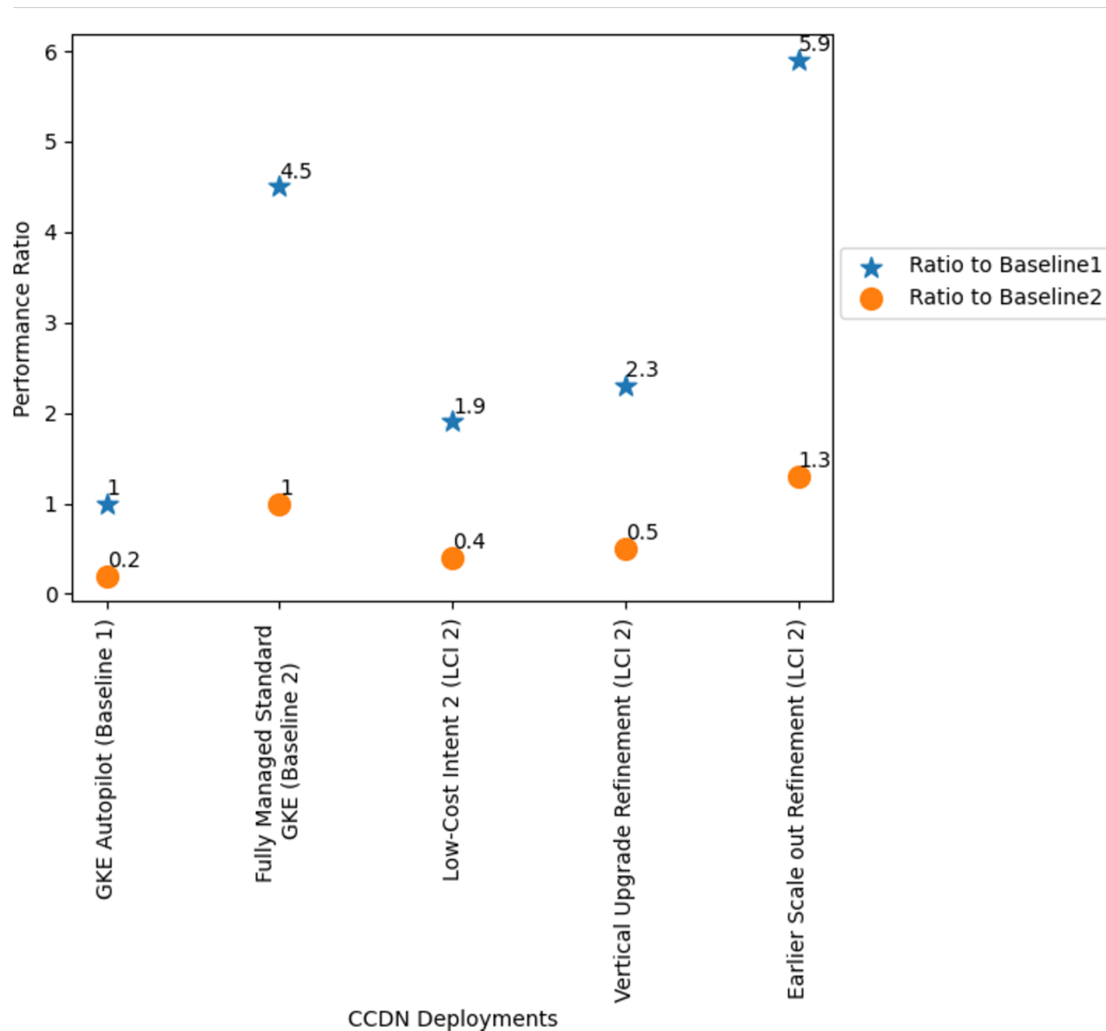

Figure 5.15: Performance Ratio Comparison Against Baselines During Traffic Bursts (higher is better).

	CCDN Deployment	Baseline	LCI 2	Refined LCI 2 (Vertical Upgrade)	Refined LCI 2 (Earlier Scaling)
Autopilot GKE (Baseline 1)	Cost per Hour ($)	1.0	0.5	0.6	0.7
	Cost Ratio to Baseline1	1.0	1.9	1.5	1.4
Fully Managed GKE (Baseline 2)	Cost per Hour ($)	0.7	0.5	0.6	0.7
	Cost Ratio to Baseline2	1.0	1.3	1.1	1.0

Table 5.9: Cost Ratio to Baselines in a Traffic Bursts Scenario

performed poorly, which was slightly better than **LCI 2** (the worst performance). This is due to the additional delay of spinning up the upgraded medium caches that could take longer in a critical time period of traffic bursts. Therefore, another refinement method was needed to deal with traffic bursts in a timely manner. So, we propose a different **LCI 2 refinement** for this case, where it scales out the small caches earlier than the other deployments, this has been achieved by lowering the scaling threshold while keeping the current cache size. This approach reduces the droppings since it spins up new caches even before the current ones get overwhelmed by requests. Interestingly, this refinement even outperformed the baselines by reducing the drops to the minimum.

5.3.2.3 Low-Cost Intents Cost Results

We followed the same cost calculation and Equation 5.3 discussed previously in Section 5.3.1.3 by referring to Table 5.1 and Table 5.2. During our 1 hour test that contained traffic bursts, **Baseline 1** had the highest cost ($1) as listed in Table 5.9. This was reduced by **LCI 2**, by using small local caches, which resulted in 1.9 times more cost reduction. However, since **LCI 2** needed to be refined to improve its performance, the previous refinement with a vertical upgrade from small caches to medium costed more than **LCI 2** but was still 1.5 times better than **Baseline 1**. As we mentioned for the bursty traffic situation, we propose another **LCI 2** refinement with faster scaling for small caches. This performed better than the previous refinement and in return, it cost a bit more. However, it was 1.4 times better than **Baseline 1**. By comparing CCDN deployments to **Baseline 2**, **LCI 2** was 30% less costly. After refining it through the vertical upgrade, the cost reduction became 10%. However, with our proposed intent refinement for traffic bursts, this cost reduction became less in return for improved performance. It costs equal to **Baseline 2**'s cost.

5.3.2.4 Discussion

For the same reasons discussed previously in Section 5.3.1.3, **Baseline 1** was the costliest CCDN deployment according to Table 5.9. Therefore, the cost reduction

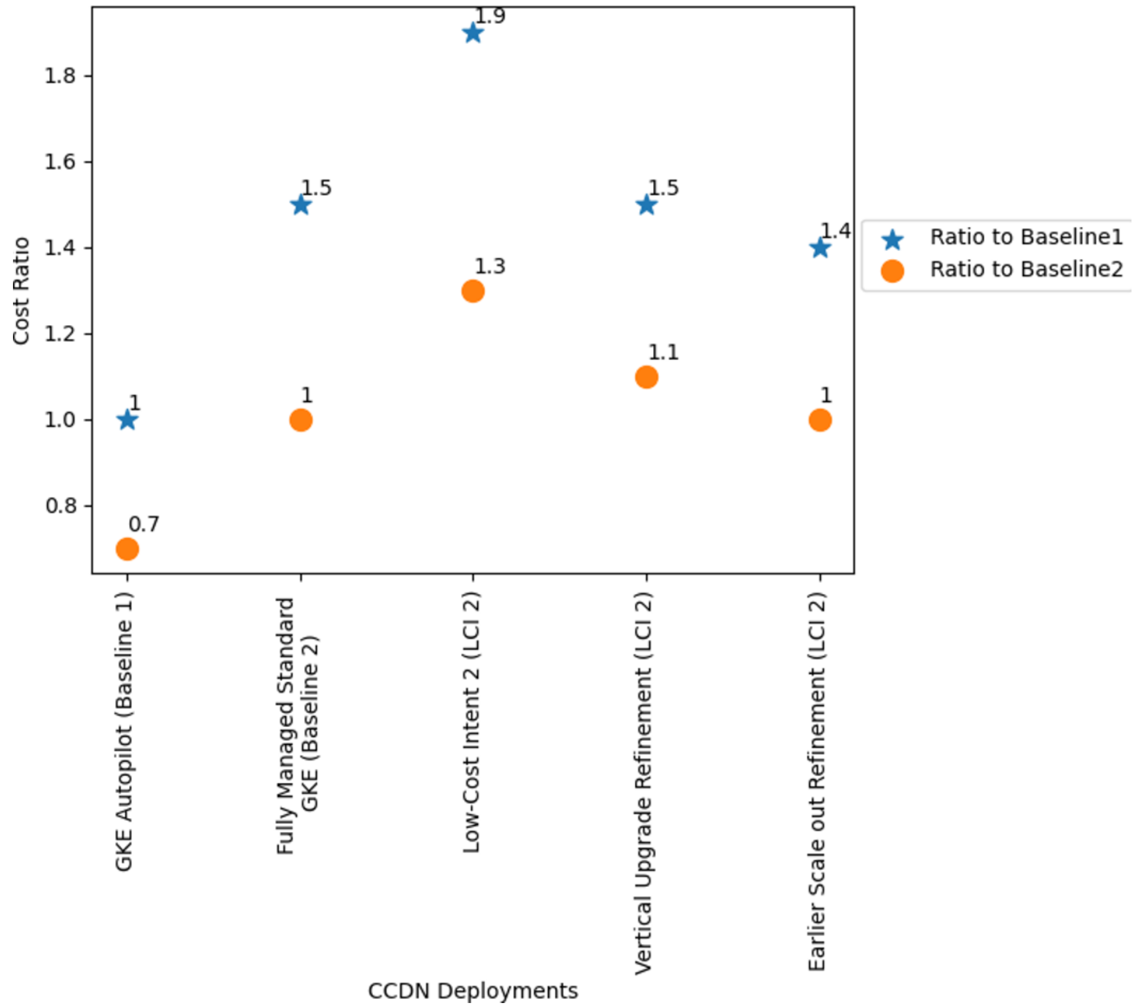

Figure 5.16: Cost Ratio Comparison Against Baselines During Traffic Bursts (higher is better).

achieved by **LCI 2** and its refinements was significant. The cost behavior of **Baseline 2** compared to other deployments is similar to **Baseline 1** but to a lower degree. **LCI 2**'s cost was the minimum as expected. On the other hand, the vertical upgrade refinement increased the cost due to the refinement process to improve the performance. However, it cost less than the newly proposed refinement with earlier scaling. This is because the latter spins up more caches due to the lowered scaling threshold before actually needing them as a preventive approach. However, it is important to note that these calculated costs were for 1 hour which included bursts of traffic. The cost of the rest of the day that does not include bursts would follow the

same results that we discussed before in normal traffic cases. Finally, the overview comparative cost ratio scores are demonstrated in Figure 5.16

5.3.2.5 Low-Cost Intents Performance-to-Cost Score Results

To compare CCDN deployments based on both performance and cost, we refer to the same Performance-to-Cost ratio Equation 5.4. This summarizes the overall score for each CCDN deployment. Compared to **Baseline 1**, **LCI 2** achieves 90% of its total score as listed in Table 5.10. Even though **LCI 2** aims to lower the cost as a tradeoff for reduced performance, yet, its overall score is almost very close to **Baseline 1**

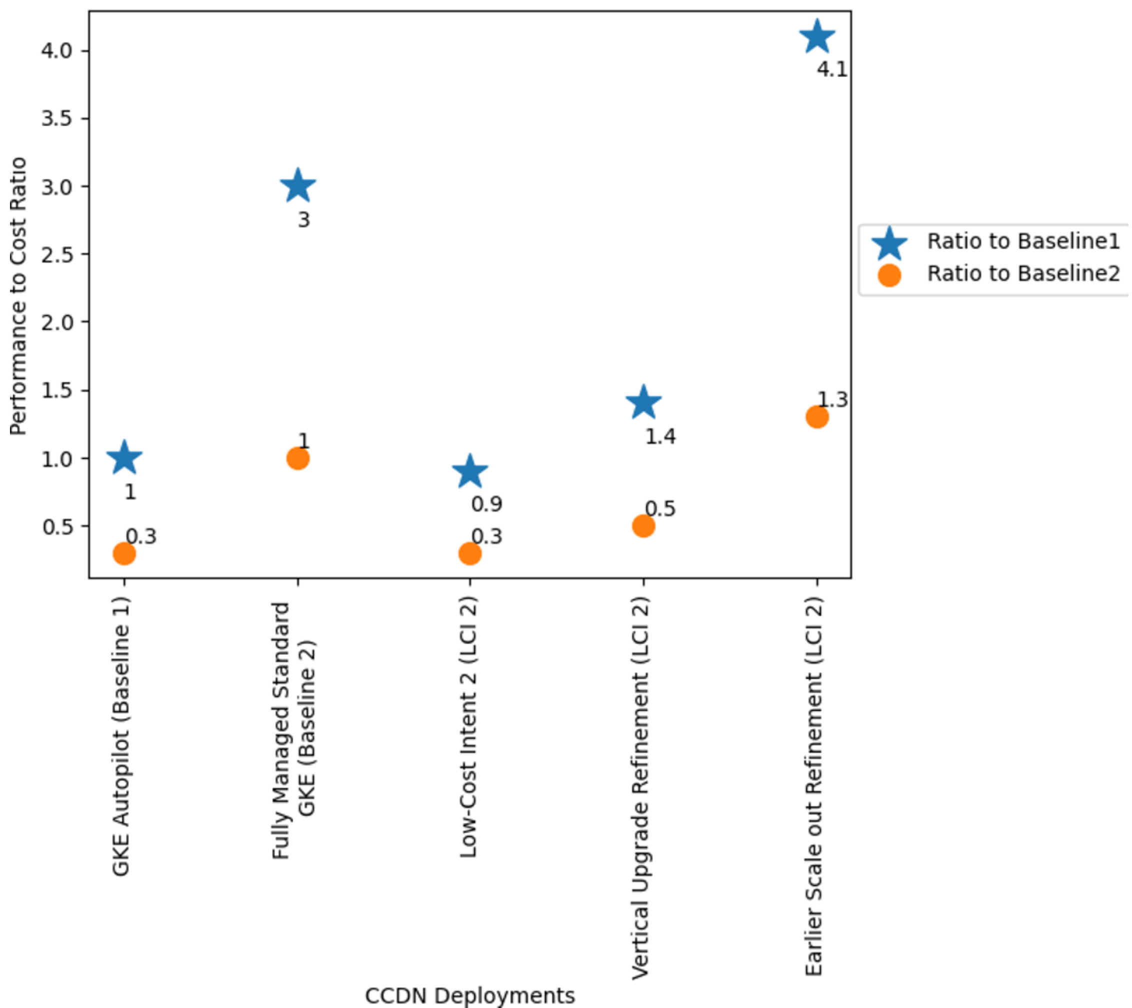

Figure 5.17: Performance-to-Cost Ratio Comparison Against Baselines During Traffic Bursts (higher is better).

CCDN Deployment	LCI 2	Refined LCI 2 (Vertical Upgrade)	Refined LCI 2 (Earlier Scaling)
Performance to Cost Ratio to Baseline 1	0.9	1.4	4.1
Performance to Cost Ratio to Baseline 2	0.3	0.5	1.3

Table 5.10: Performance-to-Cost Ratio to Baselines in a Traffic Bursts Scenario (higher is better)

since the latter's performance and cost were the worst compared to the others due to the lack of control over the placement of the caches. When **LCI 2** got refined by a vertical upgrade, the score rose up to 1.4 times better than **Baseline 1**. Finally, our proposed refinement for traffic bursts resulted in the highest score which is more than 4 times better than **Baseline 1**. On the other hand, **LCI 2** and its refinements follow a similar trend when compared to **Baseline 2**. **LCI 2** had the lowest score due to the considerably high drops which achieved 30% of **Baseline 2**'s score. After refining it with a vertical upgrade, the score slightly got improved to reach 50%. Fortunately, with our proposed intent refinement in bursty traffic cases, the overall score was the highest compared to **Baseline 2**, which even got exceeded by 30% less drops.

5.3.2.6 Discussion

In the case of traffic bursts, refining **LCI 2** with vertical cache upgrade does not perform well nor is cost-effective. Therefore, a quicker refinement is needed to keep up with the requests surge more efficiently. For that reason, we refined **LCI 2** differently, by lowering the horizontal scaling threshold to allow earlier scaling, where new caches are added before the existing ones even get overworked to reduce droppings. This threshold update is triggered by the traffic burst event depending on the CCDN definition of a burst (i.e., N continuous minutes of traffic rate above average with some percentage X%). After some stabilization time window, the horizontal scaling threshold goes back to the default value. An overview of ratio scores comparison is shown in Figure 5.17.

5.4 Summary

In this chapter, we evaluated our proposed CCDN Intent-Based solution in different CCDN deployment phases: Pre- and Post- deployment, and in different scenarios. These evaluations of specific aspects are essential to our design goals. For our proposed

CCDN to be a potential step towards next deployment phase of CCDNs, it should be able to allow CPs to express their intents, and try to achieve them throughout the lifetime of the service. Therefore, the adoption of microservices is key to this solution since it provides several benefits as mentioned in Chapter 2 Section 2.4. Accordingly, this paradigm's flexibility, scalability, agility, etc., offer a new approach to deploying and managing CCDNs with respect to CP intents. Since several available microservices could be used as alternatives for the same functionality, it is important to evaluate all possible microservice instances that belong to the same category (i.e., perform the same overall functionality) based on some evaluation criteria, that eventually meet the CP intent goal. So, thinking from a CCDN operator's perspective, they should be able to design, deploy and manage different CCDN alternatives with respect to the CP's intent, resources availability, CCDN operator's requirements, etc. Hence, a MCDM process is needed (i.e., AHP) to facilitate the decision-making during the CCDN Pre-deployment phase.

We examined the time needed to complete the intent translation with respect to the CP intent, and the defined AHP. We looked at 2 factors: the number of *evaluation criteria*, and the number of *microservice alternatives*. Even with higher numbers of each, the AHP computation time remarkably outperformed (Scheid et al., 2017), and same-wise with the CCDN deployments clustering into different possible ranked solutions. Having all CCDN deployments ranked offers more scalability, flexibility, and availability. This as well was completed in a considerably better time than (Scheid et al., 2017).

Once the CCDN gets deployed with respect to the selected deployment (i.e., intent translation output), another set of evaluations were needed in the CCDN Post-deployment phase. We proposed several Low-Cost intents that could be realized in different ways based on several factors (cache size, cache start-up delay, and both). These intents aim to reduce the cost. We evaluated the performance,cost, and performance-to-cost of each, in 2 scenarios: normal traffic (obtained from a major ISP dataset of captured CDN traffic), and bursty traffic, to demonstrate the difference in the intent behavior in each case.

As a general finding our results demonstrated the more impactful factors in terms of performance and cost. The former is mainly affected by the ***Cache Size*** more than the ***Start up Delay*** factor which is influenced by the cache placement. Whereas the cost is mainly affected by the pricing scheme corresponding to ***Cache Placement*** more than the different ***Cache Size*** pricing factor. Therefore, when designing solutions that aim at providing better performance or less cost in other problem domains, it would be useful to consider these findings.

In both traffic scenarios, we found that **LCI 1** which deployed *medium* caches but in a *nearby zone*, performed considerably better than **LCI 2** and **LCI 3**, where **LCI 2** deployed *small* caches in a *local zone*, and **LCI 3** deployed *small* caches in a

nearby zone. In fact, these intents that deployed small caches performed poorly and their refinement was very important to increase the performance to a good level in return for some compromised cost reduction. Therefore, we discussed different intent refinements for each of them. In a *normal traffic case*, the refinement takes place during the peak hours which handled the highest traffic intensity throughout the day. This period causes the highest droppings in requests; hence, refinement takes place during this period. We tested 2 levels of refinement: **optimistic** and **pessimistic**. The former used a smaller time window for the refinement, to potentially reduce the cost less than the latter. Since **LCI 1** already provided the highest performance without refinement, which was acceptable, the improvement after its refinement got even better to the point where it became almost as good as the Baseline that does not consider cost reduction which would negatively affect performance. However, the interesting parts of the refinement was the ones related to **LCI 2** and **LCI 3**. Their refinement caused a dramatic, and considerable performance improvement for each respectively. This is due to including the more dominant factor in performance, which is the *cache size*. It upgraded the *small* caches to become *medium-sized* during the peak hours. However, **LCI 3** was the worst since it ran all caches in a *nearby zone* (more startup delay) rather than the *local zone* as in **LCI 2**. Additionally, we calculated the cost for each CCDN deployment, **LCI 3** had the least cost compared to the others due to deploying *small* and *cheaper* caches that were located in a *nearby zone*. However, all three LCIs and their refinement cost reductions varied in the range between 60% - 100% less cost compared to **Baseline 1** and 10% - 30% less cost compared to **Baseline 2**.

On the other hand, since the traffic burst scenario has a different behavior compared to normal traffic, we also investigated **LCI 2** in that situation. Since it was expected to perform poorly as with the normal traffic case, we wanted to evaluate the level of its improvement after refinement. However, this scenario required handling a sudden surge of incoming requests with a considerably higher intensity than the previous traffic. Thus, we needed a different type of **LCI 2** refinement. So, we evaluated our newly proposed **LCI 2** refinement in a bursty traffic scenario which performs an *earlier* scaling of *small* caches rather than the *vertical upgrade* to *medium* ones in the previous refinement (in the normal traffic case). We compared both refinements to measure the effectiveness of both in this case. As expected, the previous refinement with *vertical upgrade* performed poorly and close to the worst performing CCDN deployment (**LCI 2**), this was due to the additional *startup delay* of cache size upgrade that took place during a critical time of traffic burst. Contrarily, our newly refined **LCI 2** which operates in a bursty traffic scenario, performed significantly better than all other deployments. In fact, it even surpassed the baseline's performance. This improvement was a result of the earlier cache scale-outs, which only occur during the traffic burst period and then go back to the default scaling after some

traffic stabilization time window.

As for the cost comparison during the bursty traffic situation, since it was throughout 1 hour that included 3 different traffic bursts, the calculated costs corresponded to that period. However, for the rest of the day which did not include bursts, we can refer to the cost calculation in the normal traffic case. Naturally, the non-refined **LCI 2** had the lowest cost which reduced **Baseline 2**'s cost by 90% and **Baseline 2**'s cost by 30%. But with our refinement which improved the poor performance of the original **LCI 2**, the cost reduction dropped to 40% and 0% correspondingly, which is a tradeoff to the performance considerable increase that even exceeded the baselines (which had the best performance previously before **LCI 2** refinement).

In both traffic scenarios, we compared the performance-to-cost ratio, which combines both metrics and provides an overall score for comparing all CCDN deployments. In terms of the original LCIs without refinement, **LCI 1** scored first, as **LCI 2** and **LCI 3** scored poorly. So, this led to the conclusion that **LCI 1** could be deployed without the urgency to refine it, unlike the 2 others that must be refined to improve their poor performances. On the other hand, all the refined intents scored high to highest scores, except for **LCI 3** refinement which was intermediate. However, **LCI 3** could possibly perform better in less stressful traffic demand cases. The technical translation and implementation of the intents and their refinements map to some conventional Kubernetes configurations. However, the specification of the corresponding operational parameters, thresholds, etc., have to be carefully well-defined and tuned based on the intent target and the problem domain requirements.

Although the results are based on the tested traffic scenarios and our technical interpretation of the abstract policies. However, our proposed abstract refinement policies (i.e., vertical upgrade and earlier horizontal scaling) could be utilized in other traffic cases but with careful consideration of the policy components' definitions. For example, in our corresponding refinement policies, the triggering policy ***conditions*** were either *peak-hours* or *traffic bursts*, but these could vary depending on the problem domain and the traffic patterns. The current ***actions*** taken when the conditions hold, are either, *scale-up*, *scale-out*, and/or *re-locate* caches. These actions are very commonly needed and could be easily utilized in other problems and use-cases but with more customized ***constraints*** that are problem-specific. Therefore, although our tested intent realization results are limited to the traffic patterns in the dataset and our tuned parameters, the abstract policies and the refinement algorithms could be handy in other cloud-based problem domains but with special consideration and tuning for the technical translation of the policy components that have to be problem-specific.

Together, these evaluations demonstrate the feasibility of moving towards an Intent-Based CCDN solution that leverages microservices. Moreover, evaluated LCIs

and their dynamic refinements showed different alternatives to CCDN deployments that could realize the CP's intent in different traffic behavior scenarios and reactively adapt to increased traffic demand. This is an important step towards understanding the future of CCDNs and their possible intent adoption.

Chapter 6

Conclusions

In this chapter, we summarize and discuss the outcomes of this work. We reflect on the research questions and highlight our contributions. Finally, we discuss future research.

6.1 Summary

Over the last couple of decades, CDNs have steadily grown to play roles of very high importance in the Internet ecosystem. This is due to their pervasiveness in several domains, their rising popularity owing to the evident increase in traffic that passes through them, and the added value they provide for both CPs and end-users. As usage patterns and end-user service expectations have changed over time, CDNs have become the predominant method used to deliver video and other content to a worldwide user base. To meet this demand, the ways in which content is delivered have also evolved, particularly for virtual and Cloud CDNs (CCDNs).

Today CCDN operators can leverage the current technical advancements in the field of Softwarization that changed system design by separating the software implementing network functions, protocols, and services from the hardware running them. Despite this promising paradigm, there are certain limitations that forestall the next generation of CCDNs, such as those addressed in this thesis. The current level of softwarized networks programmability still requires experienced programmers (i.e., network managers, admins, operators, IT personnel, etc.) who can orchestrate the services in accordance with different and conflicting customization requirements for the system and consumers (Tuncer et al., 2018; Alalmaei et al., 2020; Leivadeas et al., 2022). This imposed additional challenges by creating a cumbersome system configuration process to adjust to all different CDN stakeholders, users, and services (Leivadeas et al., 2022).

The thesis examined how the latest wave of technologies (particularly Intent-

Based Networking, Autonomic Network Management, Policy-Based-Management, and Microservices Architecture) may influence the future of CCDNs and their programmability. These technological advances can be found in other systems and problem domains, where new paradigms and services allow unprecedented control over the components of these systems. However, they have not been thoroughly discussed in the CCDN research field. This thesis investigated the adoption of an Intent-Based paradigm, not just as a replacement for conventional CCDN management, but also as a tool to create behavior and user interaction dynamics that are not often realized within traditional CCDNs.

Containerization technology is a form of softwarization that has enabled the creation of a new generation of flexible software functions (as microservices) and modular system architectures. These replace existing traditional monolithic systems and offer a realistic alternative to the hardware variants that already exist in today's systems since they promote portability, easier maintenance and updates, flexibility in terms of technology selection, cost-effectiveness, and separation of concerns owing to their modularity, and fault tolerance due to their distributed and decoupled design, which leads to higher availability.

Accordingly, several technological advances in CCDNs have improved overall CCDN management and even resulted in better end-user experience. However, to the best of our knowledge, there is insufficient research work in the CCDN domain that explores the adoption of some specific and recent technology trends, such as Intent-Based Networking and microservices. Investigating these technologies in a CCDN use case could possibly lead to taking a step forward towards CP and CCDN communication and interaction mechanisms that allow a generic CP to express a high-level intent target that could be achieved in several different ways through the evaluation and selection of different microservice alternatives that could lead to different intent outcomes.

This thesis testifies that by using a multi-state Intent-Based solution that utilizes the advantages and current advancements of the microservices architecture, a flexible, scalable, inter-operable, and more effective CCDN and CP communication mechanism could be built to open new markets by realizing new use cases for different service consumers in different domains that leverage the interaction and collaboration of several stakeholders. Such a solution includes utilizing the separation between the system behavior and the underlying realization via different microservice alternatives, the employment of an MCDM approach that helps with the decision making of the microservices selection with respect to the users intent, and an efficient design that is capable of proportionally refining intents according to the dynamic service and infrastructure status, with correspondence to the intent target throughout its entire life cycle.

6.2 Contributions

In Chapter 1, we discussed a specific motivation of technological importance and, accordingly, laid down a specific set of research questions to be answered. This guided our design and implementation of an Intent-Based CCDN. These are based on an understanding of existing CCDN design limitations (i.e., limited single-directional communication between CPs and CCDNs without considering CP's high-level targets in decision-making), as well as a forward-facing look towards the potential benefits that the utilization of new technologies and emerging architectures can bring to this area (i.e., intents and microservices).

Taking these considerations forward, we provide an Intent-Based design and implementation of such a CCDN framework. This is segmented into a number of layers, each of which is responsible for achieving a specific set of functions or behaviors. Importantly, it proposes an Intent-Based communication mechanism between CPs and CCDN operators, which translates the high-level CP intents into its corresponding CCDN deployment that could achieve the intent's target in an autonomic way and dynamically adapt to changes through its refinement during the life cycle of the intent. We tackle this process from the CCDN operator (intent developer and creator) point of view as opposed to other related works which present the intent translation as a black box with no systematic breakdown of the intent creation and translation processes.

To summarize, this thesis provides the following primary research contributions.

1. **A specification of Declarative High-level Intent Expression for CPs and Behavioral Prescriptive Policy Expression for CCDN Operators:** We considered the expressions syntax of both CPs (non-technical users) and CCDN operators (technical users). A declarative intent expression for CPs has been proposed to allow them to express their high-level target without the need for low-level or technical specifications, whereas a behavioral prescriptive policy expression has been proposed for CCDN Operators to allow them to regulate the CCDN behavior at an abstract level that could be realized in different ways according to the underlying technology (i.e., microservice alternative). These expressions have been influenced by the understanding of the limitations of existing intent/policy expressions that have been broken down in our corresponding meta-analysis in Chapter 2.

2. **Design for an Intent-Based CCDN Framework:** We designed an Intent-Based CCDN which has been achieved through our analysis of the limitations of the existing CCDN solutions in the literature, as well as integration of a variety of emerging technologies and trends. In particular, we utilized Microservices Architecture and Autonomic Networks. The former helps to achieve scalable, agile, flexible, and portable solutions, and the latter leads to less human involvement, cost

reduction, and self-managed adaptability. We addressed the design from a CCDN operator (intent creator/developer) perspective who is responsible of creating, planning and managing the intent translation from a high-level declarative syntax into some abstract behavioral policies (which could be stipulated by stakeholders), that get eventually technically realized via the underlying technology. In our context, the final intent translation outcome is a CCDN deployment which is composed of different microservice alternatives that collectively form the highest-scoring CCDN deployment that could handle the intent target. We provided a comprehensive design, which aimed to contextualize and encompass these features and elements in Chapter 3.

3. **A proof-of-concept implementation for the Intent-Based CCDN Framework design:** We built a a proof-of-concept implementation to evaluate and examine the effectiveness of our design. This followed the specification of the aforementioned design and formed the basis to evaluate the integration of intents in the CCDN scenario. This implementation also helped demonstrate the benefits of leveraging microservices advancements which could provide different microservice alternatives for the same CCDN functionality. This allows CCDN operators and stakeholders in general to evaluate and alternate between them according to the intent target, resources availability, system status, etc. We built our CCDN using Google Kubernetes Engine (GKE) platform to create and manage the CCDN cluster on the Google Cloud and manage it via Kubernetes as discussed in Chapter 4. Excluding the Kubernetes setup and configuration learning time. The implementation was completed within around three months, including the repetitive traffic generation at different times to get their overall average.

4. **An evaluation of the intent translation overhead and feasibility:** Through the use of the aforementioned proof-of-concept implementation, we measured the overhead (in terms of delay) of the translation of high-level intent targets to lower-level commands that deploy the corresponding CCDN. Since microservices allow a myriad of different realizations for the CP intent, it is important to follow a structured MCDM approach. We chose the popular AHP that leads to the best available CCDN deployment selection during run-time based on the CP's intent, evaluation criteria defined by the CCDN operator, and resource availability. Next, after the AHP calculation prioritizes all microservices with respect to the intent target, we then enumerated all possible CCDN deployments formed from these ranked microservices and clustered them into 3 different levels that denote their capability of achieving the intent target. We think that having these levels (i.e., high, medium, and low) is sufficient in our context, but this could be easily adjusted in the clustering process if more levels are needed in other cases. In specific, we evaluated the feasibility of the translation process with increasing problem sizes with

different dimensions (i.e., number of microservices, number of evaluation criteria). We scaled the number of evaluation criteria and the microservice alternatives up to 100 for each. In both cases, the translation time was very acceptable which was below 330 Milliseconds for the worst-case scenario with the biggest number of microservice alternatives and 10K CCDN deployments to enumerate and evaluate as discussed in Chapter 5.

5. **An implementation, and performance and cost tradeoffs evaluation of different Low-Cost Intent realization alternatives and their refinements:** Based on the proof-of-concept implementation for our Intent-Based CCDN design, we created and implemented several Low-Cost Intent realization alternatives that aim at provisioning CPs with lower-cost CCDNs by leveraging different microservice alternatives. We evaluated and compared their performance (in terms of dropped requests) and cost tradeoffs. However, as expected, the Low-Cost intent favors cost reduction over performance. Therefore, we proposed, implemented, and evaluated different Low-Cost intent refinements that aimed to mitigate the performance degradation (i.e., reducing dropped requests) that accompanied the original Low-Cost intents. Since this could vary according to different traffic situations, we tested these processes in different traffic patterns and suggested, implemented, and discussed different refinements accordingly. Our implemented CCDN deployment via the GKE demonstrated how the Kubernetes orchestrator could help with the autonomic Low-Cost intent refinements in different traffic cases, where we discussed the mapping of the components of our proposed framework to the IBM's MAPE-K loop of an autonomic system in Chapter 3. In our evaluations in Chapter 5, we compared all of the Low-Cost intent alternatives and their refinements against 2 GKE baseline clusters, namely, Autopilot GKE (Baseline 1) and Standard Fully-managed GKE (Baseline 2). Both baselines do not support intents and thus could resemble normal CCDNs that do not aim to reduce the cost.

The contributions listed above are a vital step towards exploring Intent-Based CCDNs. Our evaluations exercise a number of features and explore some interesting CP intent targets that could be furtherly extended with different other alternatives that could achieve the target.

At this point, we contributed towards answering **RQ1** in Chapter 2, by discussing several limitations of the current technologies that CCDN operators could leverage today. Moreover, we provided a meta-analysis for some significant current Intent-Based solutions and discussed their limitations as well. However, more limitations and challenges could be investigated and discussed as mentioned in Section 6.3. These could eventually help with exploring additional improvements on the Intent-Based framework and take them a step forward towards more mature solutions.

Concerning **RQ2**, we proposed 2 levels of expressions for CPs and CCDN operators

respectively, as discussed in ***(Contribution 1)***. Although different expressions could be suggested based on different factors and requirements, we argue that regardless of the syntax, we still need to have multiple levels of intent/policy expressions; declarative and prescriptive, that could be used by different users that interact with the CCDN based on their level of control, exposure, role, etc.

Our proposed and implemented Intent-Based CCDN framework contributed partially towards answering **RQ3** in (***Contributions 2, 3, 4 and 5***) where we discussed our design decisions that help with utilizing the advantages of the Microservices Architecture which reflect directly on achieving some of the main features of an Intent-Based solution with conjunction with containers that serve as a perfect vessel for deploying microservices on a large scale with the help of container runtimes and orchestration platforms such as Kubernetes. However, more room for future work is needed to amplify our contribution towards **RQ3** as discussed in the Sections 6.3.

6.3 Future Work

In this thesis, we provided contributions towards an Intent-Based CCDN design. This includes a framework in which extensible and more diverse CCDN deployments can be built. We showed through evaluation the benefits of some of these design decisions. There is a clear advantage to utilizing next-generation technologies, like autonomic intents and diverse microservices, and ensuring that they are integrated into the CCDN design. This will be key to achieving increased flexibility, scalability, agility, and programmability.

The following highlights the primary avenues of research for future work that are related to the main CCDN design and development contributions in this thesis:

6.3.1 Exploring the standardized Intent Common Model

The recent intent TM Forum model standardization (TMForum, 2022) has to be explored in the CCDN domain with the corresponding microservices, and the ability to map the currently proposed intent/policy expressions to the ICM expressions. The current ICM consists of two parts: a *common intent* model and *domain-specific* extensions. The former provides the constructs for intent and intent report expressions. The latter adds vocabulary and semantics as extensions of the common model, which can be independently developed and adapted to the CCDN domain.

6.3.2 Extending current CCDN with different intent targets and their translation

Intent developers/creators (i.e., CCDN operators) are responsible for determining, defining, designing, and implementing the intent targets provided to the intent consumers (i.e., CPs) along with their corresponding translation process into some low-level representation that is understood by the underlying technology. Given the current diverse domains and users that leverage CCDNs, there is a need to consider new intent definitions and their implementations to keep up with the changing and advancing requirements, and interactions with the intent consumers.

Therefore, having an abstract intermediate level of intent translation allows the collaboration of different stakeholders to stipulate the behavioral policies and overall guidelines that could help achieve the intent at a high-level without getting caught up with the burden of lower level details and complexities.

Furthermore, the collaboration between stakeholders to better understand and implement the intent translation based on current technology advancements could be assisted by some MCDM frameworks that help with the decision-making when there are multiple evaluation criteria to be considered and different technical solutions to be selected. Therefore, building the corresponding MCDM framework (e.g., AHP graph) is an important field to be explored which could help with achieving better intent translation decisions. This includes exploring and defining the suitable evaluation criteria and their level of significance according to the intent target, and also the exploration of the current technical solution alternatives (e.g., microservices) that could realize the intent goals in various ways.

Additionally, the current Microservices Architecture advancements could be furtherly explored where different microservice types e.g., databases, load balancers) and the different alternatives for each type could be investigated, considered and evaluated in the intent translation which forms a CDN deployment that is composed of these microservice alternatives.

Moreover, interactive interaction between these different intent users in these diverse CCDN domains would be essential to increase the potential gains of intents. Thus, it is important to specify the level of involvement and contribution of the users within the intent translation and the overall system status and how far it is from achieving the intent goal.

Finally, these new intent targets might need to be refined and improved throughout the intent lifetime. Thus, it would be beneficial to explore the refinement possibilities related to these targets within their corresponding problem domain. This also requires defining the level of system awareness needed to manage the refinement efficiently which includes specifying key metrics, attributes, thresholds and events to continuously monitor them and make refinements accordingly.

6.3.3 Advancing intent APIs with Natural Language Processing

More user-friendly and advanced intent APIs need to be investigated and implemented where an additional layer of Natural Language Processing is added to further help users express their intents easily and more efficiently. These explorations would include integrating some *intent suggestions* that could assist users with their sufficient intent expression. Moreover, other API representations could be offered additionally (like a list of pre-defined intent targets that could be tuned by the users) to provide a wider variety of options to different categories of intent users.

6.3.4 Resolving intent conflicts

Since different users would express their intents simultaneously, it is important to investigate the possible conflicts between different users' intent targets, and even the conflicts between the users intents and the system's requirements as well. A good understanding of the possible conflicts that could occur at different phases (i.e., intent translation and intent refinement) could lead to better resolution during the intent translation process and throughout the whole intent lifetime. These conflict resolutions may also integrate negotiations with the intent users that may include the temporary or permanent downgrade/upgrade of the intent target that eventually gets considered in the intent translation.

References

Abhashkumar, Anubhavnidhi et al. (Nov. 2017). "Supporting diverse dynamic intent-based policies using janus". In: *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. DOI: 10.1145/3143361.3143380.

Agoulmine, Nazim et al. (July 2008). "Challenges for autonomic network management". In: *1st IEEE International Workshop on Modelling Autonomic Communications Environments (MACE)*. URL: https://api.semanticscholar.org/CorpusID:2501690.

Ak, Elif et al. (June 2018). "WAE: Workload automation engine for CDN-specialized container orchestration". In: *2018 IEEE Second International Balkan Conference on Communications and Networking (BalkanCom)*. URL: https://api.semanticscholar.org/CorpusID:227119103.

Ak, Elif et al. (June 2019). "BCDN: A proof of concept model for blockchain-aided CDN orchestration and routing". In: *Computer Networks* 161, pp. 162–171. DOI: https://doi.org/10.1016/j.comnet.2019.06.018.

Alalmaei, Shiyam et al. (Dec. 2019). "OpenCache: Distributed SDN/NFV based in-network caching as a service". In: *Advances in Data Science, Cyber Security and IT Applications* 1098, pp. 265–277. DOI: 10.1007/978-3-030-36368-0_22.

Alalmaei, Shiyam et al. (Nov. 2020). "SDN heading north: Towards a declarative intent-based northbound interface". In: *16th International Conference on Network and Service Management (CNSM)*. DOI: 10.23919/CNSM50824.2020.9269118.

Altomare, Francesco (2023). *What is a content delivery network? CDN explained — globaldots.com*. https://www.globaldots.com/resources/blog/content-delivery-network-explained/. [Accessed 13-Jul-2023].

Alwis, Chamitha de et al. (Sept. 2022). "Intelligent network softwarization". In: *6G Frontiers: Towards Future Wireless Systems*. John Wiley & Sons, Ltd. Chap. 8, pp. 93–98. DOI: https://doi.org/10.1002/9781119862321.ch8.

Amazon (2023). *Key features of a content delivery network – performance, security – amazon cloudfront — aws.amazon.com*. https://aws.amazon.com/cloudfront/features/. [Accessed 13-Jul-2023].

Arezoumand, Saeed et al. (Nov. 2017). "MD-IDN: Multi-domain intent-driven networking in software-defined infrastructures". In: *13th International Conference on Network and Service Management (CNSM)*. DOI: 10.23919/CNSM.2017.8256016.

Asheralieva, Alia et al. (Aug. 2019). "Game theory and lyapunov optimization for cloud-based content delivery networks with device-to-device and UAV-enabled caching". In: *IEEE Transactions on Vehicular Technology* 68, pp. 10094–10110. DOI: 10.1109/TVT.2019.2934027.

Baktir, Ahmet Cihat et al. (June 2022). "Intent-based cognitive closed-loop management with built-in conflict handling". In: *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. DOI: 10.1109/NetSoft54395.2022.9844074.

Barakabitze, Alcardo et al. (Dec. 2022). "Network softwarization and virtualization in future networks: The promise of SDN, NFV, MEC, and fog/cloud computing". In: *Multimedia Streaming in SDN/NFV and 5G Networks*. John Wiley & Sons, Ltd. Chap. 6, pp. 99–118. DOI: https://doi.org/10.1002/9781119800828.ch6.

Behringer, Michael et al. (June 2015). "Autonomic networking: Definitions and design goals". In: *RFC Editor* 7575, pp. 1–16. DOI: 10.17487/RFC7575.

Benkacem, Ilias et al. (Mar. 2018). "Optimal VNFs placement in CDN slicing over multi-cloud environment". In: *IEEE Journal on Selected Areas in Communications* 36, pp. 616–627. DOI: 10.1109/JSAC.2018.2815441.

Berander, Patrik et al. (Jan. 2005). "Software quality attributes and trade-offs". In: *Blekinge Institute of Technology* 97, p. 19.

Bertrand, G. et al. (2012). *RFC 6770: Use Cases for Content Delivery Network Interconnection*. USA.

Beshley, Mykola et al. (Nov. 2020). "Customer-oriented quality of service management method for the future intent-based networking". In: *Applied Sciences* 10, p. 8223. DOI: 10.3390/app10228223.

Bezahaf, Mehdi et al. (June 2021). "To All Intents and Purposes: Towards Flexible Intent Expression". In: *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. DOI: 10.1109/NetSoft51509.2021.9492554.

Broberg, James et al. (Jan. 2008). "MetaCDN: Harnessing 'storage clouds' for high performance content delivery". In: *Journal of Network and Computer Applications* 32, pp. 1012–1022. DOI: 10.1016/j.jnca.2009.03.004.

Brun, Yuriy et al. (Jan. 2009). "Engineering self-adaptive systems through feedback loops". In: *Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science* 5525, pp. 48–70. DOI: 10.1007/978-3-642-02161-9_3.

Brutlag, Jake (June 2009). *Speed matters for google web search —research.googleblog.com.* URL: https://research.googleblog.com/2009/06/speed-matters.html.

Case, Jeffrey D. et al. (Apr. 1989). *RFC1098: Simple network management protocol (SNMP)*.

Cerroni, Walter et al. (Oct. 2020). "Network softwarization and management". In: *IEEE Communications Magazine* 58, pp. 14–15. DOI: 10.1109/MCOM.2020.9247516.

Chao, Wu et al. (Feb. 2018). "Intent-based cloud service management". In: *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. DOI: 10.1109/ICIN.2018.8401600.

Charyyev, Batyr et al. (Dec. 2020). "Latency Comparison of Cloud Datacenters and Edge Servers". In: *GLOBECOM 2020 - IEEE Global Communications Conference*. DOI: 10.1109/GLOBECOM42002.2020.9322406.

Chen, Buhua et al. (Feb. 2023). "Architectural design and dynamic deployment scheme of edge computing based vCDN". In: *9th International Conference on Mechatronics and Robotics Engineering (ICMRE)*. DOI: 10.1109/ICMRE56789.2023.10106523.

Chen, Fangfei et al. (Mar. 2012). "Intra-cloud lightning: Building CDNs in the cloud." In: *2012 IEEE International Conference on Computer Communications (INFOCOM)*. DOI: 10.1109/INFCOM.2012.6195782.

Chen, Lianping (Mar. 2015). "Continuous delivery: Huge benefits, but challenges Too". In: *IEEE Software* 32, pp. 50–54. DOI: 10.1109/MS.2015.27.

Chen, Min et al. (Mar. 2017). "A 5G cognitive system for healthcare". In: *Big Data and Cognitive Computing* 1, p. 2. DOI: 10.3390/bdcc1010002.

Chen, Minggang et al. (June 2018). "Data-driven parallel video transcoding for content delivery network in the cloud". In: *5th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud) / 4th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*. DOI: 10.1109/CSCloud/EdgeCom.2018.00042.

Chen, Mingkai et al. (Mar. 2019). "A computing and content delivery network in the Smart city: Scenario, framework, and analysis". In: *IEEE Network* 33, pp. 89–95. DOI: 10.1109/MNET.2019.1800253.

Chen, Xi et al. (Jan. 2020). "CompRess: Composing overlay service resources for end-to-end network slices using semantic user intents". In: *Transactions on Emerging Telecommunications Technologies* 31, e3728. DOI: https://doi.org/10.1002/ett.3728.

Cheng, Betty H. C. et al. (Jan. 2009). "Software engineering for self-adaptive systems: A research roadmap". In: *Software Engineering for Self-Adaptive Systems* 5525. Ed. by Betty H. C. Cheng et al., pp. 1–26. DOI: 10.1007/978-3-642-02161-9_1.

Chowdhury, Shihabur Rahman et al. (Jan. 2019). "Re-architecting NFV ecosystem with microservices: state of the art and research challenges". In: *IEEE Network* 33, pp. 168–176. DOI: 10.1109/MNET.2019.1800082.

Chung, Lawrence et al. (Jan. 2000). "Non-functional requirements in software engineering". In: *International Series in Software Engineering* 5, pp. 391–441. DOI: 10.1007/978-1-4615-5269-7.

Cisco (Mar. 2017). *Cisco open media distribution data sheet.* https://www.cisco.com/c/en/us/products/collateral/video/open-media-distribution/datasheet-c78-736235.html.

Clemm, Alexander et al. (Oct. 2022). *Intent-based networking - concepts and definitions.* RFC 9315. URL: https://www.rfc-editor.org/info/rfc9315.

CNCF (Dec. 2020). *Cloud native survey 2020 — cloud native computing foundation — cncf.io.* https://www.cncf.io/reports/cloud-native-survey-2020/. [Accessed 06-Jul-2023].

Cohen, Rami et al. (Jan. 2013). "An intent-based approach for network virtualization". In: *IFIP/IEEE International Symposium on Integrated Network Management (IM).* URL: https://ieeexplore.ieee.org/document/6572968/.

Comer, Douglas et al. (Oct. 2018). "OSDF: An intent-based software defined network programming framework". In: *IEEE 43rd Conference on Local Computer Networks (LCN).* DOI: 10.1109/LCN.2018.8638149.

Cox, Jacob et al. (Oct. 2017). "Advancing software-defined networks: A survey". In: *IEEE Access* 5, pp. 25487–25526. DOI: 10.1109/ACCESS.2017.2762291.

Daroui, Danesh et al. (June 2023). "State Management of Knowledge Base in Intent Management Functions in 6G Networks". In: *2023 International Balkan Conference on Communications and Networking (BalkanCom).* DOI: 10.1109/BalkanCom58402.2023.10167994.

Deep, Bhavya et al. (July 2018). "Content rating technique for cloud-oriented content delivery network using weighted slope one scheme". In: *IEEE 11th International Conference on Cloud Computing (CLOUD).* DOI: 10.1109/CLOUD.2018.00118.

Docker (2023). *Home — docker.com.* https://www.docker.com/. [Accessed 06-Jul-2023].

Donegan, HA et al. (Oct. 1991). "A note on saaty's random indexes". In: *Mathematical and Computer Modelling* 15, pp. 135–137. DOI: https://doi.org/10.1016/0895-7177(91)90098-R.

Dragoni, Nicola et al. (Nov. 2017). "Microservices: Yesterday, today, and tomorrow". In: *Present and Ulterior Software Engineering.* Ed. by Manuel Mazzara et al. Springer, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

Du, Zongpeng et al. (Feb. 2017). *ANIMA intent policy and format.* Tech. rep. draft-du-anima-an-intent-05. Work in Progress. URL: https://datatracker.ietf.org/doc/draft-du-anima-an-intent/05/.

Duan, Jie et al. (Apr. 2018). "SCDN: A novel software-driven CDN for better content pricing and caching". In: *IEEE Communications Letters* 22.4, pp. 704–707. DOI: 10.1109/LCOMM.2018.2803808.

Elhabbash, Abdessalam et al. (Nov. 2018). “Adaptive Service Deployment using In-Network Mediation”. In: *International Conference on Network and Service Management (CNSM)*, pp. 170–176.

Elkhatib, Yehia et al. (Nov. 2017). “Charting an intent driven network”. In: *13th International Conference on Network and Service Management (CNSM)*. DOI: 10.23919/CNSM.2017.8255981.

ETSI (Oct. 2013). *Network functions virtualization (NFV); architectural framework v1.1.1*. Tech. rep. ETSI GS NFV 002. URL: http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_NFV002v010101p.pdf.

Fan, Qilin et al. (Apr. 2019). “Resource reservation and request routing for a cloud-based content delivery network”. In: *IEEE International Conference on Service-Oriented System Engineering (SOSE)*. DOI: 10.1109/SOSE.2019.00048.

Flach, Tobias et al. (Sept. 2013). “Reducing web latency: The virtue of gentle aggression”. In: *Proceedings of the ACM Special Interest Group on Data Communication Conference (SIGCOMM)*. DOI: 10.1145/2486001.2486014.

FortuneBusinessInsights (July 2023a). *The global cloud microservices market size forecast, 2023-2030 — fortunebusinessinsights.com*. https://www.fortunebusinessinsights.com/video-streaming-market-103057. [Accessed 05-Sep-2023].

FortuneBusinessInsights (May 2023b). *Video streaming market size forecast, 2023-2030 — fortunebusinessinsights.com*. https://www.fortunebusinessinsights.com/video-streaming-market-103057. [Accessed 01-Sep-2023].

Franek, Jiri et al. (Dec. 2014). “Judgment scales and consistency measure in AHP”. In: *Procedia Economics and Finance* 12, pp. 164–173. DOI: 10.1016/S2212-5671(14)00332-3.

Frangoudis, Pantelis et al. (May 2016). “An architecture for on-demand service deployment over a telco CDN”. In: *IEEE International Conference on Communications (ICC)*. DOI: 10.1109/ICC.2016.7510921.

Frangoudis, Pantelis et al. (June 2017). “CDN-as-a-service provision over a telecom operator’s cloud”. In: *IEEE Transactions on Network and Service Management* 14, pp. 702–716. DOI: 10.1109/TNSM.2017.2710300.

Frank, Benjamin et al. (July 2013). “Pushing CDN-ISP collaboration to the limit”. In: *ACM SIGCOMM Computer Communication Review* 43, pp. 34–44. DOI: 10.1145/2500098.2500103.

Garlan, David et al. (Nov. 2004). “Rainbow: Architecture-based self-adaptation with reusable infrastructure”. In: *Computer* 37, pp. 46–54. DOI: 10.1109/MC.2004.175.

Gartner (2020). *Gartner forecasts strong revenue growth for global container management software and services through 2024*. https://www.gartner.com/en/newsroom/press-releases/2020-06-25-gartner-forecasts-strong-revenue-growth-for-global-co. [Accessed 06-Jul-2023].

GlobeNewsWire (July 2023a). *Global cloud content delivery network (CDN) market forecast, 2023-2030 — globenewswire.com*. https://https://www.globenewswire.com/news-release/2023/07/07/2701217/0/en/Global-Cloud-Content-Delivery-Network-CDN-Market-to-Reach-30-5-Billion-by-2030.html. [Accessed 07-Sep-2023].

GlobeNewsWire (Aug. 2023b). *Intent-based networking market forecast, 2023-2030 — globenewswire.com*. https://www.globenewswire.com/news-release/2023/08/28/2732753/0/en/Intent-based-Networking-Market-to-be-Worth-8-8-Billion-by-2030-Exclusive-Report-by-Meticulous-Research.html. [Accessed 05-Sep-2023].

Google (2023a). *About cluster configuration choices — google kubernetes engine (GKE) — google cloud*. https://cloud.google.com/kubernetes-engine/docs/concepts/types-of-clusters#regional_clusters. [Accessed 24-Jun-2023].

Google (2023b). *API and gcloud references — cloud CDN — Google cloud — cloud.google.com*. https://cloud.google.com/cdn/docs/apis. [Accessed 05-Jul-2023].

Google (2023c). *Cloud CDN overview — cloud.google.com*. https://cloud.google.com/cdn/docs/overview. [Accessed 24-Jun-2023].

Google (2023d). *Compare GKE autopilot and standard — google kubernetes engine (GKE) — google cloud — cloud.google.com*. https://cloud.google.com/kubernetes-engine/docs/resources/autopilot-standard-feature-comparison. [Accessed 24-Jun-2023].

Google (2023e). *Global locations - regions & zones — google cloud — cloud.google.com*. https://cloud.google.com/about/locations. [Accessed 16-Jul-2023].

Google (2023f). *Google edge network — peering.google.com*. https://peering.google.com/#/infrastructure. [Accessed 16-Jul-2023].

Google (2023g). *Pricing — compute engine: virtual nachines (VMs) — google cloud — cloud.google.com*. https://cloud.google.com/compute/all-pricing. [Accessed 24-Jun-2023].

GrandViewResearch (2023). *Content delivery network market Size Report, 2022-2030 — grandviewresearch.com*. https://www.grandviewresearch.com/industry-analysis/content-delivery-networks-cnd-market. [Accessed 01-Sep-2023].

Griffith, Philip (2023). *ahpy — pypi.org*. https://pypi.org/project/ahpy/. [Accessed 26-Jun-2023].

Gritli, Nour et al. (Oct. 2021). “Decomposition and propagation of intents for network slice design”. In: *4th IEEE 5G World Forum (5GWF)*. DOI: 10.1109/5GWF52925.2021.00036.

Gupta, Rohit Kumar et al. (July 2017). “2-Tiered cloud based content delivery network architecture: An efficient load balancing approach for video streaming”.

In: *International Conference on Signal Processing and Communication (ICSPC)*. DOI: 10.1109/CSPC.2017.8305885.

Han, Yoonseon et al. (Oct. 2016). "An intent-based network virtualization platform for SDN". In: *12th International Conference on Network and Service Management (CNSM)*. DOI: 10.1109/CNSM.2016.7818446.

Horn, Petr Jan (Oct. 2001). "Autonomic computing: IBM's perspective on the state of information technology". In: *Computing Systems*.

Hu, Han et al. (Jan. 2016). "Joint content replication and request routing for social video distribution over cloud CDN: A community clustering method". In: *IEEE Transactions on Circuits and Systems for Video Technology* 26, pp. 1320–1333. DOI: 10.1109/TCSVT.2015.2455712.

Huebscher, Markus et al. (Aug. 2008). "A survey of autonomic computing - degrees, models, and applications". In: *ACM Computing Surveys (CSUR)* 40, pp. 1–28. DOI: 10.1145/1380584.1380585.

IBM (2006). "An architectural blueprint for autonomic computing". In: *IBM White Paper* 31.2006, pp. 1–6.

IEEEStandards (1990). "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std* 610, pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

Ishizaka, Alessio et al. (Dec. 2006). "How to derive priorities in AHP: a comparative study". In: *Central European Journal of Operations Research* 14, pp. 387–400. DOI: 10.1007/s10100-006-0012-9.

Ishizaka, Alessio et al. (May 2011). "Review of the main developments in the analytic hierarchy process". In: *Expert systems with applications* 38, pp. 14336–14345. DOI: 10.1016/j.eswa.2011.04.143.

James, Hayden (Sept. 2020). *25 Best CDN providers 2020*. https://haydenjames.io/best-cdn-providers/.

Jayakumar, Suman et al. (Jan. 2018). "An investigational study and analysis of cloud-based content delivery network: Perspectives". In: *International Journal of Advanced Computer Science and Applications* 9, pp. 307–314. DOI: 10.14569/IJACSA.2018.091037.

Jennings, Brendan et al. (Nov. 2007). "Towards autonomic management of communications networks". In: *IEEE Communications Magazine* 45, pp. 112–121. DOI: 10.1109/MCOM.2007.4342833.

Jia, Qingmin et al. (June 2017). "The collaboration for content delivery and network infrastructures: A survey". In: *IEEE Access* 5, pp. 18088–18106. DOI: 10.1109/ACCESS.2017.2715824.

Jin, Xin et al. (Apr. 2017). "K-means clustering". In: *Encyclopedia of Machine Learning and Data Mining*. Springer US, pp. 695–697. DOI: 10.1007/978-1-4899-7687-1_431.

Junior, Ronaldo et al. (Oct. 2018). “Cloud application architecture appraiser (CA3): A multicriteria approach and tool for assessing cloud deployment options based on nonfunctional requirements”. In: *Software Practice and Experience* 48, pp. 1–24. DOI: 10.1002/spe.2644.

Karlsson, Joachim et al. (1998). “An evaluation of methods for prioritizing software requirements”. In: *Information and software technology* 39, pp. 939–947. DOI: 10.1016/S0950-5849(97)00053-0.

Kassab, Mohamad (May 2013). “An integrated approach of AHP and NFRs framework”. In: *IEEE 7th International Conference on Research Challenges in Information Science (RCIS)*. DOI: 10.1109/RCIS.2013.6577705.

Katchabaw, M.J. et al. (Jan. 1996). “Policy-driven fault management in distributed systems”. In: *Proceedings of 7th International Symposium on Software Reliability Engineering (ISSRE)*. DOI: 10.1109/ISSRE.1996.558833.

Kephart, Jeffrey et al. (Feb. 2003). “The vision of autonomic computing”. In: *Computer* 36.1, pp. 41–50. DOI: 10.1109/MC.2003.1160055.

Kephart, Jeffrey et al. (June 2004). “An artificial intelligence perspective on autonomic computing policies.” In: *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*. DOI: 10.1109/POLICY.2004.1309145.

Khemka, Bhavesh et al. (Dec. 2014). “Utility functions and resource management in an oversubscribed heterogeneous computing Environment”. In: *IEEE Transactions on Computers* 64, pp. 2394–2407. DOI: 10.1109/TC.2014.2360513.

Kiran, Mariam et al. (2018). “Enabling intent to configure scientific networks for high performance demands”. In: *Future Generation Computer Systems* 79, pp. 205–214.

Kreutz, Diego et al. (June 2014). “Software-defined networking: A comprehensive survey”. In: *ArXiv e-prints* 103, pp. 14–76. DOI: 10.1109/JPROC.2014.2371999.

Kubernetes (2023a). *Deployments — kubernetes.io*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Accessed 26-Jun-2023].

Kubernetes (2023b). *Horizontal pod autoscaling — kubernetes.io*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. [Accessed 26-Jun-2023].

Kubernetes (2023c). *Kubernetes components — kubernetes.io*. https://kubernetes.io/docs/concepts/overview/components/. [Accessed 26-Jun-2023].

Kubernetes (2023d). *Production-grade container orchestration — kubernetes.io*. https://kubernetes.io/. [Accessed 06-Jul-2023].

Leivadeas, Aris et al. (Mar. 2021). “VNF placement problem: A multi-tenant intent-based networking approach”. In: *24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. DOI: 10.1109/ICIN51074.2021.9385553.

Leivadeas, Aris et al. (Jan. 2022). "A survey on intent-based networking". In: *Commun. Surveys Tuts.IEEE Communications Surveys & Tutorials* 25, pp. 625–655. DOI: 10.1109/COMST.2022.3215919.

Li, Tangyi et al. (June 2023). "Autonomous intent detection for intent-driven satellite network". In: *International Wireless Communications and Mobile Computing (IWCMC)*. DOI: 10.1109/IWCMC58020.2023.10183156.

Linthicum, David S. (Sept. 2016). "Practical use of microservices in moving workloads to the cloud". In: *IEEE Cloud Computing* 3, pp. 6–9. DOI: 10.1109/MCC.2016.114.

Lopes, Felipe et al. (Nov. 2015). "A Software engineering perspective on SDN programmability". In: *IEEE Communications Surveys & Tutorials* 18, pp. 1255–1272. DOI: 10.1109/COMST.2015.2501026.

Lyko, Tomasz et al. (Sept. 2022). "QoE Assessment for Multi-Video Object Based Media". In: *International Conference on Quality of Multimedia Experience (QoMEX)*. IEEE. DOI: 10.1109/QoMEX55416.2022.9900905.

Mangili, Michele et al. (June 2016). "Optimal planning of virtual content delivery networks under uncertain traffic demands". In: *Computer Networks* 106, pp. 186–195. DOI: 10.1016/j.comnet.2016.06.035.

Manzalini, Antonio et al. (July 2016). *Towards 5G software-defined ecosystems*. Tech. rep. IEEE SDN White Paper. https://sdn.ieee.org/publications/towards-5g-software-defined-ecosystems.

Marie-Magdelaine, Nicolas et al. (Apr. 2019). "Demonstration of an observability framework for cloud Native microservices". In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. URL: https://ieeexplore.ieee.org/document/8717923.

Marsico, Antonio et al. (July 2017). "An interactive intent-based negotiation scheme for application-centric networks". In: *IEEE Conference on Network Softwarization (NetSoft)*. DOI: 10.1109/NETSOFT.2017.8004251.

Massa, Jacopo et al. (2023). "Declarative Provisioning of Virtual Network Function Chains in Intent-based Networks". In: *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. DOI: 10.1109/NetSoft57336.2023.10175449.

Mehmood, Kashif et al. (July 2023). "Knowledge-based Intent Modeling for Next Generation Cellular Networks". In: *2023 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*. DOI: 10.48550/arXiv.2302.08544.

Microsoft (2023a). *Azure CDN POP locations by region — learn.microsoft.com*. https://learn.microsoft.com/en-us/azure/cdn/cdn-pop-locations. [Accessed 16-Jul-2023].

Microsoft (2023b). *Azure Content Delivery Network — azure.microsoft.com.* https://https://azure.microsoft.com/en-us/products/cdn. [Accessed 16-Jul-2023].

Mijumbi, Rashid et al. (Sept. 2015). "Network function virtualization: state-of-the-art and research challenges". In: *IEEE Communications Surveys & Tutorials* 18, pp. 236–262. DOI: 10.1109/COMST.2015.2477041.

Monga, Inder et al. (Nov. 2018). "SDN for end-to-end networked science at the exascale (SENSE)". In: *IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*. DOI: 10.1109/INDIS.2018.00007.

NGINX (2023). *Welcome to NGINX wiki! — NGINX — nginx.com.* https://www.nginx.com/resources/wiki/. [Accessed 27-Jun-2023].

Nydick, Robert L et al. (Mar. 1992). "Using the analytic hierarchy process to structure the supplier selection procedure". In: *International Journal of Purchasing and Materials Management* 28, pp. 31–36. DOI: 10.1111/j.1745-493X.1992.tb00561.x.

ONF (2014). *Intent NBI definition and principles.* https://opennetworking.org/wp-content/uploads/2014/10/TR-523_Intent_Definition_Principles.pdf.

ONF (2015a). *Github - Opennetworkingfoundation/BOULDER-intent-NBI: Intent-based NBI development — github.com.* https://github.com/OpenNetworkingFoundation/BOULDER-Intent-NBI. [Accessed 16-Jul-2023].

ONF (2015b). *Northbound interfaces working group archives.* https://opennetworking.org/tag/northbound-interfaces-working-group/.

ONOS (2015). https://wiki.onosproject.org/display/ONOS/Intent+Framework.

OpenDaylight (2015). *Network intent composition (NIC) user guide - OpenDaylight Documentation Fluorine documentation.* https://docs.opendaylight.org/en/stable-fluorine/user-guide/network-intent-composition-(nic)-user-guide.html.

OpenDaylight (2016). *NEtwork MOdeling (NEMO) - OpenDaylight Documentation Fluorine documentation.* https://docs.opendaylight.org/en/stable-fluorine/release-notes/projects/nemo.html.

OpenStack (2016). *GroupBasedPolicy.* https://wiki.openstack.org/wiki/GroupBasedPolicy.

OSM (2023). *OSM.* https://osm.etsi.org/.

Ouyang, Ying et al. (2021). "A Brief Survey and Implementation on Refinement for Intent-Driven Networking". In: *IEEE Network* 35.6, pp. 75–83. DOI: 10.1109/MNET.001.2100194.

Ouyang, Ying et al. (Nov. 2022). "Ontology-based network intent refinement framework". In: *22nd IEEE International Conference on Communication Technology (ICCT)*. DOI: 10.1109/ICCT56141.2022.10072810.

Palau, Carlos et al. (Jan. 2003). "CCDN: campus content delivery network learning facility". In: *Proceedings 3rd IEEE International Conference on Advanced Technologies (ICALT)*. DOI: 10.1109/ICALT.2003.1215188.

Pang, Lei et al. (Jan. 2020). "A survey on intent-driven networks". In: *IEEE Access* 8, pp. 22862–22873. DOI: 10.1109/ACCESS.2020.2969208.

Papagianni, Chrysa et al. (Sept. 2013). "A cloud-oriented content delivery network paradigm: Modeling and assessment". In: *IEEE Transactions on Dependable and Secure Computing* 10, pp. 287–300. DOI: 10.1109/TDSC.2013.12.

Parashar, Manish et al. (Jan. 2004). "Autonomic Computing: An Overview". In: *Unconventional Programming Paradigms, International Workshop (UPP)*. DOI: 10.1007/11527800_20.

Pedregosa, F. et al. (Nov. 2011). "Scikit-learn: Machine learning in python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830. DOI: 10.48550/arXiv.1201.0490.

Pham, Minh et al. (June 2016). "SDN applications - The intent-based northbound interface realisation for extended applications". In: *IEEE NetSoft Conference and Workshops (NetSoft)*. DOI: 10.1109/NETSOFT.2016.7502469.

Qassem, Lamees M. Al et al. (Nov. 2022). "Optimal resource allocation for containerized cloud microservices". In: *International Conference on Electrical and Computing Technologies and Applications (ICECTA)*. DOI: 10.1109/ICECTA57148.2022.9990377.

Retal, Sara et al. (May 2017). "Content delivery network slicing: QoE and cost awareness". In: *IEEE International Conference on Communications (ICC)*. DOI: 10.1109/ICC.2017.7996499.

Roy, Sandip et al. (June 2015). "Fuzzy based dynamic load balancing scheme for efficient edge server selection in cloud-oriented content delivery network using voronoi diagram". In: *IEEE International Advance Computing Conference (IACC)*. DOI: 10.13140/RG.2.1.1996.5287.

Saaty, Thomas L (1985). "Decision making for leaders: The analytical hierarchy process for decisions in a complex world". In: *IEEE transactions on systems, man, and cybernetics*, pp. 450–452.

Saaty, Thomas L (Sept. 1990). "How to make a decision: the analytic hierarchy process". In: *European journal of operational research* 48, pp. 9–26.

Saaty, Thomas L (Jan. 1994a). *Fundamentals of decision making and priority theory with the analytic hierarchy process*. Vol. VI. RWS publications.

Saaty, Thomas L et al. (1979). "A new approach to performance measurement the analytic hierarchy process". In: *Design Methods and Theories* 13, pp. 62–68.

Saaty, Thomas L et al. (1991). “Analytical planning, the organization of systems; the analytic hierarchy process series”. In: *RWS Publications* 4.

Saaty, Thomas L et al. (2012). “Models, methods, concepts & applications of the analytic hierarchy process”. In: *Springer Science & Business Media* 175.

Saaty, Thomas L. (1994b). “How to make a decision: The analytic hierarchy process”. In: *Interfaces* 24, pp. 19–43. URL: http://www.jstor.org/stable/25061950.

Saboor, Abdul et al. (July 2021). “Design pattern based distribution of microservices in cloud computing Eenvironment”. In: *International Conference on Computer & Information Sciences (ICCOINS)*. DOI: 10.1109/ICCOINS49721.2021.9497188.

Sajithabanu, S. et al. (Nov. 2016). “Cloud based content delivery network using genetic optimization algorithm for storage cost”. In: *IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*. DOI: 10.1109/ANTS.2016.7947822.

Scheid, Eder J. et al. (May 2017). “INSpIRE: Integrated NFV-based intent refinement environment”. In: *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. DOI: 10.23919/INM.2017.7987279.

Serrano, Jaime Martín et al. (June 2007). “Ontology-based reasoning for supporting context-aware services on autonomic networks”. In: *IEEE International Conference on Communications (ICC)*. DOI: 10.1109/ICC.2007.347.

Simon, Herbert A. (1996). *The sciences of the artificial*. 3rd ed. MIT Press.

Singh, Vindeep et al. (May 2017). “Container-based microservice architecture for cloud applications”. In: *International Conference on Computing, Communication and Automation (ICCCA)*. DOI: 10.1109/CCAA.2017.8229914.

Sinreich, David (2006). “An architectural blueprint for autonomic computing”. In: *Autonomic Computing White Paper*. URL: https://api.semanticscholar.org/CorpusID:16909837.

Sousa, Nathan et al. (Mar. 2018). “network service orchestration: A survey”. In: *Computer Communications* 142-143, pp. 69–94. DOI: 10.1016/j.comcom.2019.04.008.

Sterritt, Roy (Apr. 2005). “Autonomic computing”. In: *Innovations in Systems and Software Engineering* 1, pp. 535–539. DOI: 10.1007/s11334-005-0001-5.

Szilágyi, Péter (June 2021). “I2BN: Intelligent intent based networks”. In: *Journal of ICT Standardization* 9, pp. 159–200. DOI: 10.13052/jicts2245-800X.926.

Taleb, Tarik et al. (Sept. 2020). “CDN slicing over a multi-domain edge cloud”. In: *IEEE Transactions on Mobile Computing* 19, pp. 2010–2027. DOI: 10.1109/TMC.2019.2921712.

Tene, Gil (2019). *Github - giltene/wrk2: A constant throughput, correct latency recording variant of wrk — github.com*. https://github.com/giltene/wrk2. [Accessed 24-Jun-2023].

TheExpressWire (Aug. 2023). *SDN and NFV market share report 2023-2030 — benzinga.com*. https://www.benzinga.com/pressreleases/23/08/33771870/sdn-and-nfv-market-share-report-2023-2030-107-pages-report. [Accessed 05-Sep-2023].

TMForum (2014). *TR218 B2B2X partnering guidebook step by step guide, version 0.6.4 — tmforum.org*. https://www.tmforum.org/resources/technical-report-best-practice/tr218-b2b2x-partnering-guidebook-_-step-by-step-guide/. [Accessed 05-Jul-2023].

TMForum (May 2021). *IG1253A Intent Modeling v1.0.0*. Tech. rep. https://www.tmforum.org/resources/how-to-guide/ig1253a-intent-modeling-v1-0-0/.

TMForum (2022). *Intent common model v2.1.0*. Tech. rep. TR292. https://www.tmforum.org/resources/how-to-guide/ig1253a-intent-modeling-v1-0-0/.

Tran, Hai-Anh et al. (Mar. 2019). "MABRESE: A new server selection method for Smart SDN-based CDN architecture". In: *IEEE Communications Letters* 23, pp. 1012–1015. DOI: 10.1109/LCOMM.2019.2907948.

Triantaphyllou, Evangelos (Jan. 2000). *Multi-criteria decision making methods*. Vol. 44. Springer US, pp. 5–21. DOI: 10.1007/978-1-4757-3157-6.

Trois, Celio et al. (2016). "A survey on SDN programming languages: Toward a taxonomy". In: *IEEE Communications Surveys & Tutorials* 18, pp. 2687–2712. DOI: 10.1109/COMST.2016.2553778.

Tsai, Wei-Tek (Nov. 2005). "Service-oriented system engineering: A new paradigm". In: *IEEE International Workshop on Service-Oriented System Engineering (SOSE)*. DOI: 10.1109/SOSE.2005.34.

Tuncer, Daphne et al. (Dec. 2018). "A northbound interface for software-based networks". In: *14th International Conference on Network and Service Management (CNSM)*. URL: https://ieeexplore.ieee.org/document/8584931.

Ustok, Refik Fatih et al. (2022). "Asset Administration Shell as an Enabler of Intent-Based Networks for Industry 4.0 Automation". In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. DOI: 10.1109/ETFA52439.2022.9921446.

Villamizar, Mario et al. (May 2016). "Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures". In: *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. DOI: 10.1109/CCGrid.2016.37.

Whitaker Rozann, Saaty (Dec. 1987). "The analytic hierarchy process—what it is and how it is used". In: *Mathematical Modelling* 9, pp. 161–176. DOI: https://doi.org/10.1016/0270-0255(87)90473-8.

Woo, Honguk et al. (Apr. 2014). "A virtualized, programmable content delivery network". In: *IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. DOI: 10.1109/MobileCloud.2014.32.

Xie, Min et al. (Dec. 2022). "Intent-Driven Management for Multi-Vertical End-to-End Network Slicing Services". In: *2022 IEEE Globecom Workshops (GC Wkshps)*. DOI: 10.1109/GCWkshps56602.2022.10008673.

Xiong, Wenjie et al. (Oct. 2018). "Real-time processing and storage of multimedia data with content delivery network in vehicle monitoring system". In: *6th International Conference on Wireless Networks and Mobile Communications (WINCOM)*. DOI: 10.1109/WINCOM.2018.8629708.

Yrjönen, Anton et al. (Jan. 2009). "Extending the NFR framework with measurable nonFunctional requirements". In: *The 2nd International Workshop on Non-functional System Properties in Domain Specific Modeling Languages (NF-PinDSML)*. URL: https://ceur-ws.org/Vol-553/paper2.pdf.

Zeydan, Engin et al. (May 2020). "Recent Advances in Intent-Based Networking: A Survey". In: *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*. DOI: 10.1109/VTC2020-Spring48590.2020.9128422.

Zheng, Haomian et al. (June 2023). "From Automation to Autonomous: Driving the Optical Network Management to Fixed Fifth-generation (F5G) Advanced". In: *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. DOI: 10.1109/NetSoft57336.2023.10175446.

Zolfaghari, Behrouz et al. (Apr. 2020). "Content delivery networks: State of the art, trends, and future roadmap". In: *ACM Computing Surveys* 53, pp. 1–34. DOI: 10.1145/3380613.

Appendix A

Evaluation Extended Results

A.1 Detailed Experiment Tables for Normal Traffic Scenario

The following tables list the detailed experiment results for all 10 repetitions. These tables show tested time and traffic segments from the ISP's CDN dataset and the number of dropped requests. Tables A.1 and A.2 list the number of dropped requests in Baseline 1 and Baseline 2 respectively. LCI 1 and its optimistic and pessimistic refinement results are listed in Tables A.3, A.4 and A.5. LCI 2 results are listed in Tables A.6, A.7 and both its refinements are listed in Tables A.8, A.9 and A.10. Finally, LCI 3 results are listed in Tables A.11, A.12, its optimistic refinement results are in Tables A.13 and A.14, and its pessimistic refinement results are in Table A.15.

Date 2018	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Sat 13/10	17:01	5043	74	624	26	83	29	74	129	83	40	74	123.6	394.8
	19:44	7972	193	204	200	295	233	393	288	379	241	286	271.2	
Sun 14/10	13:21	5172	542	915	252	1074	832	683	1384	444	617	1032	777.5	3586.9
	16:51	7578	589	144	833	308	390	568	755	347	804	209	494.7	
	19:35	11129	3870	2312	2445	1745	1184	2182	2020	2317	2845	2227	2314.7	
Mon 15/10	13:37	2601	827	915	862	892	863	1182	1096	1083	1079	557	935.6	4410
	17:17	5062	432	519	481	754	305	663	443	1109	834	219	575.9	
	18:33	8416	470	1466	1747	1828	848	1589	1375	1456	1373	892	1304.4	
	19:19	10353	1837	1560	1521	1600	2272	1064	1563	1559	1780	1185	1594.1	
Tue 16/10	14:54	2674	812	1241	997	1209	865	1016	1066	816	626	961	960.9	2968.3
	17:59	5072	1235	557	1472	573	1190	983	564	590	915	1643	972.2	
	18:39	7474	965	911	1265	973	1067	1301	1450	519	1231	670	1035.2	
Wed 17/10	14:55	2955	866	887	1082	1214	1283	1031	1004	792	838	990	998.7	4140.9
	17:54	5019	520	889	431	718	384	317	403	993	1390	908	695.3	
	19:04	7874	889	1512	1005	1899	681	1467	946	1294	1415	654	1176.2	
	19:27	10137	2055	570	1788	730	1174	860	1160	1845	1492	1033	1270.7	
Thur 18/10	15:12	2793	1071	882	1075	670	925	1091	669	1287	1418	955	1004.3	3770.4
	17:52	5131	1117	840	846	483	815	570	1018	780	300	638	740.7	
	18:40	7737	1082	1450	491	1266	678	940	1294	489	760	860	931	
	19:05	9896	1072	1245	1041	790	945	1060	1466	1061	1556	708	1094.4	
Fri 19/10	14:55	2614	1158	715	1133	1013	1038	1222	1018	1340	794	992	1042.3	3237.7
	18:07	5180	1535	1080	508	470	1082	844	1061	542	686	893	870.1	
	19:12	7981	2415	1836	1747	1612	1286	958	734	775	1172	718	1325.3	

Table A.1: Baseline1 (GKE Autopilot) Dropped Requests During Scale-outs.

Date 2018	Time	Input Req Rate/Minute	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Sat 13/10	17:01	5043	21	21	104	103	103	22	65	103	21	21	58.4	264
	19:44	7972	105	274	204	207	35	34	368	619	105	105	205.6	
Sun 14/10	13:21	5172	84	183	85	85	85	22	22	109	62	69	80.6	880.5
	16:51	7578	39	189	124	104	104	32	117	216	130	130	118.5	
	19:35	11129	946	972	165	891	1207	810	158	687	150	828	681.4	
Mon 15/10	13:37	2601	13	13	43	43	42	11	33	11	34	34	27.7	1596.2
	17:17	5062	74	27	26	132	27	111	67	59	21	21	56.5	
	18:33	8416	117	211	943	733	913	1328	600	1363	35	1257	750	
	19:19	10353	724	338	723	387	182	789	1336	1537	911	693	762	
Tue 16/10	14:54	2674	13	13	43	69	13	31	50	30	11	34	30.7	201.8
	17:59	5072	84	139	86	28	28	22	68	22	22	69	56.8	
	18:39	7474	125	111	111	124	124	88	32	88	242	98	114.3	
Wed 17/10	14:55	2955	31	56	31	40	10	11	30	37	62	37	34.5	786
	17:54	5019	22	67	148	67	67	108	63	22	29	22	61.5	
	19:04	7874	167	167	35	35	167	105	352	338	105	121	159.2	
	19:27	10137	852	920	740	48	164	273	329	882	244	856	530.8	
Thur 18/10	15:12	2793	30	10	10	30	10	36	11	10	10	31	18.8	574.9
	17:52	5131	64	22	64	111	64	68	68	69	22	22	57.4	
	18:40	7737	33	33	102	160	51	33	160	91	33	103	79.9	
	19:05	9896	536	333	493	511	354	109	138	519	828	367	418.8	
Fri 19/10	14:55	2614	55	10	31	10	10	31	55	31	33	11	27.7	226.9
	18:07	5180	22	69	22	69	22	22	110	69	22	22	44.9	
	19:12	7981	99	33	92	93	34	179	104	115	154	640	154.3	

Table A.2: Baseline2 (Fully-Managed GKE) Dropped Requests During Scale-outs.

Date 2018	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Sat 13/10	17:01	5043	60	21	66	21	21	109	66	66	67	104	60.1	316
	19:44	7972	35	94	303	166	356	537	494	106	433	35	255.9	
Sun 14/10	13:21	5172	27	185	78	77	134	62	62	63	109	62	85.9	1528.5
	16:51	7578	42	188	250	46	104	244	203	228	219	179	170.3	
	19:35	11129	1253	1298	888	1307	996	51	1670	1720	1750	1790	1272.3	
Mon 15/10	13:37	2601	13	71	72	14	42	11	34	58	11	35	36.1	1956
	17:17	5062	84	131	132	29	26	25	72	21	22	26	56.8	
	18:33	8416	1121	846	211	1066	932	97	1303	1210	1280	1260	932.6	
	19:19	10353	587	963	797	396	596	868	1078	1375	1323	1322	930.5	
Tue 16/10	14:54	2674	39	43	43	68	35	57	55	11	31	11	39.3	237.9
	17:59	5072	139	28	29	85	85	114	66	23	35	22	62.6	
	18:39	7474	124	40	195	124	125	156	99	366	32	99	136	
Wed 17/10	14:55	2955	32	11	57	30	38	30	35	64	31	67	39.5	1057.6
	17:54	5019	67	103	67	67	109	81	72	22	68	23	67.9	
	19:04	7874	98	452	247	168	147	175	706	99	704	705	350.1	
	19:27	10137	44	120	134	707	295	1016	909	1035	994	747	600.1	
Thur 18/10	15:12	2793	10	11	11	56	32	55	57	37	57	11	33.7	759.5
	17:52	5131	23	23	68	65	70	119	96	115	70	65	71.4	
	18:40	7737	141	226	35	56	161	504	467	282	103	604	257.9	
	19:05	9896	692	569	622	201	436	255	455	259	371	442	430.2	
Fri 19/10	14:55	2614	10	31	32	10	34	11	56	11	11	53	25.9	428
	18:07	5180	23	112	37	22	63	96	64	71	111	111	71	
	19:12	7981	100	33	99	93	159	979	1127	222	341	158	331.1	

Table A.3: LCI1 Dropped Requests During Scale-outs.

Date 2018	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Sat 13/10	17:01	5043	60	21	66	21	21	109	66	66	67	104	60.1	265.7
	19:44	7972	105	274	204	207	35	34	368	619	105	105	205.6	
Sun 14/10	13:21	5172	27	185	78	77	134	62	62	63	109	62	85.9	937.6
	16:51	7578	42	188	250	46	104	244	203	228	219	179	170.3	
	19:35	11129	946	972	165	891	1207	810	158	687	150	828	681.4	
Mon 15/10	13:37	2601	13	71	72	14	42	11	34	58	11	35	36.1	1604.9
	17:17	5062	84	131	132	29	26	25	72	21	22	26	56.8	
	18:33	8416	117	211	943	733	913	1328	600	1363	35	1257	750	
	19:19	10353	724	338	723	387	182	789	1336	1537	911	693	762	
Tue 16/10	14:54	2674	39	43	43	68	35	57	55	11	31	11	39.3	216.2
	17:59	5072	139	28	29	85	85	114	66	23	35	22	62.6	
	18:39	7474	125	111	111	124	124	88	32	88	242	98	114.3	
Wed 17/10	14:55	2955	32	11	57	30	38	30	35	64	31	67	39.5	797.4
	17:54	5019	67	103	67	67	109	81	72	22	68	23	67.9	
	19:04	7874	167	167	35	35	167	105	352	338	105	121	159.2	
	19:27	10137	852	920	740	48	164	273	329	882	244	856	530.8	
Thur 18/10	15:12	2793	10	11	11	56	32	55	57	37	57	11	33.7	570.1
	17:52	5131	23	23	68	65	70	119	96	115	70	65	71.4	
	18:40	7737	33	33	102	160	51	33	160	91	33	103	79.9	
	19:05	9896	536	333	493	511	354	109	138	519	828	367	418.8	
Fri 19/10	14:55	2614	10	31	32	10	34	11	56	11	11	53	25.9	225.1
	18:07	5180	22	69	22	69	22	22	110	69	22	22	44.9	
	19:12	7981	99	33	92	93	34	179	104	115	154	640	154.3	

Table A.4: Optimistically Refined LCI1 Dropped Requests During Scale-outs (green rows represent the refinement occurrence).

Date 2018	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Sat 13/10	17:01	5043	21	21	104	103	103	22	65	103	21	21	58.4	264
	19:44	7972	105	274	204	207	35	34	368	619	105	105	205.6	
Sun 14/10	13:21	5172	27	185	78	77	134	62	62	63	109	62	85.9	937.6
	16:51	7578	42	188	250	46	104	244	203	228	219	179	170.3	
	19:35	11129	946	972	165	891	1207	810	158	687	150	828	681.4	
Mon 15/10	13:37	2601	13	71	72	14	42	11	34	58	11	35	36.1	1604.6
	17:17	5062	74	27	26	132	27	111	67	59	21	21	56.5	
	18:33	8416	117	211	943	733	913	1328	600	1363	35	1257	750	
	19:19	10353	724	338	723	387	182	789	1336	1537	911	693	762	
Tue 16/10	14:54	2674	39	43	43	68	35	57	55	11	31	11	39.3	210.4
	17:59	5072	84	139	86	28	28	22	68	22	22	69	56.8	
	18:39	7474	125	111	111	124	124	88	32	88	242	98	114.3	
Wed 17/10	14:55	2955	32	11	57	30	38	30	35	64	31	67	39.5	791
	17:54	5019	22	67	148	67	67	108	63	22	29	22	61.5	
	19:04	7874	167	167	35	35	167	105	352	338	105	121	159.2	
	19:27	10137	852	920	740	48	164	273	329	882	244	856	530.8	
Thur 18/10	15:12	2793	10	11	11	56	32	55	57	37	57	11	33.7	556.1
	17:52	5131	64	22	64	111	64	68	68	69	22	22	57.4	
	18:40	7737	33	33	102	160	51	33	160	91	33	103	79.9	
	19:05	9896	536	333	493	511	354	109	138	519	828	367	418.8	
Fri 19/10	14:55	2614	10	31	32	10	34	11	56	11	11	53	25.9	225.1
	18:07	5180	22	69	22	69	22	22	110	69	22	22	44.9	
	19:12	7981	99	33	92	93	34	179	104	115	154	640	154.3	

Table A.5: Pessimistically Refined LCI1 Dropped Requests During Scale-outs (green rows represent the refinement occurrence).

Date 2018	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Sat 13/10	13:25	3799	50	52	59	49	45	105	16	83	45	44	54.8	659.9
	17:01	5043	66	170	103	60	60	103	170	59	65	66	92.2	
	19:34	5822	156	29	343	193	434	245	657	201	554	204	301.6	
	19:41	7484	94	105	94	35	35	252	470	34	197	797	211.3	
Sun 14/10	14:15	5729	154	28	151	263	436	553	273	654	232	413	315.7	7438.4
	15:43	6067	255	196	291	127	86	25	571	26	164	276	201.7	
	16:50	6988	239	656	723	744	684	99	724	981	797	898	654.5	
	17:59	8099	643	983	516	668	784	768	589	909	623	1021	750.4	
	19:35	12414	3485	3306	3176	3367	3602	2912	2156	2546	3491	2611	3065.2	
	19:40	11129	2364	2277	2057	2156	2424	2295	2661	2668	2533	3074	2450.9	
Mon 15/10	13:50	2953	21	119	103	188	1739	120	16	50	51	16	242.3	5682.6
	15:48	3883	26	263	177	301	287	104	277	191	67	59	175.2	
	17:17	5062	2449	2158	2330	2247	2375	139	704	673	452	290	1381.7	
	18:07	6638	294	478	1018	602	693	158	559	449	242	199	469.2	
	18:44	7775	1006	1627	1147	1221	938	1452	1208	1074	648	190	1051.1	
	19:01	9189	1275	1311	1180	1058	1170	688	2018	928	1270	1196	1209.4	
	19:17	9748	1263	1222	1084	1682	1534	933	1073	1427	772	547	1153.7	
Tue 16/10	15:16	3061	42	32	33	42	31	36	38	41	30	31	35.6	3429.9
	16:35	3894	93	32	70	66	101	94	161	38	172	163	99	
	17:53	4808	473	385	562	483	398	486	431	386	397	401	440.2	
	18:26	6352	602	610	778	543	460	697	574	568	451	601	588.4	
	18:47	7636	930	1041	1204	1198	1007	1022	1118	901	1052	906	1037.9	
	19:48	8990	1021	993	1305	1508	1510	1641	996	1009	1203	1102	1228.8	

Table A.6: LCI2 Dropped Requests During Scale-outs (Part1: Sat - Tue).

Date 2018	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
	15:03	2930	53	53	53	33	11	10	11	19	53	23	31.9	
	16:55	4027	47	53	72	87	47	72	18	38	14	72	52	
Wed	17:47	4884	102	85	20	57	57	64	85	21	20	56	56.7	
	18:19	5981	378	30	31	406	496	688	591	683	84	674	406.1	5838.1
17/10	18:39	6788	2322	2868	2474	2265	1826	1809	2187	1551	2319	2002	2162.3	
	19:13	8684	1788	1335	1169	1319	1515	1397	1785	1083	2172	2311	1587.4	
	19:51	10033	1589	1901	1651	1629	1642	852	1478	1565	1537	1573	1541.7	
	15:04	32	11	11	54	54	55	34	11	32	11	30.5	32	
	329	81	490	351	43	527	391	340	507	439	349.8	329	81	
	133	178	329	59	21	134	530	332	512	308	253.6	133	178	
Thur	137	90	531	55	426	134	81	487	186	169	229.6	137	90	
	1634	1305	650	1508	1225	1391	1154	1612	1050	1766	1329.5	1634	1305	5933
18/10	1037	1029	953	1514	858	1680	643	489	1217	119	953.9	1037	1029	
	1346	1666	1188	1285	1545	1320	1206	1474	1664	1566	1426	1346	1666	
	2508	1552	868	877	984	490	571	768	826	995	1043.9	2508	1552	
	14:55	10	13	55	31	32	11	32	11	68	11	27.4	10	
	16:36	78	16	51	45	51	80	30	51	45	16	46.3	78	
Fri	17:41	22	22	257	67	22	110	67	22	313	63	96.5	22	
	18:48	630	513	73	29	641	675	577	92	507	490	422.7	630	2346
19/10	19:12	844	1065	1045	912	450	1509	1061	1229	2050	1456	1162.1	844	
	19:36	184	349	514	522	398	766	781	913	547	792	576.6	184	

Table A.7: LCI2 Dropped Requests During Scale-outs (Part2: Wed - Fri).

	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Sat 13/10	13:25	3799	50	52	59	49	45	105	16	83	45	44	54.8	352.6
	17:01	5043	66	170	103	60	60	103	170	59	65	66	92.2	
	19:44	7972	105	274	204	207	35	34	368	619	105	105	205.6	
Sun 14/10	14:15	5729	154	28	151	263	436	553	273	654	232	413	315.7	1853.3
	15:43	6067	255	196	291	127	86	25	571	26	164	276	201.7	
	16:50	6988	239	656	723	744	684	99	724	981	797	898	654.5	
	18:07	8741	946	972	165	891	1207	810	158	687	150	828	681.4	
Mon 15/10	13:50	2953	21	119	103	188	1739	120	16	50	51	16	242.3	1929.5
	17:17	5062	26	263	177	301	287	104	277	191	67	59	175.2	
	18:07	6638	117	211	943	733	913	1328	600	1363	35	1257	750	
	18:36	7413	724	338	723	387	182	789	1336	1537	911	693	762	
Tue 16/10	15:16	3061	39	32	39	32	32	40	39	30	11	31	32.5	559.4
	16:35	3894	159	19	55	84	107	154	60	88	27	106	85.9	
	17:53	4808	610	31	373	390	417	380	252	405	329	80	326.7	
	18:35	6816	125	111	111	124	124	88	32	88	242	98	114.3	

Table A.8: Optimistically Refined LCI2 Dropped Requests During Scale-outs (Part1: Sat - Tue) (green rows represent the refinement occurrence).

	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Wed 17/10	15:03	2930	12	48	29	49	33	33	30	29	10	52	32.5	831.2
	16:55	4027	47	53	72	87	47	72	18	38	14	72	52	
	17:47	4884	102	85	20	57	57	64	85	21	20	56	56.7	
	18:22	5600	167	167	35	35	167	105	352	338	105	121	159.2	
	19:03	6984	852	920	740	48	164	273	329	882	244	856	530.8	
Thur 18/10	15:04	2526	32	11	11	54	54	55	34	11	32	11	30.5	936.4
	17:00	4225	329	81	490	351	43	527	391	340	507	439	349.8	
	18:16	6188	64	22	64	111	64	68	68	69	22	22	57.4	
	18:35	6820	33	33	102	160	51	33	160	91	33	103	79.9	
	19:04	7090	536	333	493	511	354	109	138	519	828	367	418.8	
Fri 19/10	14:55	2614.	10	33	33	33	33	10	55	11	31	11	26	271.5
	16:36	3945	78	16	51	45	51	80	30	51	45	16	46.3	
	18:06	5180	22	69	22	69	22	22	110	69	22	22	44.9	
	19:11	7398	99	33	92	93	34	179	104	115	154	640	154.3	

Table A.9: Optimistically Refined LCI2 Dropped Requests During Scale-outs (Part2: Wed - Fri) (green rows represent the refinement occurrence).

	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Sat 13/10	13:25	3799	50	52	59	49	45	105	16	83	45	44	54.8	352.6
	17:01	5043	66	170	103	60	60	103	170	59	65	66	92.2	
	19:44	7972	105	274	204	207	35	34	368	619	105	105	205.6	
Sun 14/10	14:15	5729	154	28	151	263	436	553	273	654	232	413	315.7	1853.3
	15:43	6067	255	196	291	127	86	25	571	26	164	276	201.7	
	16:50	6988	239	656	723	744	684	99	724	981	797	898	654.5	
	18:07	8741	946	972	165	891	1207	810	158	687	150	828	681.4	
Mon 15/10	13:50	2953	21	119	103	188	1739	120	16	50	51	16	242.3	1810.8
	17:16	4255	74	27	26	132	27	111	67	59	21	21	56.5	
	18:07	6638	117	211	943	733	913	1328	600	1363	35	1257	750	
	18:36	7413	724	338	723	387	182	789	1336	1537	911	693	762	
Tue 16/10	15:16	3061	39	32	39	32	32	40	39	30	11	31	32.5	289.5
	16:35	3894	159	19	55	84	107	154	60	88	27	106	85.9	
	17:58	4848	84	139	86	28	28	22	68	22	22	69	56.8	
	18:35	6816	125	111	111	124	124	88	32	88	242	98	114.3	
Wed 17/10	15:03	2930	12	48	29	49	33	33	30	29	10	52	32.5	836
	16:55	4027	47	53	72	87	47	72	18	38	14	72	52	
	17:53	4570	22	67	148	67	67	108	63	22	29	22	61.5	
	18:22	5600	167	167	35	35	167	105	352	338	105	121	159.2	
	19:03	6984	852	920	740	48	164	273	329	882	244	856	530.8	
Thur 18/10	15:04	2526	32	11	11	54	54	55	34	11	32	11	30.5	586.6
	17:51	4745	64	22	64	111	64	68	68	69	22	22	57.4	
	18:35	6820	33	33	102	160	51	33	160	91	33	103	79.9	
	19:04	7090	536	333	493	511	354	109	138	519	828	367	418.8	
Fri 19/10	14:55	2614.	10	33	33	33	33	10	55	11	31	11	26	271.5
	16:36	3945	78	16	51	45	51	80	30	51	45	16	46.3	
	18:06	5180	22	69	22	69	22	22	110	69	22	22	44.9	
	19:11	7398	99	33	92	93	34	179	104	115	154	640	154.3	

Table A.10: Pessimistically Refined LCI2 Dropped Requests During Scale-outs (green rows represent the refinement occurrence).

Date 2018	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Sat 13/10	13:25	3799	51	16	88	63	76	16	92	80	46	16	54.4	1748.6
	17:01	5043	154	201	60	239	22	279	210	361	259	207	199.2	
	19:34	5822	297	476	627	389	623	313	413	501	236	652	452.7	
	19:41	7484	547	652	702	1790	1467	1111	1155	1146	729	1124	1042.3	
Sun 14/10	14:15	5729	106	189	72	171	28	538	595	485	386	425	299.5	8394.9
	15:43	6067	207	225	161	272	178	276	340	503	670	611	344.3	
	16:50	6988	857	818	778	459	782	1101	833	842	764	922	815.6	
	17:59	8099	976	799	574	661	622	1128	1088	1110	1066	935	895.9	
	19:35	12414	3598	3430	3589	3427	3513	3079	3080	2978	3309	3320	3332.3	
	19:40	11129	2488	2345	2432	2619	2750	2892	2851	2948	2745	3003	2707.3	
Mon 15/10	13:50	2953	169	64	261	147	320	63	49	17	48	82	122	5727.9
	15:48	3883	276	284	273	193	207	62	95	230	265	287	217.2	
	17:17	5062	2439	2550	2233	2159	2373	680	197	827	549	923	1493	
	18:07	6638	480	396	441	389	687	308	346	260	34	386	372.7	
	18:44	7775	1160	1059	1208	1192	1143	993	993	867	1097	1206	1091.8	
	19:01	9189	1332	1147	1332	1189	1314	1065	1312	1310	857	1360	1221.8	
	19:17	9748	1275	1311	1180	1058	1170	688	2018	928	1270	1196	1209.4	
Tue 16/10	15:16	3061	39	38	39	42	32	40	60	39	11	32	37.2	4045.2
	16:35	3894	186	33	68	84	107	159	93	181	37	174	112.2	
	17:53	4808	610	371	573	490	417	480	452	405	429	439	466.6	
	18:26	6352	613	778	606	543	472	730	555	534	469	534	1107.8	
	18:47	7636	917	1060	1190	1246	1004	1013	1117	907	1084	903	1044.1	
	19:48	8990	963	1029	1352	1607	1463	1751	1055	1119	1244	1190	1277.3	

Table A.11: LCI3 Dropped Requests During Scale-outs (Part1: Sat - Tue).

Date 2018	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
	15:03	2930	12	48	29	49	33	33	30	29	10	52	32.5	
	16:55	4027	40	44	22	22	16	38	135	110	209	145	78.1	
	17:47	4884	124	166	65	217	243	147	343	218	360	364	224.7	
Wed 17/10	18:19	5981	489	97	609	606	463	823	327	685	300	480	487.9	6953.3
	18:39	6788	2876	2241	3337	2877	2644	1917	2134	2018	1826	1888	2375.8	
	19:13	8684	1506	1350	1283	1444	1670	2343	2350	2624	2481	2322	1937.3	
	19:51	10033	1617	1526	2126	1496	1508	1877	2253	1805	1929	2033	1817	
	15:04	2526	33	11	34	11	34	36	11	11	32	11	22.4	
	17:00	4225	538	461	456	468	521	350	361	569	302	339	436.5	
	17:44	5240	130	506	308	225	416	509	437	362	60	449	340.2	
Thur 18/10	18:13	5910	434	668	238	147	451	136	337	293	220	1010	393.4	6716
	18:35	6820	1620	1485	1721	1753	1716	1560	1338	1601	1346	1703	1584.3	
	19:02	9039	1357	1411	1655	1315	1002	1129	1335	1186	846	921	1215.7	
	19:11	10470	1677	1624	1943	1729	1624	1535	1654	1314	1438	1194	1573.2	
	19:22	11124	1220	1299	1327	1107	1094	1098	1344	924	1039	1051	1150.3	
	14:55	2614	10	33	33	33	33	10	55	11	31	11	26	
	16:36	3945	46	16	140	226	16	148	347	250	73	318	158	
Fri 19/10	17:41	5180	324	69	70	257	110	301	103	202	63	183	168.2	3321
	18:48	6677	615	970	917	862	685	806	961	772	675	740	800.3	
	19:12	7981	1165	1210	859	1395	1487	1770	1397	1670	1223	1180	1335.6	
	19:36	10093	794	904	610	925	358	1143	1130	670	827	965	832.6	

Table A.12: LCI3 Dropped Requests During Scale-outs (Part2: Wed - Fri).

	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Sat 13/10	13:25	3799	51	16	88	63	76	16	92	80	46	16	54.4	509.5
	17:01	5043	154	201	60	239	22	279	210	361	259	207	199.2	
	19:44	7972	35	94	303	166	356	537	494	106	433	35	255.9	
Sun 14/10	14:15	5729	106	189	72	171	28	538	595	485	386	425	299.5	2731.7
	15:43	6067	207	225	161	272	178	276	340	503	670	611	344.3	
	16:50	6988	857	818	778	459	782	1101	833	842	764	922	815.6	
	18:07	8741	1253	1298	888	1307	996	51	1670	1720	1750	1790	1272.3	
Mon 15/10	13:50	2953	169	64	261	147	320	63	49	17	48	82	122	2202.3
	17:17	5062	276	284	273	193	207	62	95	230	265	287	217.2	
	18:07	6638	2449	2158	2330	2247	2375	139	704	673	452	290	932.6	
	18:36	7413	587	963	797	396	596	868	1078	1375	1323	1322	930.5	
Tue 16/10	15:16	3061	39	32	39	32	32	40	60	39	11	32	35.6	673.5
	16:35	3894	159	19	55	84	107	154	83	141	27	164	99.3	
	17:53	4808	610	31	373	390	417	480	452	405	429	439	402.6	
	18:35	6816	124	40	195	124	125	156	99	366	32	99	136	
Wed 17/10	15:03	2930	53	53	53	33	11	10	11	19	53	30	32.6	1285.6
	16:55	4027	40	44	22	22	16	38	135	110	209	145	78.1	
	17:47	4884	124	166	65	217	243	147	343	218	360	364	224.7	
	18:22	5600	98	452	247	168	147	175	706	99	704	705	350.1	
	19:03	6984	44	120	134	707	295	1016	909	1035	994	747	600.1	

Table A.13: Optimistically Refined LCI3 Dropped Requests During Scale-outs (Part1: Sat - Wed) (green rows represent the refinement occurrence).

	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Thur 18/10	15:04	2526	33	11	34	11	34	36	11	11	32	11	22.4	1487.2
	17:00	4225	538	461	456	468	521	350	361	569	302	339	436.5	
	18:16	6188	130	506	308	225	416	509	437	362	60	449	340.2	
	18:35	6820	141	226	35	56	161	504	467	282	103	604	257.9	
	19:04	7090	692	569	622	201	436	255	455	259	371	442	430.2	
Fri 19/10	14:55	2614	10	33	33	33	33	10	55	11	31	11	26	586.1
	16:36	3945	46	16	140	226	16	148	347	250	73	318	158	
	18:06	5180	23	112	37	22	63	96	64	71	111	111	71	
	19:11	7398	100	33	99	93	159	979	1127	222	341	158	331.1	

Table A.14: Optimistically Refined LCI3 Dropped Requests During Scale-outs (Part2: Thurs - Fri) (green rows represent the refinement occurrence).

Date 2018	Time	Input Req Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total
Sat 13/10	13:25	3799	51	16	88	63	76	16	92	80	46	16	54.4	370.4
	17:01	5043	60	21	66	21	21	109	66	66	67	104	60.1	
	19:44	7972	35	94	303	166	356	537	494	106	433	35	255.9	
Sun 14/10	14:15	5729	106	189	72	171	28	538	595	485	386	425	299.5	2731.7
	15:43	6067	207	225	161	272	178	276	340	503	670	611	344.3	
	16:50	6988	857	818	778	459	782	1101	833	842	764	922	815.6	
	18:07	8741	1253	1298	888	1307	996	51	1670	1720	1750	1790	1272.3	
Mon 15/10	13:50	2953	169	64	261	147	320	63	49	17	48	82	122	2041.9
	17:16	4255	84	131	132	29	26	25	72	21	22	26	56.8	
	18:07	6638	1121	846	211	1066	932	97	1303	1210	1280	1260	932.6	
	18:36	7413	587	963	797	396	596	868	1078	1375	1323	1322	930.5	
Tue 16/10	15:16	3061	39	32	39	32	32	40	60	39	11	32	35.6	333.5
	16:35	3894	159	19	55	84	107	154	83	141	27	164	99.3	
	17:58	4848	139	28	29	85	85	114	66	23	35	22	62.6	
	18:35	6816	124	40	195	124	125	156	99	366	32	99	136	
Wed 17/10	15:03	2930	53	53	53	33	11	10	11	19	53	30	32.6	1128.8
	16:55	4027	40	44	22	22	16	38	135	110	209	145	78.1	
	17:53	4570	67	103	67	67	109	81	72	22	68	23	67.9	
	18:22	5600	98	452	247	168	147	175	706	99	704	705	350.1	
	19:03	6984	44	120	134	707	295	1016	909	1035	994	747	600.1	
Thur 18/10	15:04	2526	33	11	34	11	34	36	11	11	32	11	22.4	781.9
	17:51	4745	23	23	68	65	70	119	96	115	70	65	71.4	
	18:35	6820	141	226	35	56	161	504	467	282	103	604	257.9	
	19:04	7090	692	569	622	201	436	255	455	259	371	442	430.2	
Fri 19/10	14:55	2614	10	33	33	33	33	10	55	11	31	11	26	586.1
	16:36	3945	46	16	140	226	16	148	347	250	73	318	158	
	18:06	5180	23	112	37	22	63	96	64	71	111	111	71	
	19:11	7398	100	33	99	93	159	979	1127	222	341	158	331.1	

Table A.15: Pessimistically Refined LCI3 Dropped Requests During Scale-outs (green rows represent the refinement occurrence).

A.2 Detailed Experiment Tables for Traffic Bursts Scenario

The following tables list the detailed experiment results for all 10 repetitions. These tables show tested time and traffic bursts injected into the ISP's CDN dataset and the number of dropped requests. Tables A.16 and A.17 list the number of dropped requests in Baseline 1 and Baseline 2 respectively. LCI 2, its earlier scaling refinement and vertical upgrade refinement results are listed in Tables A.18, A.19 and A.20.

	Incoming Request Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total Drops
Traffic Burst 1	3235	1325	1188	44	45	8	1055	871	1017	17	1016	659.1	35634.3
	7132	4953	4964	1257	3006	902	4955	4847	4995	440	4723	3504.7	
	8099	5888	5885	1701	4503	2217	5893	5827	2037	496	2807	3725.9	
	9409	7206	7190	2978	5379	2989	8112	7128	193	2280	919	4437.8	
	6881	4683	4661	2953	3108	3102	5217	4576	304	194	41	2884.4	
	4129	1245	1883	411	404	530	1914	1883	568	442	117	940.2	
Traffic Burst 2	4794	399	91	89	2561	293	2527	89	2537	400	18	900.6	
	6728	2383	26	127	4484	2179	4471	126	4451	1423	1644	2131.8	
	7760	3208	95	146	5501	2471	5537	145	5477	873	1462	2492.1	
	8015	1906	158	31	5754	1742	5795	33	5751	549	1334	2305.8	
	5442	21	101	21	3182	101	1915	21	21	101	101	558.8	
Traffic Burst 3	7578	1284	1292	1264	37	1034	1272	1148	143	2632	1294	1140.4	
	8501	2215	2376	2785	270	915	2133	2029	162	5220	2286	2039.6	
	10376	3292	4019	6049	1824	1756	3878	2959	109	7225	6231	3734.7	
	9195	703	3054	5376	47	299	2739	594	173	6175	5576	2473.9	
	8076	32	1818	4108	53	152	1719	43	36	4775	4303	1704.5	

Table A.16: Baseline1 (GKE Autopilot) Dropped Requests During Traffic Bursts.

	Incoming Request Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total Drops
Traffic Burst 1	3235	44	45	8	45	8	8	45	45	8	8	27.1	7835.2
	7132	1460	1508	1603	1734	1553	1828	1942	2061	2004	2019	1771.8	
	8099	1640	2438	2104	2632	2496	2862	2884	2390	3024	2024	2449.9	
	9409	1406	2003	1740	1625	3082	2175	2175	2337	2439	2239	2122	
	6881	20	97	97	20	20	97	20	97	97	20	59	
	4129	58	11	11	11	11	59	58	11	11	58	30.5	
Traffic Burst 2	4794	13	67	67	13	67	13	13	13	14	67	35.3	
	6728	19	95	19	19	94	19	20	19	21	95	42.6	
	7760	69	252	161	22	22	495	22	549	23	446	206.7	
	8015	113	23	113	113	113	113	23	113	23	23	77.6	
	5442	77	77	77	15	15	15	15	15	76	77	46	
Traffic Burst 3	7578	22	22	22	22	22	108	107	22	107	22	48.1	
	8501	120	25	25	120	25	120	25	25	25	120	63	
	10376	966	941	839	629	989	389	280	1179	919	30	716	
	9195	130	27	27	27	129	27	27	129	129	27	68.5	
	8076	24	23	23	114	24	114	24	115	114	114	69.4	

Table A.17: Baseline2 (Fully-Managed GKE) Dropped Requests During Traffic Bursts.

	Incoming Request Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total Drops
Traffic Burst 1	3235	45	45	8	8	8	8	8	8.	45	8	19	19233.7
	7132	2133	2048	2106	2150	2159	2696	2530	2436	2621	2448	2333	
	8099	3116	3159	3063	2807	2931	3161	3032	3424	3378	3416	3149	
	9409	4130	3489	3331	3378	3358	3913	3743	3198	3894	3774	3621	
	6881	829	794	968	971	872	1428	1281	370	1265	1190	997	
	4129	11	58	58	58	11	11	58	58	58	11	39.6	
Traffic Burst 2	4794	13	4780	67	13	13	13	67	13	13	120	511.8	
	6728	907	5950	821	845	806	723	692	1080	998	1129	1395.5	
	7760	1341	6457	1509	1283	1921	1688	1685	1888	1431	1322	2052.9	
	8015	1060	7003	1110	178	686	911	25	1391	1271	1287	1492.5	
	5442	77	5364	15	15	15	77	15	77	77	77	581.5	
Traffic Burst 3	7578	86	107	22	22	22	22	22	22	22	109	46	
	8501	832	734	748	690	679	699	565	967	954	1034	790.5	
	10376	1619	2598	2547	2215	2105	1893	2256	1954	1130	2194	2051	
	9195	27	130	27	27	129	354	27	130	27	27	90.9	
	8076	115	115	23	23	114	23	23	114	23	23	60.3	

Table A.18: LCI2 Dropped Requests During Traffic Bursts.

	Incoming Request Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total Drops
Traffic Burst 1	3235	45	8	45	8	45	8	45	44	45	8	30.7	6076.9
	7132	1049	1304	1089	1091	1584	1685	1896	1684	1439	1559	1438.5	
	8099	1440	1483	1106	1367	2021	2715	2455	1663	1350	1887	1749.1	
	9409	1388	453	720	354	1763	1520	1805	1972	1803	1250	1303.2	
	6881	96	20	98	20	97	97	97	20	97	20	66.7	
	4129	11	58	58	11	11	11	57	11	58	58	34.9	
Traffic Burst 2	4794	67	67	67	13	13	13	13	13	13	13	29	
	6728	24	19	124	20	19	117	150	41	213	608	133	
	7760	26	22	109	688	32	310	535	227	22	1662	363.9	
	8015	115	23	113	114	113	23	24	23	23	113	68.9	
	5442	16	77	15	77	77	15	136	15	15	15	46.4	
Traffic Burst 3	7578	107	107	22	22	107	22	22	22	106	22	56.4	
	8501	25	120	25	26	120	25	120	119	133	120	83.8	
	10376	202	978	446	367	1830	32	34	1346	32	31	530.2	
	9195	27	129	27	130	28	138	27	27	39	130	70.7	
	8076	114	113	24	24	24	24	115	23	115	113	69.3	

Table A.19: Refined LCI2 (By Earlier Scaling) Dropped Requests During Traffic Bursts.

	Incoming Request Rate/Min	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG	Total Drops
Traffic Burst 1	3235	45	45	8	19	8	8	19	19	34	8	22	15814.2
	7132	1931	1886	1955	2025	1977	2435	2353	2324	2436	2319	2164.7	
	8099	2673	2942	2775	2754	2801	3071	2987	3114	3272	2998	2939.3	
	9409	3312	3043	2853	2852	3275	3392	3273	2939	3457	3314	3171.6	
	6881	586	585	707	686	616	1029	902	288	915	839	715.7	
	4129	25	44	44	44	11	25	58	44	44	25	36.8	
Traffic Burst 2	4794	13	3366	67	13	29	13	51	13	13	104	368	
	6728	640	4194	581	597	592	512	490	762	705	819	989.6	
	7760	960	4596	1105	905	1351	1330	1186	1487	1008	1059	1499.1	
	8015	776	4909	811	158	514	671	24	1008	896	908	1068	
	5442	77	3778	34	15	15	58	15	58	76	77	420.9	
Traffic Burst 3	7578	66	81	22	22	22	48	47	22	47	83	46.6	
	8501	618	521	531	519	482	525	403	684	675	760	572.3	
	10376	1423	2101	2034	1739	1770	1442	1663	1722	1067	1545	1651.1	
	9195	58	99	27	27	129	256	27	130	58	27	84.2	
	8076	87	87	23	51	87	50	24	114	51	51	63.1	

Table A.20: Refined LCI2 (By Vertical Upgrade) Dropped Requests During Traffic Bursts.

A.3 Snapshots for CP and CCDN interaction

```
Please enter your request:
 {'service': 'caching', 'resources' : 'London', 'conjunction' : 'with',
'target' : 'low cost'}


Your suggested CDN deployment is:
Cache Server Size:  Small
Zone: London
Performance improvement Level: Medium


Do you wish to continue? Yes / No / Update
 Yes

Your CDN has been deployed and it is  ready!
```

Figure A.1: CP's Approval on Suggested CCDN Deployment.

```
Please enter your request:
 {'service': 'caching', 'resources' : 'London', 'conjunction' : 'with',
'target' : 'low cost'}


Your suggested CDN deployment is:
Cache Server Size:  Small
Zone: London
Performance improvement Level: Medium


Do you wish to continue? Yes / No / Update
 No

Would you like to update your request? Yes / No
 No

Your CDN request has been cancelled!
```

Figure A.2: CP's Rejection of Suggested CCDN Deployment.

```
Please enter your request:
 {'service': 'caching', 'resources' : 'London', 'conjunction' : 'with',
'target' : 'low cost'}


Your suggested CDN deployment is:
Cache Server Size:  Small
Zone: London
Performance improvement Level: Medium


Do you wish to continue? Yes / No / Update
 Update

Please enter your updated request:
 {'performanceImprovement': 'Low'}


Your suggested CDN deployment is:
Cache Server Size:  Small
Zone: London
Performance improvement Level: Low
```

Figure A.3: CP's Update on Suggested CCDN Deployment.

```
Your suggested performance improvement is:
Improvement action: Cache size upgrade during peak hours
Upgraded cache  size: Medium
Peak hours start: 17:00
Peak hours end: 20:00

Do you wish to proceed with this performance improvement? Yes / No / Update
 Update



Please enter your performance improvement:
 {'peakStart': '18:30', 'peakEnd': '21:00'}



Your suggested performance improvement is:
Improvement action: Cache size upgrade during peak hours
Upgraded cache  size: Medium
Peak hours start: 18:30
Peak hours end: 21:00
```

Figure A.4: CP's Update on Suggested Performance Improvement.