# Intent-Based Service Graph Generation and Selection for Cost-Effective Service Deployment in the Cloud

**Sasan Sabour**

**A Thesis**

**In**

**the Department**

**of**

**Electrical and Computer Engineering**

**Presented in Partial Fulfillment of the Requirements**

**For the Degree of**

**Master of Applied Science (Electrical and Computer Engineering) at**

**Concordia University**

**Montréal, Québec, Canada**

**July 2025**

**© Sasan Sabour, 2025**

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Sasan Sabour**

Entitled: **Intent-Based Service Graph Generation and Selection for Cost-Effective Service Deployment in the Cloud**

and submitted in partial fulfillment of the requirements for the degree of

**Master of Applied Science (Electrical and Computer Engineering)**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
*Dr. Rodolfo Coutinho*

_____ Internal Examiner
*Dr. Rodolfo Coutinho*

_____ Internal Examiner
*Dr. Ferhat Khendek*

_____ Supervisor
*Dr. Roch Glitho*

Approved by _____
Dr. Yousef R. Shayan
Chair of Department Electrical and Computer Engineering

Date of Defence: July 16 , 2025

_____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

# Abstract

Intent-Based Service Graph Generation and Selection for Cost-Effective Service Deployment in the Cloud

Sasan Sabour

Cloud computing provides a wide range of virtualized services, such as computing power, storage, and applications, which can be accessed on-demand. Cloud-native applications are built using a microservices architecture, where independent, function-specific services communicate through APIs and messaging protocols. Service graph is a directed graph that shows interaction and dependencies between these services. An application can be achieved by using different services with same functionality which result in having multiple service graphs. However, selecting an optimal service graph among all possible ones is a challenging task that requires expertise in cloud applications, their dependencies, and compatibility, along with meet non-functional requirements such as bandwidth, and latency.Tradition approaches rely on domain experts who manually generate service graphs, a time-consuming and error-prone approach that often results in suboptimal configurations, higher deployment costs, and unmet performance criteria. These challenges highlight the need for automated solutions to efficiently produce optimal service graphs aligned with cloud consumers expectations.

Intent-Based Networking (IBN) is a new paradigm that allows cloud consumers to specify their requirements at a high level, without dealing with the underlying technical complexities. Intent is a request at a high-level of abstraction (e.g., Natural Language), which describes what the cloud consumers expect. More specifically, it enables cloud consumers to focus on defining ”what” they need, such as performance targets, cost constraints, or latency requirements, without needing to specify ”how” these objectives should be achieved. Instead of manually selecting and configuring services, cloud consumers express their intent in terms of objectives such as performance targets, cost constraints, or latency requirements. Then, these high-level intents translates into low-level configurations, automatically generating and deploying the service graph in the cloud.

In this thesis, we address the problem of service graph generation and selection while considering both functional and non-functional requirements derived from cloud consumer intent. Our objective is to minimize the total deployment cost of the service graph in a data center network and to determine its placement within the distributed data center network. To address the problem, first, we translate high-level intent using a domain ontology into its functional and

non-functional requirements which are specified in terms of initial services, latency, and bandwidth. Then, we use a service catalog along with the initial services to generate all possible service graphs that can meet the functional requirements of the given intent. We formulate the problem as an Integer Linear Programming (ILP), taking into account the bandwidth and latency requirements of the intent. To solve this, we propose our Service Graph Selection (SGS) solution, which aims to achieve a near-optimal solution in a computationally efficient manner. Our results demonstrate that the proposed solution achieves a deployment cost that is only 4-6% larger than the lower bound of the optimal deployment cost.

# Acknowledgments

First and foremost, I am deeply grateful to my supervisor, Dr. Roch Glitho, for his invaluable guidance, unwavering encouragement, and insightful feedback throughout my research journey. His patience and support have been instrumental in shaping this work and fostering my development as a researcher. I am truly indebted to him for his patience and understanding.

I would like to express my sincere gratitude to Dr. Ferhat Khendek and Dr. Rodolfo Coutinho for serving as the committee members for my thesis.

I would also like to express my sincere gratitude to Ericsson for their support throughout this research. I am especially thankful to Dr. Mbarka Soualhia and Dr. Fetahi Wuhib for their generous time, thoughtful guidance, and valuable contributions. Their support has greatly enriched this work and made the research experience truly rewarding.

I am profoundly thankful to Dr. Amin Ebrahimzadeh for his invaluable mentorship, continuous support, and insightful suggestions, which significantly contributed to the development of this thesis.

I also wish to extend my appreciation to the members, colleagues, and friends at the Telecommunication Service Engineering (TSE) research lab for their encouragement and support. A special thanks to Vahid Maleki Raee for his kind support, encouragement, and the insightful discussions we have always shared.

I gratefully acknowledge Concordia University and the Department of Electrical and Computer Engineering for providing the academic environment, resources, and support necessary to conduct this research.

Last but not least, I would like to extend my heartfelt gratitude to my parents and family for their unwavering love, care, and support. No words can truly express my appreciation and love for you. None of my achievements or dreams would have been possible without you. I am also deeply thankful to my dear brother, Ehsan Sabour, and his wife, Sajedeh Khaledi, for their constant encouragement, support, and positive energy throughout my studies.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction [1]

This chapter starts with an overview of the essential terminologies relevant to our research. Following that, we delve into the motivation and problem statement for this study. We then provide a summary of the contributions made through this work. Lastly, we outline the structure of the remaining chapters in this thesis.

## 1.1. Defnitions

This section is dedicated to providing definitions of the key terms that form the foundation of this thesis. These definitions encompass critical concepts such as Intent-Based Networking, Service Graphs, and Cloud Computing.

### 1.1.1 Intent-Based Networking

Intent-Based Networking (IBN) is a new concept that allows network administrators or operators to express requirements in natural language to the network [1, 2]. IBN has gained significant interest due to its aim to simplify network management for human operators [3]. This is done by allowing network administrators to express WHAT should be done rather than HOW it should be done. For example, in a cloud environment, a cloud consumer can express WHAT he wants, "*I want to have a holographic conference between Concordia University and Ericsson*", instead of specifying HOW should have a holographic conference, "*Use a holographic encoder in the source (Concordia University) and decoder in the destination (Ericsson) to have a holographic conference*". The example implies the difference between IBN and traditional networks where the network administrator in IBN just specifies what she needs without going into configuration details (e.g., required services, bandwidth, and latency).

[1] This Thesis is based on three published papers: [1] Sabour S, Ebrahimzadeh A, Soualhia M, Wuhib F, Glitho RH. Intent- based Service Graph Selection for Cost-Effective Cloud Deployment. In 2025 IEEE 28th Conference on Innovation in Clouds, Internet and Networks (ICIN), [2] Sabour S, Ebrahimzadeh A, Wuhib F, Soualhia M, Glitho RH. Service Graphs Generation in Intent-Based Networks. In IEEE 21st Consumer Communications Networking Conference (CCNC) 2024 Jan 6 (pp. 90-97), and [3] W O2024201109A1, Method for generating service graphs from high-level intents, Sasan Sabour, Fetahi Wuhib, Amin Ebrahimzadeh, Roch Glitho, publication date: 03 October 2024.

### 1.1.2 Service Graph

Services are self-contained, technology-neutral computing units that enable the quick and cost-effective development of web applications in distributed systems [4]. Service-oriented architecture (SOA) structures software by offering independent, reusable, and automated services within a secure framework. A key evolution of SOA is the microservices architecture, which breaks down monolithic applications into loosely coupled, fine-grained services that communicate via lightweight protocols. A service graph is a directed graph that comprises nodes and links, where the nodes denote services/microservices, and the links define the connections between these services. Unlike VNF-Forwarding Graphs (VNF-FGs), which are limited to VNFs, service graphs generalize the concept to include a broader range of services, offering a flexible framework for defining and deploying complex service compositions.

### 1.1.3 Cloud Computing

Cloud computing is a model that provides on-demand network access to a shared pool of configurable computing resources, including data storage, networks, servers, applications, and services [5]. It "refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services. The services themselves have long been referred to as Software as a Service (SaaS). The datacenter hardware and software is what we will call a Cloud" [6].

Cloud computing includes three main service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS provides users with access to computing resources such as servers, networking, and storage. PaaS offers users software development environments to build and deploy their own applications. SaaS delivers specific applications or software to users, allowing them to utilize these services directly.

## 1.2. Motivation and Problem Statement

Over the past decade, cloud computing has evolved into a transformative paradigm, offering a wide range of virtualized services [7]. In cloud environments, service providers, such as IaaS or PaaS providers, offer computing, storage, and networking resources that enable cloud consumers to deploy services based on specific requirements of their applications. A cloud-native application is a distributed and scalable solution designed to take full advantage of cloud computing [4]. It is composed of independent microservices, which communicate with each other to deliver functionality. These applications are designed to be elastic and horizontally scalable, meaning they can easily adapt to varying workloads. A Service Graph (SG) represents the logical composition of microservices, where nodes correspond to the services and links represent their connections.A proper SG is essential to ensure that performance (e.g., bandwidth,

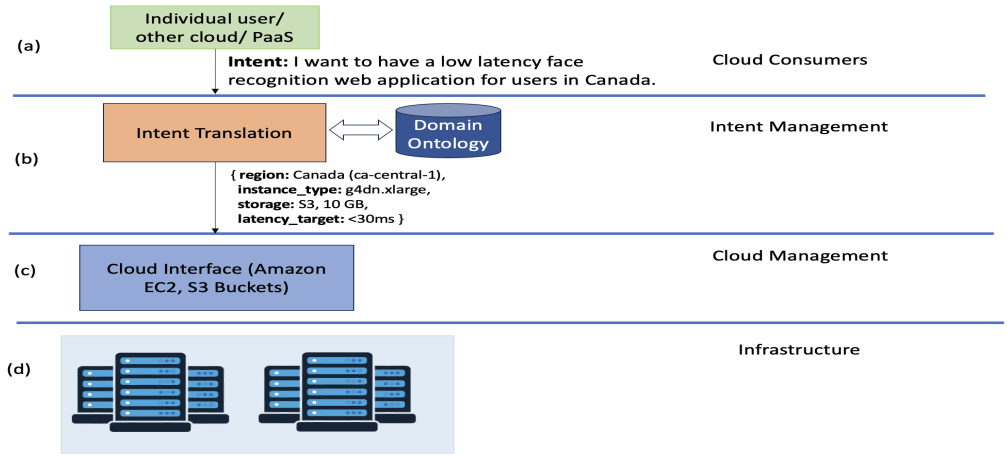



Figure 1.1: Intent-based cloud management [8]. (a) Cloud consumers express their requests through high-level intent.(b) The intent management layer translates the intent into required services and configurations using a domain ontology.(c) The cloud management layer creates, controls, and monitors virtualized resources on an Infrastructure-as-a-Service (IaaS) platform.(d) The infrastructure layer manages the physical hosts and datacenters.

latency), scalability, reliability, and other requirements of applications are met. However, there may be multiple microservices with same functionality, leading to the creation of several service graphs for a single application.Therefore, navigating through the multitude of available service graphs can be complex and challenging. An expert with extensive networking knowledge is typically required to select and compose the services among hundreds of possibilities (e.g., each input intent can correspond to hundreds of different service graphs that can implement the intent) while considering the specific application requirements. This process can be slow and may not always meet all the application requirements. For example, in figure 1.1, let us consider a cloud consumer (e.g., Individual user, other cloud or PaaS) wants to have a low latency face recognition web application for users in Canada [8]. He would need a deep understanding of face recognition web applications, required services, and how these services should be connected as a service graph to have the face recognition. There may exist different service graphs and the challenge is to select the optimal one that not only satisfies the requirements defined in the intent (e.g., bandwidth, latency) but also complies with the infrastructure deployment constraints to ensure successful execution of the application.

One approach to address this complexity is to allow cloud consumer to specify her requirements as an intent, which, at a high level of abstraction, defines WHAT needs to be done (e.g., I want to have a high-quality connection between A and B) rather than HOW it should be done (e.g., connect A and B using path x if bandwidth $<$ 500MB, otherwise use path y). To understand the given intent, its functional and non-functional requirements should be extracted. Functional Requirement (FR) is a user request (e.g., have a face recognition web application) and Non-Functional Requirement (NFR) is the quality of the request (e.g., quality or tolerable latency). Cloud consumers with limited cloud knowledge can specify their desired use-case (e.g., holographic communication) through intent in plain language, and it would

be translated into cloud configurations. In Figure 1.1, the cloud consumer's intent is translated into cloud configurations through intent translation and domain ontology. The domain ontology contains domain-specific knowledge that facilitates the accurate interpretation of user intent. The translated configuration outlines the necessary resources for deploying a face recognition web application, for instance, the geographic region for service deployment, GPU-enabled instance types to support machine learning workloads, and specific storage and latency requirements. Then, the cloud management layer select the required services and resources, which are subsequently deployed in the infrastructure.

In cloud services, an SG is essential to express cloud consumer intent effectively. Services in an SG are organized based on the intent, including FRs and NFRs, and each SG contains a specific order of services. Traditionally, this graph is manually provided by experts with prior knowledge of the various services, their expected performances, and the data centers where the services will be deployed. This process can be time-consuming and often slow. It also may generate sub-optimal results that cannot fulfill the intent requirements entirely.

In this thesis, we address the problem of service graph generation and selection using high-level intent while considering FRs and NFRs derived from high-level intent. We first generate all possible service graphs considering FRs and then select one cost-effective service graph while satisfying NFRs and infrastructure deployment constraints. The extraction of FRs and NFRs from high-level intent can be achieved through domain ontology. The input of the problem is translated intent containing initial services and bandwidth and latency requirements obtained from a domain ontology and a service catalog which includes information about services and their expected input/output contents. The output is a cost-effective service graph that can satisfy the given FRs and NFRs derived from user intent.

## 1.3. Thesis Contributions

The main contributions of this thesis can be summarized as follows:

- We derive the requirements of IBN via an illustrative use-case with the aim of addressing the service graph generation and selection problems while considering FRs and NFRs captured from high-level intent.
- We formulate the problem of service graph selection as an Integer Linear Programming (ILP) and then propose a solution to solve the ILP efficiently.
- We solve the problem of service graph generation and selection by proposing a solution, which translates high-level intent to low-level intent, extracts FRs and NFRs, and selects the cost-effective service graph among all possible generated service graphs while considering infrastructure deployment constraints and determining its placement within the distributed data center network in an automated manner.
- We evaluate the performance of our proposed solution in service catalogs with different sizes and content diversities, and compare the results with Service Function Graph Construction (SFGC) [9] and a naive algorithm.

Moreover, We evaluate the performance of our proposed solution using the Holographic Extended Reality (HXR) use-case in different data center sizes with diverse NFRs, and compare the results with lower-bound optimal cost approach.

## 1.4. Thesis Organization

The structure of this thesis is organized as follows: Chapter 2 provides an overview of the key concepts and foundational knowledge relevant to this research. In Chapter 3, we introduce a motivating scenario to illustrate the intent-based service graph generation and selection problem and derive the associated requirements. This is followed by a comprehensive evaluation of existing literature, analyzing how well it addresses the identified requirements. Chapter 4 details the proposed solution for service graph generation and selection. Chapter 5 outlines the evaluation environment, including the setup, parameter configurations, and programming tools employed, and presents the results of the evaluation. Finally, Chapter 6 concludes the thesis by summarizing the contributions of this work and highlighting potential directions for future research.

# Chapter 2

# Background Information

This chapter introduces the background concepts relevant to the research domain of this thesis. The first section defines intent-based networking, starting with an overview of autonomic networks and the concept of intent, along with related terminologies. It further explores intent-based networking, its key components, and presents a taxonomy of intent expression and translation methods. The second section focuses on cloud computing, outlining its essential characteristics, describing the layers of cloud computing, and exploring the major cloud deployment models. This section also includes a discussion on Amazon Elastic Compute Cloud (EC2), which delivers on-demand, scalable computing capacity within the Amazon Web Services (AWS) Cloud. The third section introduces service graphs and their associated concepts. The fourth section provides an overview of domain ontology. The fifth section examines the service catalog and its defining characteristics. Finally, the sixth section delves into holographic communication, which serves as the use case in the following chapter.

## 2.1. Intent-Based Networking (IBN)

This section presents the background information and fundamental technical concepts related to the domain of Intent-Based Networking (IBN). It covers autonomic networks, the concept of intent, and its distinction from policy. Additionally, it discusses IBN, its key components, and presents a taxonomy of intent expression and translation methods.

### 2.1.1 Autonomic Networks

The requirement for an autonomic network to automatically configure itself based on user requests is not something that has been raised recently [1]. In early 2001, IBM proposed an innovative vision of computing systems that are

autonomic [10]. The objective of this vision was to have a network that has self-management properties including self-configuration, self-optimization, self-protection, and self-healing with the capability of reconfiguring without human intervention [11].

Autonomic is defined as self-managing with high-level guidance through intent [12]. However, autonomous has a broader definition and means to operate independently without external control. On the other hand, automatic is "a process that occurs without human intervention, with step-by-step execution of rules" [12]. It's important to note that automation alone does not equate to autonomic functionality. For instance, a network administrator might create a script to automate router configurations, such as assigning IP addresses. While this script can automate the configuration process, it is not autonomic since it still requires human intervention to define the sequence of commands and settings. The primary objective of autonomic networks is self-management, which encompasses various "self" attributes. Among the most frequently mentioned are:

- **Self-configuration**: It is the ability to operate without the need for configuration by either an administrator or a management system. Instead, it does configurations based on utilizing self-knowledge, discovery, and intent [12].
- **Self-healing**: It involves adjusting to changes in the environment and automatically resolving issues [12].
- **Self-optimizing**: It automatically identifies methods to optimize the performance according to a set of well-defined objectives [12].
- **Self-protection**: It automatically protects the network from potential security threats [12].

### 2.1.2 Intent

The concept of "intent" originated in the context of autonomic networks, where it refers to "An abstract, high-level policy used to operate the network" [12]. Based on the definition, an intent is a specific type of policy provided by a user to guide the autonomic network, which would otherwise operate without human intervention. However, intent and policy are not identical or interchangeable terms. Internet Research Task Force (IRTF) defines a policy as "A set of rules that governs the choices in behavior of a system" [13]. Similarly, intent is defined as "A set of operational goals (that a network should meet) and outcomes (that a network is supposed to deliver) defined in a declarative manner without specifying how to achieve or implement them" [13].

### 2.1.3 Distinguishing between Intent and Policy

Network policies are often confused with user intents, however, they provide more detailed specifications for network configuration compared to the high-level and abstracted nature of intent. The policy comprises a set of rules,

usually structured as Events-Conditions-Actions (ECA) [13]. Intent represents a declarative goal at a high level, without specifying how it should be satisfied.

To better understand the differences between policy and intent, let's consider a scenario where a cloud consumer wants to have a holographic conference between Concordia University and Ericsson. In this scenario, a policy might state, "*Utilize a holographic encoder and decoder for a holographic conference between Concordia University and Ericsson. Allocate the holographic encoder to Datacenter1 and the decoder to Datacenter2. If the bandwidth on the connection between Concordia and Ericsson drops below 1Gb/s, switch to Datacenter3 for the decoder.*" On the other hand, an intent could be, "*I would like to have a high-quality holographic conference between Concordia University and Ericsson.*" As demonstrated, in a policy, the cloud consumer must have good knowledge of the required services for having a holographic conference. Additionally, he needs to specify policies to determine what actions should be taken if the bandwidth drops below 1Gb/s. However, in an IBN framework, the cloud consumer can simply express the aforementioned intent. The IBN automatically translates and generates a corresponding network policy, eliminating the need for predefined policies within the network. The example illustrates the difference: intent is what the user wants (holographic conference with high quality), while a network policy details how to achieve it through configurations and actions based on events and conditions.

Similar to intent, policies provide a high level of abstraction by capturing user requirements through rules that define triggers for specific actions. However, unlike intent, users are required to describe the details of what should be done through rules and actions. If conflicts arise within or between policies, resolving them may require user interaction or external logic outside of policy-based management [14]. On the other hand, if conflicts occur within or between intents, IBN has an intent resolution component to automatically address conflicts [1].

### 2.1.4 Intent-Based Networking

Intent-Based Networking seeks to simplify network operations, requiring minimal external intervention. It allows network administrators to specify their requirements in a high-level of abstraction as an intent. Intent provides declarative goals without specifying implementation details [13]. To better understand what constitutes an intent and what cannot be considered as an intent, we provide examples for both. For the sake of clarity, the examples are expressed in natural language. The following is an intent: "*Ensure VPN services have path protection at all times for all paths.*" [13], because there is a desired outcome without precisely specifying how it can be achieved. In contrast, the following is not an intent: "*Configure a VPN with a tunnel from A to B over path P.*" [13], because it would be considered a configuration of a service.

### 2.1.5 IBN Components

While IBN is a relatively new term and technology, substantial efforts have been made to define and standardize it. Based on [1], the complete process, from user intent expression to removal from the network, involves the following five components:

(1) **Intent Profiling:** The initial component of IBN involves user interaction with the IBN to express her intent. Intent is submitted in a human-friendly manner, such as through natural language expressions or drop-down menus, and the IBN may guide users toward expressing meaningful intents.

(2) **Intent Translation:** Once the intent is submitted to the IBN, it needs translation into a network policy, which is then converted into low-level configurations for network devices.

(3) **Intent Resolution:** Users may independently submit their intents, potentially leading to conflicting network configurations. An IBN should have a policy resolution module to disambiguate conflicts, propose resolutions and/or alert users and administrators if conflicts cannot be resolved.

(4) **Intent Activation:** Once the IBN ensures that activating a new intent won't impact existing intents, it proceeds with provisioning the requested service.

(5) **Intent Assurance:** The intent assurance component ensures the satisfaction of the intent at the time of deployment. It proactively and reactively ensures that the network aligns with the intent throughout its lifetime, assisting users in refining intent when a gap is detected between their desires and network behavior.

In Fig. 2.1, the illustration outlines the interaction process among the five components crucial for achieving a Closed-Loop Automation (CLA) IBN. Closed-loop control is an important aspect of autonomic (e.g., self-managing) networks [12]. The closed loop establishes the foundation for an autonomic and agile learning network, aiming to minimize errors introduced by human intervention. As depicted in the figure, the user is outside of the closed loop and the produced intent is entered into a CLA IBN, which automatically translates the intent, resolves conflicts, configures the network to achieve what is requested in the intent without needing human intervention, and constantly monitors the network to ensure that all requirements of the intent are met throughout its lifetime.

### 2.1.6 Taxonomy of Intent Expression Methods

Intents can be categorized based on how they are expressed [1]. Some common ways to express intent are as follows:

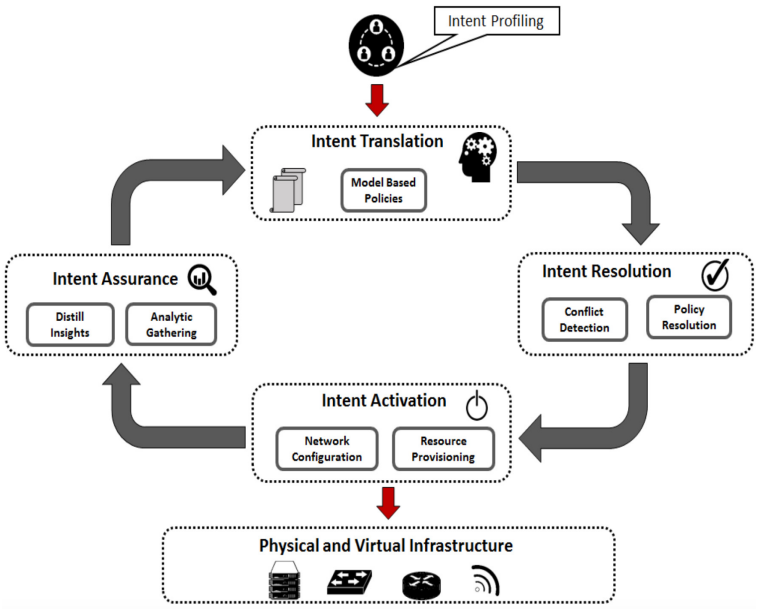



Figure 2.1: Interaction of the main IBN components [1].

### Graphical User Interface (GUI)

The most straightforward way for users to express their intent in IBN is through a Graphical User Interface (GUI), ensuring a safe and guided approach through drop-down and drag-and-drop menus [1]. Users can follow a template-based approach to specify attributes for customizing their network service. For instance in Fig. 2.2, a user might choose a video conferencing service with medium quality, medium security, and set the start and end time.

### Natural Language

The GUI is a semi-flexible tool for users to express intent but may limit options to those provided. IBN can also be communicated in human language, either written or oral, requiring the IBN to incorporate a Natural Language Processing (NLP) tool, a subset of Artificial Intelligence (AI), for converting human language into machine language [1]. NLP could have different use cases based on the type of processing that needs to be done. For instance in Name Entity Recognition (NER), NLP classifies extracted named entities into predetermined categories (e.g., names, organizations, etc.) by using methods like unsupervised (e.g., clustring) and supervised algorithms (e.g., decision trees, support vector machine (SVM)) [15]. In IBN, NLP can be used for a chatbot-like conversational interface where the user expresses her intent in natural language. In this case, IBN usually asks questions to understand the user's requirements, and the user provides answers. Here, the IBN aims to extract network-related labels such as bandwidth, timeframe, and endpoints [16, 17]. Besides the traditional chatbot, users can express their intent by voice. For example, In [18] the Amazon Alexa skill developer service is utilized to personalize the voice assistant functionality.

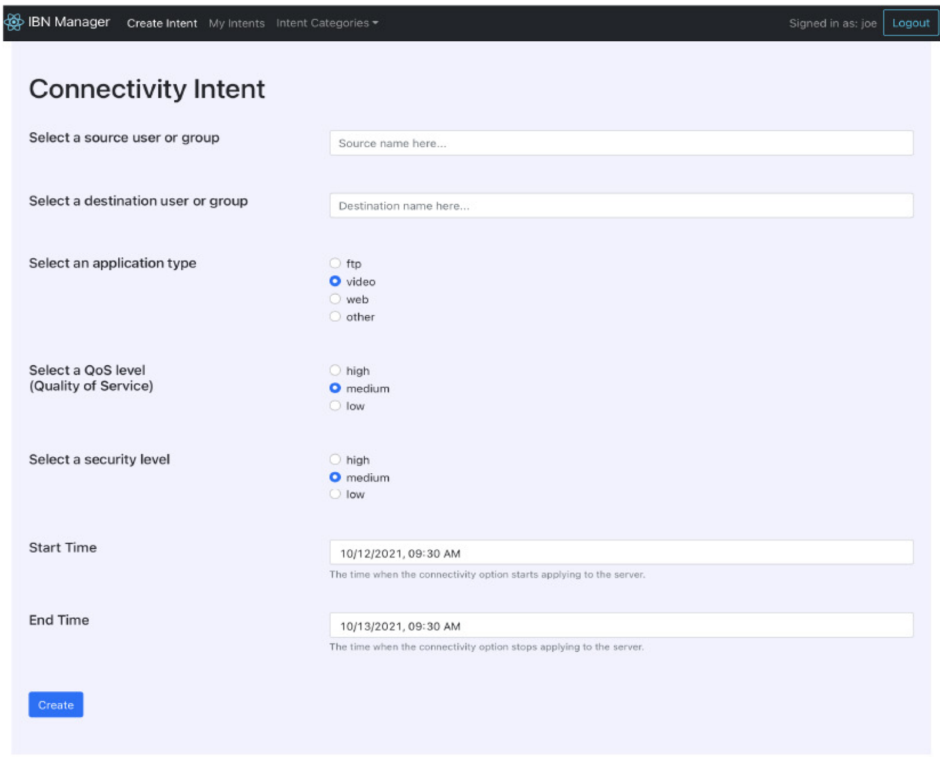



Figure 2.2: A GUI example for intent expression [1].

**Intent-Based Languages**

An alternative to GUI and NLP is the use of IBN-specific languages. These languages serve as programming languages, minimizing the gap between human and machine readability and are suitable for more technical users like network operators/administrators [1]. Nile [19] is a language that identifies intent through a name and a set of tags or keywords, specifying the intent scope, including middleboxes, Quality of Service (QoS) metrics, traffic manipulation (e.g., allow/block), endpoints, and the intent's lifecycle. Fig. 2.3 provides an example inspired by [19] to express an intent like "*I want to have a Video Conference between Concordia University and Ericsson with bandwidth more or equal to 500 Mbps and latency less than 20 ms, through a video encoder and video decoder, from 3 pm to 4 pm.*".

### 2.1.7 Taxonomy of Intent Translation

In [1], various methods for translating intent are introduced, and we will discuss some of these methods here. Below are some of the popular translation methods:

**Template/Blueprint Translation**

The easiest way to translate an intent is through having a template or blueprint, where the user expresses their intent. User intent expression using the GUI/Template method is a good match for this type of translation, as user

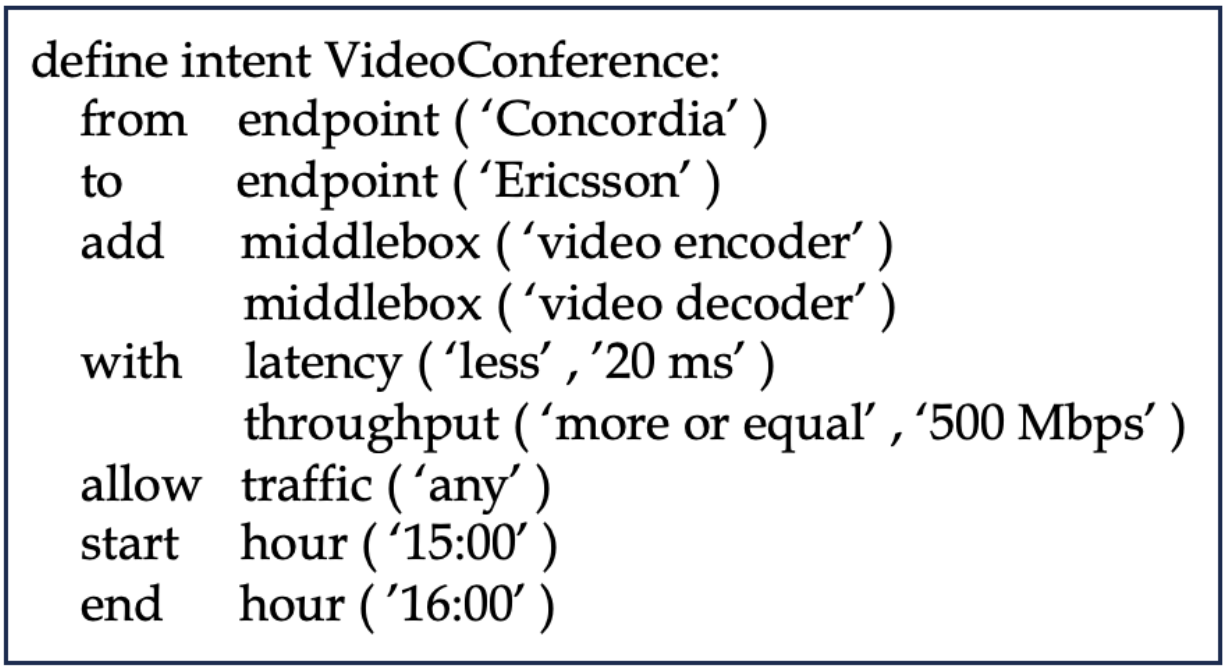



Figure 2.3: An example of intent expression with Nile.

selections can automatically be translated into network policies and configurations.

**Keyword**

A keyword-based mechanism is particularly convenient for NLP intent translation. For instance, specific connectivity services can be chosen based on keywords used in the intent (e.g., wireless connection). The natural language-based intent can be tokenized into keywords, parsed using a rule-based matching function, and associated with specific tags.

**Semantics**

When translating an intent, structuring semantic relationships is facilitated using Resource Description Framework (RDF) graphs [1]. RDF graphs construct intent semantics, with nodes representing entities and edges depicting relations between entities [16]. Fig. 2.4 shows an example of an RDF. In the root of the graph, there is the user's main request (e.g., video conference) which is followed by its requirements (e.g., endpoints, duration, bandwidth), and each requirement has its argument.

**Machine Learning (ML)**

Keywords in intent can be automatically mapped to predefined classes of rules or template policies using Machine Learning (ML) techniques, eliminating the need for manual intervention [3]. One approach is using ML classification techniques to extract classes that show the relationship between user intentions and service characteristics. These

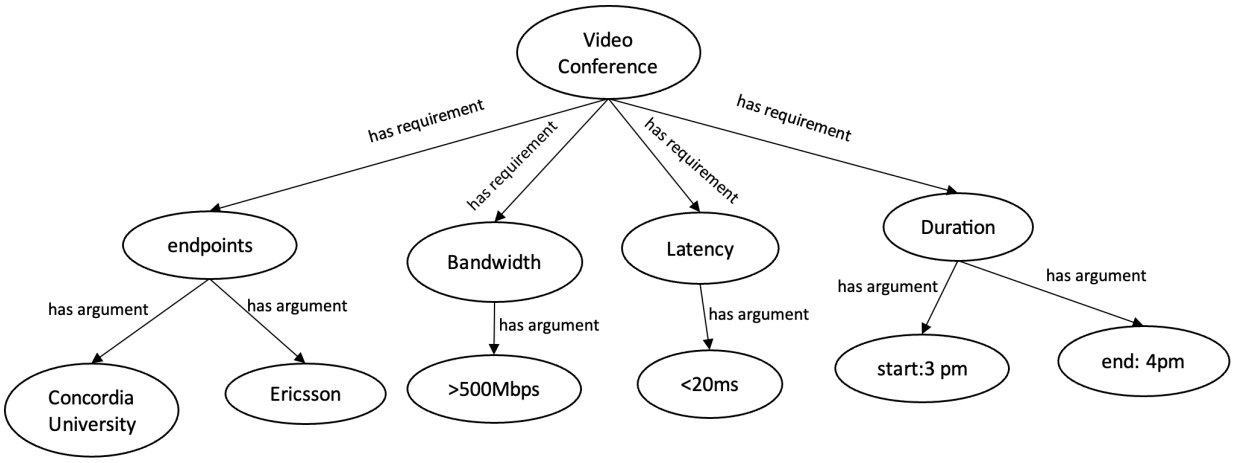



Figure 2.4: An example of a RDF.

classes aid in automating the mapping of keywords to network characteristics in a policy. To ensure accuracy, the models should be trained on a library of past intents, and the translated intent can be stored in a policy database for reinforcement learning to identify new policies when needed [1]. Another approach is applying a sequence-to-sequence learning model, specifically using Recurrent Neural Network (RNN) techniques, predicts expected keywords, and extracts semantic correlations between different elements of the intent [19].

## 2.2. Cloud Computing

In this section, we first define cloud computing. Then, we discuss the essential characteristics of cloud computing, the layers of cloud computing, and various cloud deployment models. Finally, we talk about Amazon EC2, which provides on-demand, scalable computing capacity in the AWS Cloud.

### 2.2.1 Definition

In [5], the authors integrate various definitions of cloud computing and present a unified, comprehensive definition: ”Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs”. Moreover, the National Institute of Standards and Technology (NIST) defines cloud computing as a ”model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction” [20].

### 2.2.2 Essential Characteristics

In this section, we provide a brief overview of the characteristics that distinguish cloud computing as a unique paradigm [20, 21].

- **On-demand self-service** A consumer can automatically provision computing resources, like server time and storage, without human interaction. For example, this can be achieved by making cloud services accessible on the web through a programmable interface, such as an Application Programming Interface (API).
- **Easy access** Capabilities are network-accessible via standard mechanisms, supporting diverse client platforms like phones, tablets, and laptops.
- **Resource pooling** The provider combines resources to serve many customers, adjusting them based on demand. Customers can't control where the resources are located but can choose general preferences like country or datacenter. These resources include storage, processing power, memory, and bandwidth.
- **Scalability** Cloud providers offer vast resources, enabling services to scale up or down as required. This capability is particularly beneficial during sudden increases in demand, ensuring that services remain responsive and available. The ability to handle rapid changes in usage without disruption highlights the flexibility and reliability of cloud-based solutions, making them an ideal choice for dynamic and growing applications.
- **Elasticity** Capabilities can be rapidly scaled up or down, often automatically, to match demand. To consumers, resources seem unlimited and available as needed. For instance, IaaS can offer elasticity based on the provider's capabilities. When a task with a high workload is running, IaaS can dynamically allocate additional resources to ensure the service operates smoothly under the increased demand. Once the workload decreases, these extra resources can be released, helping to reduce costs.
- **Measured service** Cloud systems automatically manage resources using metering to monitor, control, and report usage (e.g., storage, processing, bandwidth), ensuring transparency for providers and consumers.

### 2.2.3 Cloud Computing Layers

Cloud computing services are structured into three distinct layers—Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS)—depending on the level of abstraction in the provided capabilities and the service model employed by the provider [20, 21].

#### Infrastructure as a Service (IaaS)

Infrastructure as a Service (IaaS) represents the foundational layer of cloud computing. It provides consumers with access to essential computing resources such as processing power, storage, and networking, enabling them to deploy and manage their own software, including operating systems and applications. Prominent examples of IaaS include Amazon Elastic Compute Cloud (EC2), Microsoft Azure, Google Compute Engine, and IBM Cloud.

#### Platform as a Service (PaaS)

Platform as a Service (PaaS) serves as the intermediate layer in the cloud computing hierarchy. It provides developers with a ready-to-use environment for building and deploying applications, utilizing programming languages, libraries, tools, and services supported by the provider. This approach eliminates the need for developers to manage or configure the underlying cloud infrastructure, such as servers, networks, operating systems, or storage. Notable examples of PaaS include Microsoft Azure, Google App Engine, and Cloud Foundry.

#### Software as a Service (SaaS)

Software as a Service (SaaS) represents the highest layer in the cloud computing model. It enables consumers to access applications hosted on the provider's cloud through web interfaces or APIs. This model transforms traditional desktop applications, such as word processors, into web-based services, relieving users from software maintenance responsibilities while streamlining development and testing processes for providers. Prominent examples of SaaS include Google Docs, Microsoft Office 365, Salesforce, and Dropbox.

### 2.2.4 Cloud Deployment Models

Cloud deployment models vary based on factors such as ownership, user(s), management responsibilities, security requirements, and the physical location or distribution of the underlying infrastructure. This section provides a brief overview of three primary cloud deployment models [20, 21].

#### Public Cloud

A public cloud is typically owned by a large organization, with its infrastructure hosted on the provider's premises, and is accessible to the public on a pay-per-use basis. Users do not need to make an upfront investment in infrastructure, but they have limited control over aspects such as data, network, and security configurations.

#### Private Cloud

A private cloud is dedicated to a single organization and can be constructed and managed either by an external service provider, the organization itself, or a collaboration of both. Although it demands a significant initial investment, it offers the highest level of control over aspects such as security, performance, reliability, and other critical factors.

#### Hybrid Cloud

A hybrid cloud combines public and private clouds to leverage the advantages of both. For example, an organization could use a private cloud for running critical applications with sensitive data on-premises, while utilizing a connected public cloud to easily scale resources up or down as needed.

### 2.2.5 Amazon Web Services (AWS)

Amazon Web Services (AWS) is the leading and most widely used cloud platform, offering over 200 fully equipped services from data centers worldwide [22]. Millions of customers, including rapidly growing startups, large enterprises, and major government agencies, use AWS to reduce costs, increase agility, and accelerate innovation. AWS offers the most comprehensive and feature-rich cloud services, ranging from core infrastructure technologies like compute, storage, and databases to emerging fields such as machine learning, AI, data lakes, and the Internet of Things (IoT), making it easier, faster, and more cost-effective to move applications to the cloud and build innovative solutions. AWS serves millions of customers worldwide, including startups, enterprises, and public sector organizations.

### 2.2.6 Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2) is a scalable, on-demand computing service within the AWS Cloud that enables users to provision and manage virtual servers [23]. EC2 helps reduce hardware costs and accelerates the development and deployment of applications. It allows for flexible scaling, enabling users to increase capacity to handle compute-intensive tasks or traffic spikes and decrease capacity when demand subsides. Additionally, EC2 provides the ability to configure security, networking, and storage to meet specific application requirements. EC2 instances are virtual servers within the AWS Cloud. The instance type selected during the launch process determines the hardware resources allocated to the instance, with each type offering a distinct configuration of compute, memory, network, and storage resources.

### 2.2.7 Amazon EC2 Regions and Zones

Amazon EC2 operates globally across AWS Regions, Availability Zones, Local Zones, AWS Outposts, and Wavelength Zones [24]. Regions are distinct geographic areas, while Availability Zones are isolated locations within each

Region for redundancy. Local Zones place resources closer to end-users, AWS Outposts extend AWS services to on-premises facilities, and Wavelength Zones deliver ultra-low latency by integrating AWS services with 5G networks. AWS data centers are highly reliable, but failures can occur, so distributing instances across multiple Zones or Regions ensures continued availability.

Each AWS Region is divided into multiple isolated locations called Availability Zones, identified by a Region code and a letter (e.g., us-east-1a) [24]. When launching an instance, users choose a Region and a Virtual Private Cloud (VPC), and instances can be placed in different Availability Zones to improve reliability. Distributing instances across Zones ensures that if one instance fails, another in a different Zone can handle the workload. Elastic IP addresses can also help manage failures by quickly redirecting traffic to a working instance in another Zone. As Zones grow, capacity limits may restrict new instance launches in certain Zones, leading to differences in the number of available Zones across Regions.

## 2.3. Service Graph

In this section, we begin by introducing the concepts of Virtual Network Function (VNF), Service Chain, and VNF Forwarding Graph (VNF-FG). Following this, we define microservices and service graphs.

### 2.3.1 Virtualized Network Function (VNF)

A Network Function (NF) is a functional component within a physical network infrastructure, characterized by well-defined external interfaces and specific functional behavior [25]. Examples include elements like Residential Gateways in home networks or traditional functions such as Dynamic Host Configuration Protocol (DHCP) servers and firewalls. In Network Function Virtualization (NFV), NFs are implemented as software modules that can be virtualized and deployed on a Virtual Network Infrastructure (VNI). A Virtualized Network Function (VNF) may consist of multiple components, which can be distributed across several Virtual Machines (VMs), with each VM hosting a single component. A Network Service (NS) comprises one or more NFs. In NFV, these NFs are virtualized and deployed on virtual resources like VMs hosted on high-capacity servers.

### 2.3.2 Service Chain

In the Network Function Virtualization (NFV) ecosystem, a Network Service (NS) represents a collection of interconnected Virtual Network Functions (VNFs) arranged in a specific sequence, as illustrated in Fig. 2.5 [25]. The creation and deployment of an NS within NFV involve three key aspects: (i) determining the number of VNFs required, (ii) defining the sequence in which these VNFs are chained to ensure the desired service functionality, and (iii)

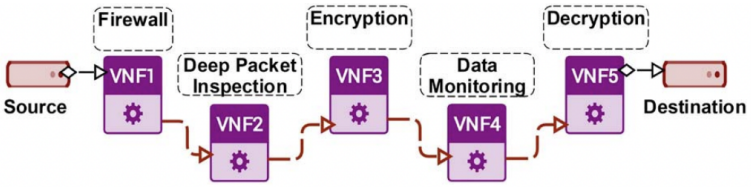



Figure 2.5: Service chain [25].

mapping and allocating the entire chain onto the underlying Network Functions Virtualization Infrastructure (NFVI), also referred to as the Substrate Network (SN). This process ensures that the service is effectively integrated into the virtualized network environment while meeting performance and resource requirements.

### 2.3.3 VNF Forwarding Graph (VNF-FG)

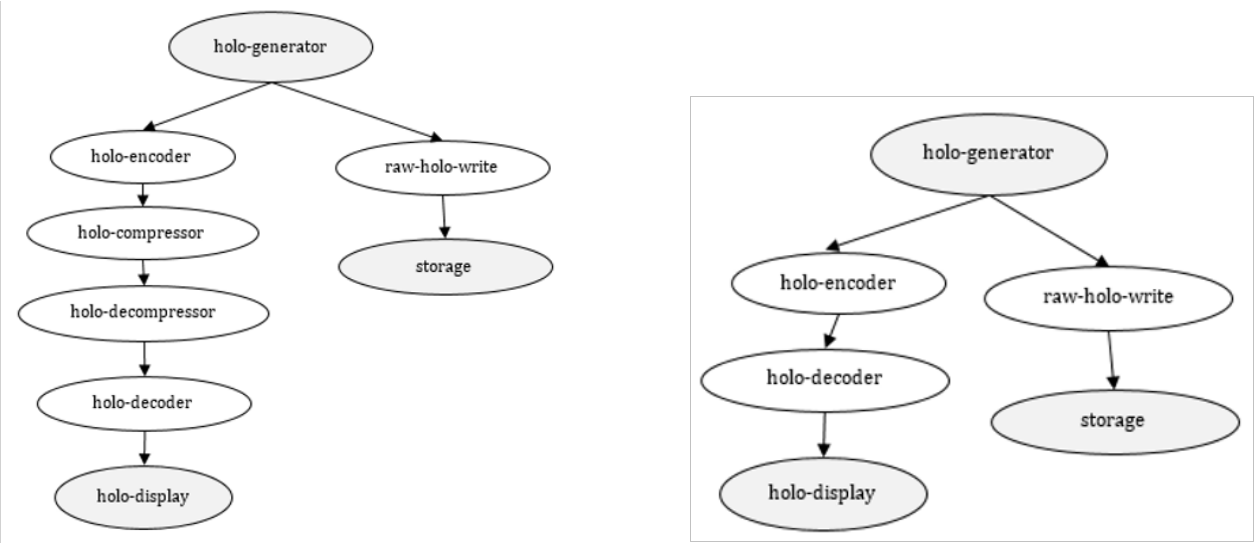
A VNF Forwarding Graph (VNF-FG), as defined by ETSI, represents the logical topology of interconnected VNFs and the traffic flows between them [26]. It specifies how data packets traverse through a sequence of VNFs to achieve the desired network service. Unlike a simple service chain, which enforces a strict order of VNFs, a VNF-FG allows for more complex topologies, including branching, merging, and conditional traffic forwarding. This flexibility makes VNF-FG a powerful abstraction for designing and orchestrating dynamic and scalable network services in an NFV environment.

### 2.3.4 Microservice

A microservice is a small, independently deployable, and loosely coupled service that focuses on performing a specific business function within a larger application [27]. Microservices communicate with each other over lightweight protocols, typically Representational State Transfer (REST) or messaging queues, and are designed to be modular, scalable, and resilient. Each microservice can be developed, deployed, and scaled independently, often using different programming languages or technologies, making them ideal for building complex, distributed systems. This architectural style promotes flexibility, easier maintenance, and faster development cycles, as changes to one service do not directly impact others.

### 2.3.5 Service Graph

In NFV, Virtual Network Functions (VNFs) are traditional Network Functions (NFs) that can be run as virtualized network functions [28]. VNF-Forwarding Graph (VNF-FG) consists of an ordered set of VNFs, which are needed to meet the requirements of the given network service [25]. The concept of specifying network services in terms of VNF-FG can be generalized to any application that is follows the service-based architecture (SBA). In this case, the




(a) (b)

Figure 2.6: Two service graphs for transmitting holographic content to 'holo-display' and 'storage'.

service graph is a directed graph containing a set of nodes and links, where the nodes and links represent services (e.g., microservices) and connection between them, respectively. In this case services are composed according to the user intent, which includes FRs and NFRs, and each service graph consists of an ordered set of services, but unlike VNF-FGs, which contain only VNFs, service graphs contain services including VNFs as well. Figure 2.6 illustrates two distinct service graphs for transmitting holographic content. In Figure 2.6(a), the content is first encoded and then compressed before transmission. At the destination, the data is decompressed and decoded for display, with the raw content also stored in a storage system. In contrast, Figure 2.6(b) depicts a service graph where the content is encoded but not compressed; it is directly decoded for display and simultaneously stored as raw content.

## 2.4. Domain Ontology

In this section, first, we introduce the concept of ontology, which provides a structured framework for representing knowledge. Then, we discuss domain ontology, which focuses on capturing and organizing the unique concepts, relationships, and rules relevant to a particular area of interest.

### 2.4.1 Ontology

An ontology is a structured framework for representing knowledge within a specific domain, detailing concepts, entities, their attributes, and the relationships between them [29]. It serves as a formal specification of a shared conceptualization, enabling consistent communication and understanding among humans and machines. Ontologies are

pivotal in fields like AI, semantic web, and knowledge management, as they facilitate data integration, search, reasoning, and decision-making. By providing a common vocabulary and set of rules, ontologies enable interoperability across systems and support advanced reasoning capabilities. For example, an ontology in the healthcare domain might define entities such as ”patient,” ”disease,” and ”treatment,” and specify how these entities relate to one another.

### 2.4.2 Domain Ontology

A domain ontology is a type of ontology tailored to a specific field or area of expertise [30]. It captures the unique concepts, relationships, and rules relevant to that domain, providing a detailed and formalized representation of domain-specific knowledge. Domain ontologies enable stakeholders to use a common language for communication, support semantic integration of data, and facilitate reasoning within the domain. For instance, in the financial domain, a domain ontology might include concepts like ”account,” ”transaction,” and ”currency,” along with their inter-relationships and rules governing them. Unlike general ontologies, which cover broad and universal concepts, domain ontologies focus on the specificities of a particular field, making them highly valuable for applications such as expert systems, semantic search, and domain-specific AI.

## 2.5. Service Catalog

A service catalog is a comprehensive repository that details all the services an organization offers to its users or customers. It serves as a centralized resource, providing information about each service’s functionality, features, pricing, and usage policies, thereby facilitating efficient service provisioning and management. In the context of cloud computing, a service catalog enables organizations to create and manage catalogs of IT services approved for use. These services can range from virtual machine images and servers to software and databases, encompassing complete multi-tier application architectures [31].

## 2.6. Holographic Communication

Holographic-type communication (HTC) will facilitate the transmission and interaction with holographic data across networks from remote locations [32]. This technology offers practical applications beyond novelty. For instance, holographic telepresence can project remote participants as lifelike holograms into meeting rooms, enabling real-time collaboration with those on-site. Similarly, HTC can revolutionize training and education by allowing users to engage with highly realistic holographic objects for dynamic and immersive learning experiences.

Understanding the details of holographic-type communication (HTC) requires distinguishing between holographic and 3D content [32]. A 3D image is created by combining two static 2D views of a scene, one for each eye, and remains

unchanged regardless of the viewer's position. In contrast, a hologram incorporates parallax, allowing the image to shift dynamically based on the viewer's perspective, enabling interaction with the content. This shift transforms the user's role from a passive observer in 2D and 3D content to an active participant with holograms. Consequently, HTC demands significantly higher capabilities for capturing, transmitting, and enabling interactive experiences.

Streaming a hologram over a network involves capturing, rendering, and transmitting the target object. The process begins with a camera array capturing images of the object from various angles and perspectives. These camera feeds are then combined, processed into a hologram, and encoded for transmission. Once the hologram is streamed across the network, the receiving client decodes the data and renders it for display on a holographic device or projector.

Holographic-type communication (HTC) introduces three critical requirements: massive bandwidth, ultra-low latency, and precise synchronization [32]. Despite advancements in compression, holograms demand significant bandwidth. Innovative techniques aim to minimize transmitted data by eliminating content portions that are irrelevant from the user's perspective, such as areas obstructed or angles outside the user's view. These adaptive streaming schemes dynamically adjust content quality and focus based on the user's position and movement. However, their success relies on accurately predicting user behavior and rapidly adapting data, addressing the "user interactivity challenge." This challenge necessitates not only high bandwidth but also ultra-low latency to maintain seamless interaction with the content. Additionally, HTC requires perfect synchronization of concurrent data streams, even for prerecorded content, as user interactivity—like adjusting viewing angles—demands real-time responsiveness.

## 2.7. Conclusion

In this chapter, we discussed the background concepts relevant to this thesis. First, we introduced IBN, starting with a discussion on autonomic networks and the concept of intent, followed by distinguishing between intent and policy. We then elaborated on IBN, its components, and provided taxonomies for intent expression and translation methods. Next, we explored cloud computing, beginning with its definition and essential characteristics, followed by an overview of its layers, deployment models, and a discussion on AWS, focusing on Amazon EC2. Subsequently, we delved into the concept of service graphs, introducing VNFs, service chains, and VNF-FGs, followed by defining microservices and service graphs. We then discussed domain ontology, starting with the general concept of ontology and progressing to domain-specific ontology. Finally, we discussed service catalogs and holographic communication concepts.

# Chapter 3

# Motivating Scenario, Requirements, and State of the Art

This chapter begins with a motivating scenario to explain the concept of intent-based service graph generation and selection in a cloud environment. We then use this scenario to derive the requirements for an appropriate solution. Finally, we summarize the state of the art service graph selection approaches and assess their applicability against these requirements.

## 3.1. Motivating Scenario

Here, we present a motivating scenario using holographic communication as our use case. Figure 3.1 illustrates this scenario for service graph generation and selection in the context of IBN. Consider a cloud consumer with minimal knowledge of network/cloud configurations but who can describe a request and send a high-level intent in natural language to the cloud. For example, the request could specify the desired use case (i.e., holographic communication), source, destination, and Non-Functional Requirements (NFRs). A typical example of such an intent might be: "I would like to have holographic communication between Concordia University and Ericsson with high interaction and medium quality, and I would also like to store it.". Since the intent is expressed in natural language, it must be translated into a low-level intent using semantic analysis, which can be achieved through a domain ontology. Low-level intent represents the user's request in machine-readable language, such as the Resource Description Framework (RDF) [33], and includes FRs and NFRs extracted from the high-level intent. In this scenario, the FRs are "Holographic Generator," "Holographic Display," and "Storage," while the NFRs include latency between 2–4 ms and bandwidth between 2–4 Gb/s. The service graph generation and selection solution should automatically generate all possible service graphs

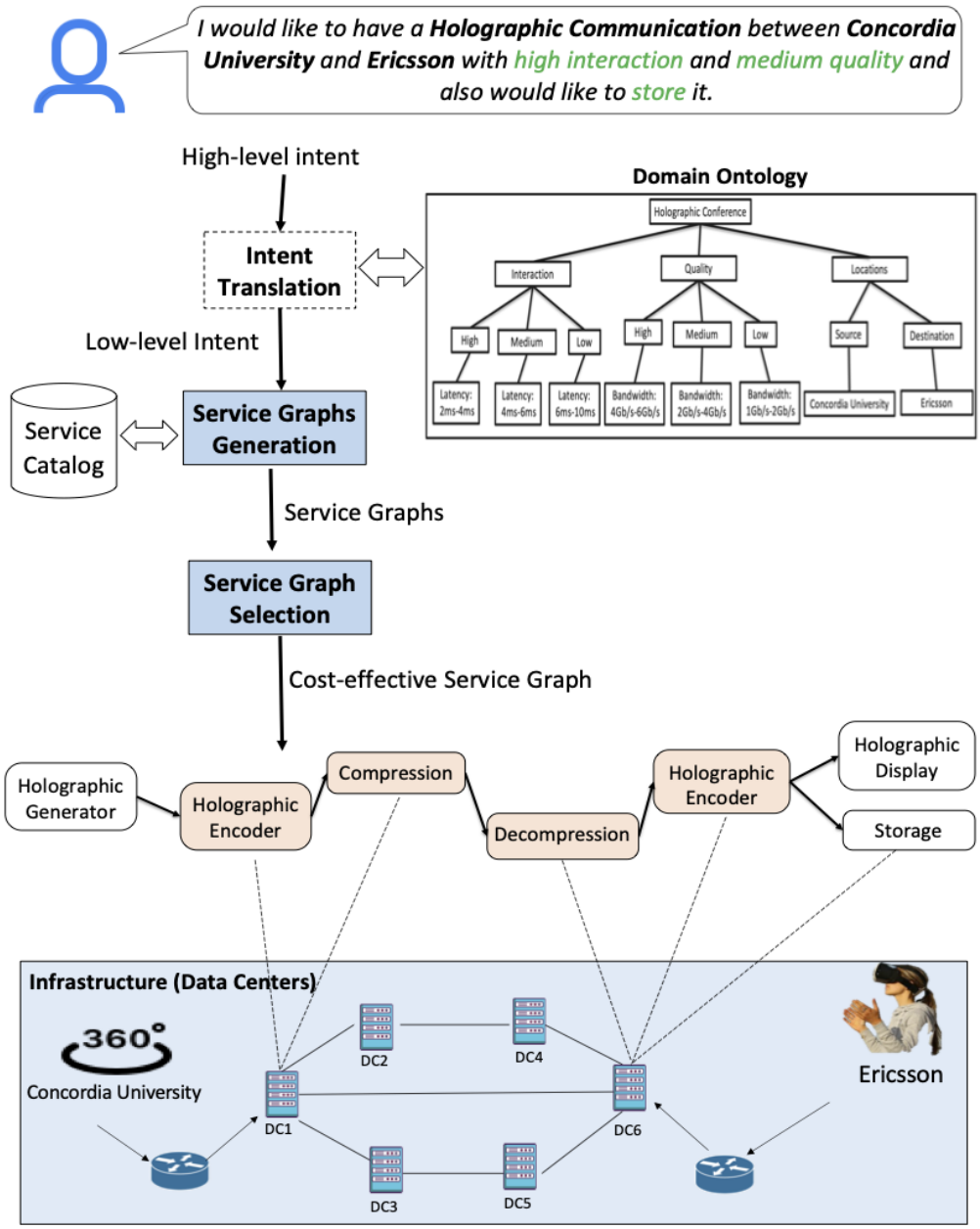



Figure 3.1: Motivating scenario and high-level solution for intent-based service graph generation and selection for cost-effective service deployment in the cloud.

and select the optimal service graph among all possible options, ensuring it meets the FRs and NFRs specified in the low-level intent while minimizing services deployment costs. The service catalog plays a key role in this process, as it contains information about available services, their relationships, and deployment requirements, such as CPU, GPU, and RAM capacities. Additionally, infrastructure details are critical and may include a list of data centers, each with information about available resources (e.g., CPU, GPU, operating systems), as well as details about latency, bandwidth, and the cost of using data center resources and links. The output service graph connects the required services, ensuring the intent is fulfilled from the sources to the destinations. This automated and cost-effective selection process is essential for enabling user-friendly, intent-driven service deployment in the cloud.

## 3.2. Requirements

From the above motivating scenario, it is clear that we need a solution to automatically generate and select a service graph with the minimum deployment cost while meeting the following requirements:

- **Requirement 1:** The first requirement is that cloud consumers with minimum network/cloud knowledge should be able to benefit from IBN by sending their intents in natural language. These users should be able to interact with the cloud by simply expressing their intents in natural language, without needing to understand technical details. This ensures that the cloud is accessible to a broader audience, including cloud consumers who may lack in-depth technical knowledge but are familiar with expressing their requirements and objective in the form of an intent.

- **Requirement 2:** Given that intents are expressed in natural language, the second requirement is the semantic analysis, which helps translate the high-level intent to low-level intent. Semantic analysis plays a critical role in interpreting the intent by extracting meaningful information, such as FRs and NFRs, from unstructured input. This process involves leveraging advanced NLP techniques alongside domain ontology to ensure accurate translation. For instance, in the motivating scenario, it must recognize that a request for “holographic communication between Concordia University and Ericsson with high interaction and medium quality” implies specific services like ”Holographic Generator,” ”Holographic Display,” and ”Storage,” along with latency between 2–4 ms and bandwidth between 2–4 Gb/s.

- **Requirement 3:** The third requirement is that the process of service graphs generation and selection should be fully automated without any human intervention. Once the high-level intent has been translated into low-level intent through semantic analysis, the cloud should autonomously generate all possible service graphs and select a service graph with minimum deployment cost that connect the necessary services to fulfill the specified requirements.

- **Requirement 4:** Given that the cloud consumer in our scenario should not specify the required services, requirement 4 is that service graph generation and selection does not require the cloud consumer to have a profound knowledge about the required services. In the context of the motivating scenario, the cloud consumer only needs to express the high-level intent, such as requesting holographic communication, without explicitly identifying the individual services needed to achieve it, such as ”Holographic Generator,” ”Holographic Display,” or ”Storage”. The solution should be capable of autonomously identifying these services based on the intent, leveraging domain ontology and the service catalog to determine the appropriate services and their relationships.

- **Requirement 5:** Considering the deployment constraint while selecting a service graph is another requirement. This means that the service graph selection process must account for various resource and infrastructure limitations to ensure the feasibility and efficiency of the deployment. Deployment constraints can include factors such as the availability of computational resources (e.g., CPU, GPU, RAM) in datacenters, the operating systems supported, and the geographic distribution of resources. Additionally, network-related constraints, such as the latency and bandwidth between datacenters, must also be considered to ensure the selected service graph meets the specified NFRs like low latency or high bandwidth. For example, in the context of holographic communication, the solution must evaluate whether the selected datacenters have sufficient computational capacity to support services such as holographic encoder and decoder. At the same time, it must ensure that the network links between these datacenters can meet the required latency and bandwidth thresholds to provide a seamless user experience. It should also factor in cost-related constraints, such as the expense of using specific resources or network links, to optimize the overall service graph deployment cost in the cloud.

## 3.3. State of the Art

In this section, we review the state-of-the-art research from two key perspectives: traditional service graph selection and intent-based service graph selection.

### 3.3.1 Traditional Service Graph Selection

Kim et al. in [34] proposed a latency-based Graph Selection Manager (GSM) to select one or more VNF forwarding graphs. Before delving into the details of this paper, it’s essential to understand the NFV architecture, which consists of three main components: Virtual Network Functions (VNFs), NFV Infrastructure (NFVI), and NFV Management and Orchestration (NFV MANO). VNFs are software-based instances of network functions, that replace traditional hardware. NFVI provides the hardware and virtualization resources to support VNF execution, including servers, storage, and networking equipment. NFV MANO is responsible for automating and orchestrating the NFV environment which

encompasses NFV Orchestrator (NFVO), VNF Managers (VNFMs), and Virtualized Infrastructure Manager (VIM). NFVO is responsible for coordinating and managing the lifecycle of VNFs and the underlying NFVI resources. VNFM manages the lifecycle of individual VNFs, including instantiation, scaling, and termination. VIM manages the NFVI resources and allocates them to VNF instances.

The proposed GSM in [34] addresses the challenge of selecting VNF forwarding graphs in NFV-MANO systems while ensuring network QoS. It integrates with the NFV Orchestrator (NFVO) and uses a common southbound interface to communicate with SDN hypervisors and Virtualized Infrastructure Managers (VIMs). The GSM focuses on selecting VNF forwarding graphs that meet Service Level Agreement (SLA) requirements, specifically targeting parameters like latency and network capacity. An SLA is a formal definition of the relationship between a service provider and its customer, detailing the specific services to be provided and the expected level of service [35]. The GSM first gathers latency information between VIM pairs by querying the VIMs for inter-latency, which is then used to calculate the total latency of potential VNF forwarding graphs. This total latency, combined with other network conditions, is used by the NFVO to select the most suitable VNF forwarding graph that satisfies the requested SLA. The system ensures that the selected service chain maintains low latency and sufficient network capacity, even under limited resources. However, they did not consider the information of data centers (e.g., CPU, GPU, service deployment cost.) when selecting these graphs.

In [36], a graph-based Particle Swarm Optimization (PSO) technique is presented, which optimizes web service compositions for QoS properties. Traditional methods, such as those based on greedy algorithms, often involve selecting a preconfigured workflow to optimize, which can lead to suboptimal solutions due to the inherent limitations of the preselection process. In contrast, the proposed graph-based PSO technique eliminates the need for a preselected workflow by utilizing a Directed Acyclic Graph (DAG) representation. This graph captures all possible interactions between candidate web services based on their input and output requirements. The approach begins with an algorithm that identifies all relevant services that meet the composition task's input and output criteria. Once relevant services are identified, a master graph is constructed, with each node representing a service and edges denoting the possible input-output relationships. The PSO algorithm operates on this graph, where each particle in the swarm represents a potential composition by selecting edges from the master graph. This simultaneous optimization of both the workflow structure and service selection allows for greater flexibility and the potential to discover optimal or near-optimal solutions. The fitness function developed in this approach integrates four key QoS properties: availability, reliability, response time, and execution cost. The function is designed to ensure that all properties are normalized within a range [0,1], facilitating a balanced evaluation of the composition of the services. The results demonstrate that this method improves adaptability and optimization quality compared to traditional methods, although it incurs a higher computational overhead. However, this approach does not account for scenarios that involve multiple sources and destinations.

Sailer et al. [37] introduced a graph-based solution for semi-automated service creation in IT clouds considering

data centers deployment costs. This paper addresses the challenges of manual service development, which is often inefficient and error-prone, particularly in complex IT environments. The proposed solution integrates Business Support Services (BSS) with Operations Support Services (OSS) through a comprehensive graph representation, effectively mapping the various service components and their inter-dependencies. Note that, BSS refers to the set of services that facilitate essential business operations, including customer management, order management, revenue management, and product management. Conversely, OSS is defined as the service layer responsible for processing requests from BSS to manage and allocate the underlying IT resources within a data center. This includes functionalities like virtual machine management, network and storage allocations, and image management. At the core of their solution is the Services Catalog, a hierarchical graph structure that illustrates existing services, their features, requirements, and operational rules. This catalog serves as a central repository from which service developers can explore and select modular building blocks to create new services. The graph is designed such that the leaves represent fundamental IT operations, while higher-level nodes encapsulate more complex service offerings. This modularity allows developers to recombine existing services to define new offerings with minimal effort. The creation process begins with the identification of service requirements, which are expressed as rules associated with various attributes such as CPU, memory, storage, and quality of service metrics. These requirements are aggregated based on conditions defined for each service type, ensuring that the resulting configurations meet both operational and customer needs. Once a new service is defined, the deployment phase involves selecting appropriate data centers equipped with the necessary resources. The algorithm evaluates potential resource allocations, taking into account both availability and cost considerations. It performs a matching process to ensure that the requirements of the service align with the capabilities of the data centers, thus optimizing resource utilization and minimizing costs. The paper illustrates the effectiveness of this approach through a case study centered around a Desktop service, highlighting how different user profiles (e.g., administrative personnel, developers, graphical designers) can be accommodated by leveraging existing OSS and BSS functionalities. However, it does not account for bandwidth costs between data centers.

Authors in [38] formulated VNF Selection and Chaining Problem (VNF-SCP) as a Binary Integer Programming (BIP) model to compute the routing path for each Service Function Chain Request (SFCR) aiming for the minimum end-to-end delay. They propose a novel solution called the Deep Learning-based Two-Phase Algorithm (DL-TPA). The DL-TPA consists of two main components: a VNF selection network and a VNF chaining network, both designed using Deep Belief Networks (DBNs). In the first phase, the VNF selection network intelligently selects the optimal instances of virtual network functions based on real-time network conditions and user requirements. This involves analyzing available resources, such as CPU and memory capacities, along with the specific demands of each service function request. The second phase, the VNF chaining network, takes the selected VNFs and computes the optimal routing paths to link them according to predefined orders. This two-phase approach allows the DL-TPA to process multiple service function requests simultaneously, significantly enhancing time efficiency compared to traditional methods that

rely on massive iterations for path computation. The results of the evaluation demonstrate that DL-TPA achieves prediction accuracy, with about 82.8% for VNF selection and 98.7% for VNF chaining.

### 3.3.2 Intent-based Service Graph Selection

Authors in [39] proposed an automated approach for designing Network Services (NSs) that begins with NS Requirements (NSReq), which include functional, and non-functional characteristics. The approach uses the Network Function Ontology (NFO), derived from past knowledge about Network Functions (NFs), which aids in translating NSReq. They used NFO to capture the knowledge about the NS design. Then, VNFs that match the functionality of NSReq will be selected from the VNF catalog. VNF catalog contain information about the characteristic of VNFs. Each VNF in the VNF catalog has a Virtual Network Function Descriptor (VNFD) which describes the attribute of a VNF from the perspective of management and orchestration. Since there is no information about the functionality of the VNF, they extend VNFD and added new elements to describe the functionality of the VNF. In this paper, the user should have knowledge about the functional or non-functional requirements of his/her request. There is no implementation for the solution. Ref. [39] meets "R2" and "R3". In [28] an approach to select VNFs for Network Slices (NwSs) based on high-level intents is proposed. It uses an ontology as a knowledge base for NwS design to decompose high-level intents. Their approach considers the placement of some functionalities in certain locations and/or isolation constraints for different parts of NwS. The decomposed intent allows for the selection of suitable VNFs/PNFs and the determination of their constraints in terms of location and anti-affinity. This work meets "R2". We note that both [28, 39] rely on the fact that the FRs and NFRs are provided by the end-users, who are then required to have a notable amount of knowledge about the network/service.

iNDIRA which is a tool for intent-based networking using SDN north-bound interfaces that allow automated and dynamic setup of paths between network providers is introduced in [33]. iNDIRA is designed for science applications and utilizes natural language processing to construct semantic graphs, addressing challenges like descriptive language development, keyword identification, user negotiation, and automated network provisioning. Moreover, it can identify conflicts, align potential network services with intents, provision dynamic network resources across diverse network architectures, and communicate with users and applications as necessary. This work meets "R1", "R2", and "R3".

In [40], an Intent-Based Cloud Service Management (ICSM) is introduced to process cloud operator intents expressed in natural language. The input of ICSM is a service requester (e.g., intent). The requirement analyzer extracts details such as service type (e.g., web service) and security level (high, medium, low) from intent. The resource composer automates resource composition, determining VNF types based on user intent. It comprises a resource composition template repository, customization engine, and customization rule repository. Resource composition templates store a small number of resource composition templates for each basic category of services which shows the required

VNFs for each service. The customization engine checks if the user's intent aligns with service types in the repository and satisfies associated rules. The customization rule repository stores customized policies for resource composition templates. This work meets "R1", "R2", and "R3".

Authors in [41] proposed INSpIRE, an IBN solution for refining intents into configurations to achieve desired service chains. INSpIRE involves three steps: modeling the intent domain based on functional and non-functional requirements (FRs and NFRs), quantitatively calculating NFRs for VNFs through a methodology based on Softgoal Interdependency Graphs (SIGs), and parsing intents for VNF clustering. SIGs represent relationships among various softgoals, which are qualitative objectives or criteria that cannot be precisely quantified. For example, consider one non-functional requirement: Security. Using the SIG method, we break down "Security" (softgoal) into four specific capabilities (leaf-softgoals): Information Gathering, Logging, Detection, and Prevention, common in Intrusion Detection and Prevention Systems (IDPS). Each capability is assigned a weight indicating its importance in achieving the initial security goal. if a network operator specifies that a VNF does not store logs on a server (Logging [Middlebox]) operationalization is going to be zero. Operationalization scores are calculated from top to bottom of SIG, and then based on these scores the VNFs will be clustering as high, medium, and low levels. INSpIRE automatically assesses and assigns scores for VNFs' non-functional requirements, utilizing these scores to cluster and select suitable VNFs for a given intent. The solution selects VNFs from the cluster according to user intent (e.g., high security). Although, the authors proposed a SIG model for security, however, there are other NFRs such as bandwidth, latency, and reliability that are not supported. Additionally, the process of VNF selection is not optimized and VNFs are selected randomly from each cluster.

Although these papers [33, 40, 41] satisfy "R1" to "R3", since required services are mentioned in intent, they did not meet "R4". All these works capture the required VNFs from user intent (specified in the intent) or use-case requirements. It means there is knowledge about the required service(s) to generate service graphs, and selecting services is based on their NFRs. As a consequence, none of these papers satisfies "R4".

Leivadeas and Falkner in [42] introduced a hybrid VNF placement algorithm designed to capture users intent and deploy it via a customized service function chain deployment solution, ensuring various levels of Quality of Service (QoS). This paper presents a framework that combines IBN and NFV to automate the deployment and configuration of VNFs in cloud infrastructures, minimizing manual intervention. The proposed approach allows users to specify intents, such as desired QoS, security requirements, and activation timeframes, through a template-based approach. These intents are translated by the NFV Orchestrator into Service Function Chains (SFCs) using predefined blueprints stored in a database. Multi-tenancy is supported by grouping identical intents, optimizing resource usage, and reducing costs, while single-tenant configurations are employed for high-security requirements. The NFV Manager oversees resource allocation, ensuring the intent attributes are satisfied while balancing load and adhering to infrastructure constraints. An assurance module monitors the deployed SFCs to verify compliance with user intents, enabling dynamic adjustments.

Di Riccio et al. in [43] presented MultiDips, a declarative tool, along with a Mixed-Integer Linear Programming (MILP) solution to meet multiple intents on Cloud-Edge IBN infrastructures through the placement of VNF chains. This paper addresses the challenge of deploying VNF chains in IBN for Cloud-Edge infrastructures. The problem involves fulfilling user intents, like low-latency video streaming or secure data storage, while balancing environmental sustainability, energy efficiency, and infrastructure costs. To solve this, the authors propose a heuristic approach implemented in an open-source tool called MultiDips. This method simplifies decision-making compared to complex mathematical models, allowing faster placement of VNFs while considering energy consumption, carbon emissions, and infrastructure profits. Their solution supports multiple intents by defining VNF chains based on user requirements and optimally placing these chains across Cloud-Edge nodes. For example, one scenario involves deploying a video streaming service with both edge and cloud components to meet user demands for low bandwidth and high-quality streaming. Another scenario involves securely storing data in the cloud with strict latency requirements. The results show that MultiDips effectively balances trade-offs between energy use, carbon emissions, and operational costs.

In [44], an automated network assurance model is introduced, leveraging model predictive control to ensure the QoS and security requirements of multi-tenant and IBN-enabled service function chains. Moreover,it addresses the challenge of network assurance in modern infrastructures, particularly focusing on Service Function Chains (SFCs) within the context of NFV, Internet of Things (IoT), and 5G applications. As networks grow and new applications with strict QoS requirements emerge, managing performance becomes increasingly complex. The proposed solution is an automated model based on Model Predictive Control (MPC) that proactively adjusts user flow allocations to maintain QoS and security. The model handles dynamic network conditions, ensuring minimal QoS violations through real-time decision-making. It uses a multi-tenant approach to optimize resource usage and minimize relocations. Two optimization problems are formulated: one for user relocation and another for admission control, triggered when relocations alone fail to meet SLA requirements. The MPC is applied iteratively to solve these problems, adjusting user flow allocation over a prediction horizon. The relocation optimization minimizes disruptions while prioritizing relocations based on QoS and security. If relocations are insufficient, admission control is employed, rejecting users to maintain SLA satisfaction. This dual MPC approach ensures system stability and SLA compliance. The end-to-end delay modeled as a time-varying state and formulates the problem as a finite optimization problem, ensuring that user intents, such as high QoS or security, are consistently met. The model's effectiveness is demonstrated through a simulation, showing that it can guarantee high application performance while minimizing disruptions.

The evaluation of related works, summarized in Table 3.1, reveals that most existing solutions address only a subset of the requirements discussed in section 3.2. For example, Kim et al. [34] and da Silva et al. [36] focus on automation (R3) and select services without prior knowledge (R4) but lack natural language support (R1) and semantic analysis (R2). Sailer et al. [37] and Pei et al. [38] integrate some aspects of deployment constraint handling (R5) but fail to provide intent-based interaction (R1, R2). Notably, solutions such as Kiran et al. [33], Chao et al. [40], and Scheid et

Table 3.1: Related Work Evaluation.

| References | R1 | R2 | R3 | R4 | R5 |
|---|---|---|---|---|---|
| Kim et al. [34] | ✗ | ✗ | ✓ | ✓ | ✗ |
| da Silva et al. [36] | ✗ | ✗ | ✓ | ✓ | ✗ |
| Sailer et al. [37] | ✗ | ✗ | ✓ | ✗ | ✓ |
| Pei et al. [38] | ✗ | ✗ | ✓ | ✓ | ✓ |
| Nazarzadeoghaz et al. [39] | ✗ | ✓ | ✓ | ✗ | ✗ |
| Gritli et al. [28] | ✗ | ✓ | ✓ | ✗ | ✗ |
| Kiran et al. [33] | ✓ | ✓ | ✓ | ✗ | ✗ |
| Chao et al. [40] | ✓ | ✓ | ✓ | ✗ | ✗ |
| Scheid et al. [41] | ✓ | ✓ | ✓ | ✗ | ✗ |
| Leivadeas and Falkner in [42] | ✗ | ✗ | ✓ | ✗ | ✓ |
| Di Riccio et al. [43] | ✗ | ✗ | ✓ | ✗ | ✓ |
| Avgeris et al. [44] | ✗ | ✗ | ✓ | ✗ | ✓ |

al. [41] perform well in using natural language intent (R1, R2) but lack automation (R3) and fail to consider deployment constraints (R5).

Additionally, Gritli et al. [28] and Nazarzadeoghaz et al. [39] satisfy semantic analysis (R2) and automation (R3) but do not consider deployment constraints (R5). Similarly, Leivadeas et al. [42], Di Riccio et al. [43], and Avgeris et al. [44] emphasize deployment constraints (R5) but do not natural language intent interaction (R1). This evaluation highlights the fragmented nature of current solutions, where no single approach comprehensively meets all five requirements, underscoring the need for a solution that bridges these gaps for advanced use cases like holographic communication.

## 3.4. Conclusion

In this chapter, we presented a detailed analysis of intent-based service graph generation and selection in the cloud, focusing on its application to the holographic communication use case. This use case served as a foundation to identify a comprehensive set of requirements essential for designing an effective solution. These requirements included enabling non-technical cloud consumers to interact with the cloud through natural language (R1), performing semantic analysis to translate high-level intents into actionable low-level intents (R2), achieving full automation in service graph generation and selection (R3), eliminating the need for cloud consumers to specify required service details (R4), and considering deployment constraints such as resource availability, network latency, and cost optimization (R5).

We then evaluated existing state-of-the-art approaches against these requirements. While many solutions addressed individual aspects, such as semantic analysis or deployment constraints, none of them provided an approach that fulfilled all the requirements simultaneously. For instance, some works excelled in automation but lacked natural

language processing capabilities, while others prioritized cost optimization without ensuring accessibility for non-technical users. Additionally, certain solutions demonstrated strengths in semantic analysis but failed to integrate deployment feasibility and resource constraints into their solutions. This fragmented landscape of existing approaches underscores a critical gap in the current state of research. It highlights the absence of a unified solution that can comprehensively address the requirements of intent-based service graph generation and selection for cost-effective service deployment in the cloud .

# Chapter 4

# Proposed Solution

This chapter presents the proposed intent-based service graph selection for cost-effective cloud deployment solution while meeting the requirements defined in Section 3.2 We begin with a detailed description of high-level intent translation to low-level intent. Subsequently, we describe service graphs generation and service graph selection using FRs and NFRs derived from high-level intent. Finally, we conclude with a review of the proposed solution in light of the requirements defined in Section 3.2.

## 4.1. High-level Intent Translation

Figure 4.1 shows a high-level intent provided by a cloud consumer. It requests to have a Holographic Extended Reality SPONZA Game. The expected application location is Ericsson Canada. The NFRs are low interaction and medium quality. Finally, the objective is to have a minimum cost. This high-level intent should be translated to low-level intent. The intent translation part which is responsible for preparing the input for our solution, can be done using the requirements parser method proposed in [40], which translates the high-level intent into requested application, source(s), destination(s), and NFRs using tokenizing, Part-of-Speech Tagging (POS), and rule-based matching. We use this information in the next step to achieve a low-level intent.

- **Tokenizing:** This step involves breaking down the input from the high-level intent into individual words or tokens. For instance, the intent in Fig. 4.1 would be split into tokens such as ”Holographic” ”Extended” ”Reality” ”SPONZA” and ”interaction”. Tokenization simplifies subsequent processing by isolating the elements of the input.

- **Part-of-Speech (POS) Tagging:** The tokenized words are then tagged with grammatical or contextual labels to identify their roles in the sentence. Tags such as *ADJ* (adjective), *NUM* (numerical value), and *WORKLOAD*

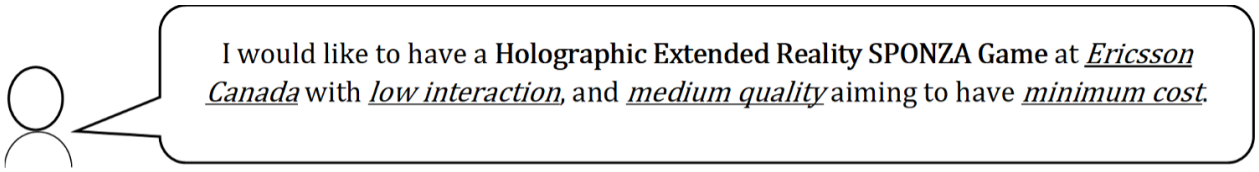



Figure 4.1: High-level intent requesting for Holographic Extended Reality SPONZA Game provided by cloud consumer.

(specific workload or service-related term) are applied to the tokens. For example, in the input ”low interaction”, ”low” might be tagged as *ADJ* and ”interaction” as *WORKLOAD*.

- **Rule-Based Matching:** In this step, the tagged tokens are processed using a set of predefined parsing rules. The matching function uses these rules to classify and organize the tagged words into specific service feature categories (e.g., ”low interaction” as a quality requirement) or application type (e.g., ”Holographic Extended Reality SPONZA Game”).

By determining the application type, location, and NFRs, we can utilize domain ontology to derive a low-level intent that is machine-readable. Figure 4.2 illustrates a domain ontology for the requested Holographic Extended Reality (HXR) SPONZA Game application. The cloud consumer request for low interaction is translated into a latency requirement of less than 120 ms. Similarly, the request for high quality is translated into a bandwidth requirement of greater than 50 MB/s. Additionally, the HXR SPONZA Game is hosted at Ericsson Canada, which translates to the data center locations for the source and destination services. Finally, based on the HXR SPONZA Game application, the initial services required to generate service graphs can be identify which are Oculus Sensors on the source side and Oculus Player and Oculus Audio on the destination side.

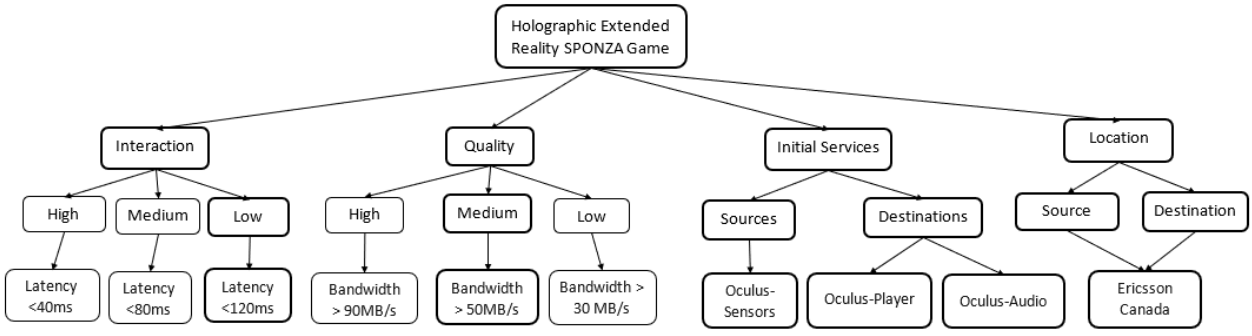



Figure 4.2: Domain Ontology.

Resource Description Framework (RDF) graph is used to construct intent semantics in which the node corresponds to the entity, and the edge represents all kinds of relations between entities [33]. Figure 4.3 depicts an example of translated high-level user intent and extracted features in the scenario shown in Figure 4.1. The first feature, which is

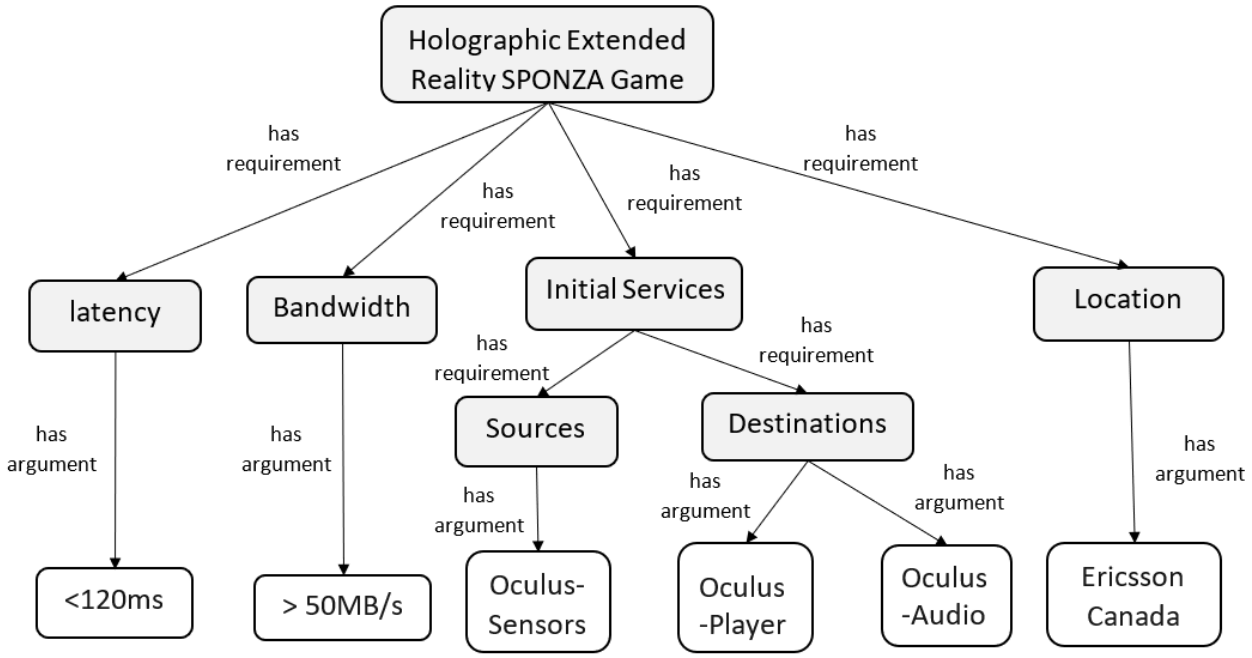



Figure 4.3: High-level user intent translated into an RDF format.

the root of RDF graph, is the requested application, which may have some requirements.

## 4.2. Service Graphs Generation

The overview of our proposed solution for service graphs generation is depicted in Figure 4.4. Our proposed solution comprises two main parts: (1) Intent Translation and (2) Service Graphs Generation.

We discussed the Intent Translation part in the previous section. The initial services (e.g., FRs) for the source and destination sides are determined by referencing the requested application in Fig. 4.3. According to this, the HXR SPONZA game requires ”Oculus-Sensors” as the source and ”Oculus-Player” and ”Oculus-Audio” as the destinations to generate and display the HXR content. The service graphs generation process uses this low-level intent as its input.

As shown in Figure 4.4, the service graphs generation part comprises two modules, namely, Compatible Graph Generating Module (CGGM) and Service Graphs Generating Module (SGGM). The CGGM is responsible for finding the compatibilities between services using the information about initial services captured from the translated intent and service compatibilities specified in the service catalog. The SGGM extracts the candidate service graphs from the compatible graph generated by CGGM. This can be achieved by finding all possible routes between source and destination services in the compatible graph. To help better understanding the different steps of the proposed solution for the service graphs generation part, we explain each step in more detail using our mentioned high-level intent of HXR SPONZA Game.

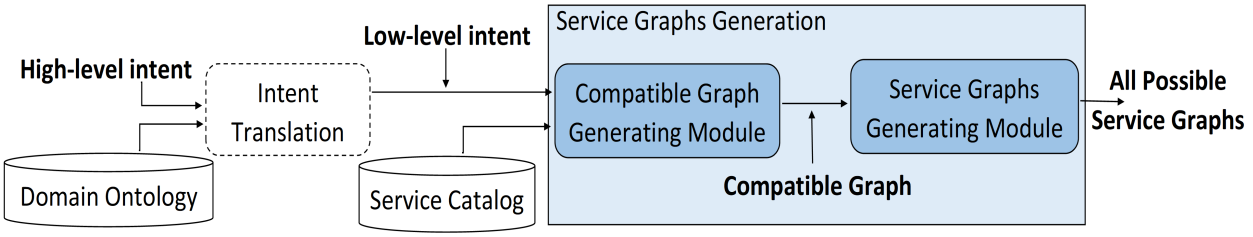



Figure 4.4: Overview of proposed Service Graphs Generation.

We use initial services as a starting point for searching the service catalog to find their compatible services. Compatibility is defined as if the output content of one service is the input content of the other, thus; the services are compatible. To cope with this, We design a service catalog that has different elements for each service. The *"Service Type"* indicates the type of services it provides. *"Service Name"* indicates the name of a service. Each *"Service Type"* can have different *"Service Names"* that perform the same functionality as it does. *"Input Interface"* is the type of content that this service accepts as input. *"Output Interface"* is the type of content that this service exports as output. *"Network Interface"* indicates the type of network that the service uses such as mobile, Ethernet, and WiFi. *"Affinity Group"* are services that should run in the same data center. *"CPU"*, *"GPU"*, *"RAM"*, and *"OS"* are the recommended resources for a service to function properly. *"Output Bandwidth"* refers to the amount of bandwidth that the service outputs. *"Latency (Execution Time)"* refers to how long it takes for a service to be executed. *"Is Physical Device?"* determines if the service is not a physical device, then needs to be placed in a data center. *"License/price ($/hour)"* indicates the cost of using a service. Table 4.1 presents the proposed service catalog for our example. This table contains various services that are needed for the HXR SPONZA Game. Services in Table 4.1 are as follows:

Table 4.1: Proposed service catalog.

| Row Number | Service Type | Service Name | Input Interface | Output Interface | Network Interface | Affinity Group | CPU | GPU | RAM | OS | Output Bandwidth | Latency (Execution Time) | Is Physical Device? | License Price ($/hour) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | IMU_CAM | Oculus- Sensors | None | imu, stereo_cam, rgb_depth | wifi | None | None | None | None | None | 6 MB/s | None | Yes | None |
| 2 | 3D_Display | Oculus-Player | hologram_stream_out | None | wifi | None | None | None | None | None | None | None | Yes | None |
| 3 | 3D_Audio | Oculus-Audio | 3D_audio | None | wifi | None | None | None | None | None | None | None | Yes | None |
| 4 | XR_APP | godot-SPONZA-OpenXR | imu_raw, slow_pos | pose_prediction, eye_buffer, audio_data | ethernet | 1 | 4 | 8 | 4 | Linux | 50 MB/s | 3 ms | No | 1 $ |
| 5 | VIO | OpenVINS | imu, stereo_cam | imu_integrator_input | ethernet | 2 | 4 | 4 | 8 | Windows/Linux | 1 KB/s | 10 ms | No | 0.15 $ |
| 6 | VIO | Kimera-VIO | imu, stereo_cam | imu_integrator_input | ethernet | 2 | 2 | 2 | 6 | Windows/Linux | 1 KB/s | 12 ms | No | 0.15 $ |
| 7 | IMU_INTEGRATOR | GTSAM | imu_integrator_input | imu_raw | ethernet | 2 | 4 | 0 | 4 | Windows/Linux | 1 KB/s | 0.04 ms | No | 0.1 $ |
| 8 | IMU_INTEGRATOR | RK4 | imu_integrator_input | imu_raw | ethernet | 2 | 2 | 0 | 4 | Windows/Linux | 1 KB/s | 0.05 ms | No | 0.12 $ |
| 9 | SLAM | ORBSLAM | imu, stereo_cam, rgb_depth | slow_pose, imu_raw, scene_map | ethernet | None | 4 | 2 | 6 | Linux | 1 KB/s | 0.11 ms | No | 0.2 $ |
| 10 | TIME_WARP | vp_matrix_reprojection | pose_prediction, eye_buffer | hologram_in, texture_pose | ethernet | 1 | 4 | 6 | 8 | Linux | 50 MB/s | 1 ms | No | 0.5 $ |
| 11 | HOLO_STREAMER | webrtc_holo_streamer | hologram_in | holo_stream_out | ethernet | None | 4 | 6 | 12 | Windows/Linux | 50 MB/s | 10 ms | No | 0.6 $ |
| 12 | Audio_encoder | ambisonic encoding | pose_prediction, audio_data | encoded_audio | ethernet | None | 2 | 0 | 4 | Windows/Linux | 1 MB/s | 1.5 ms | No | 0.12 $ |
| 13 | Audio_playback | binauralization | encoded_audio | 3D_audio | ethernet | None | 2 | 0 | 4 | Windows/Linux | 1 MB/s | 1 ms | No | 0.1 $ |

(1) **Oculus-Sensors:** Oculus-Sensors collect motion data from cameras and Inertial Measurement Units (IMUs) to

track the user's position and movement. This helps understand how the user is moving in real-time.

(2) **Oculus-Player:** Oculus-Player shows 3D visuals for virtual reality, updating the display based on the user's movements. It ensures a smooth and immersive visual experience.

(3) **Oculus-Audio:** Oculus-Audio creates 3D sound, making it feel like the sound is coming from different directions. It matches the audio to the user's movements for a more realistic experience.

(4) **Godot-SPONZA-OpenXR:** This is a virtual reality app built using the Godot engine and OpenXR framework. It provides an environment to test and demonstrate XR features.

(5) **OpenVINS:** OpenVINS combines camera and motion sensor data to figure out the user's position and orientation. It helps track the user accurately as they move.

(6) **Kimera-VIO:** Kimera-VIO also combines camera and motion sensor data to track the user's movements. It's designed to work well in dynamic environments.

(7) **GTSAM:** GTSAM processes IMU data to provide detailed motion tracking. It ensures precise tracking even in complex situations.

(8) **RK4:** RK4 uses a mathematical method to process IMU data, giving smooth and accurate motion tracking at high speeds.

(9) **ORBSLAM:** ORBSLAM creates a 3D map of the environment while tracking the user's position. This helps the system understand the surroundings in real-time.

(10) **vp˙matrix˙reprojection:** This service adjusts the visuals to match the user's updated position and reduce lag. It ensures the display looks correct even if there is a delay.

(11) **webrtc˙holo˙streamer:** This service streams 3D visuals over the internet in real-time. It ensures smooth and fast delivery of holographic content.

(12) **Ambisonic Encoding:** This service processes normal audio to make it sound 3D. It helps create a sound environment that matches the visuals.

(13) **Binauralization:** This service adds realistic sound effects to make the audio feel like i is coming from different directions. It aligns the sound with the user's movements.

**Algorithm 4.1** Compatible Graph Generating Module (CGGM)

**Input:** $Sources, Destinations$ //initial services from domain ontology
**Output:** $CG$ //compatible graph
1: $Q = \{\}$ //empty queue
2: $SC$ //service catalog
3: **for** $s\ in\ Sources$ **do**
4: $\quad Q.put(s)$
5: **end for**
6: **while** $Q\ not\ Empty$ **do**
7: $\quad service = Q.get()$
8: $\quad cs = SC.search(service)$
9: $\quad$ **for** $srv\ in\ cs$ **do**
10: $\quad\quad$ **if** $srv\ is\ unvisited$ **then**
11: $\quad\quad\quad Q.put(srv)$
12: $\quad\quad$ **end if**
13: $\quad\quad CG.add(srv)$
14: $\quad$ **end for**
15: **end while**
16: $v = CG.Sources$
17: **while** $v- > next\ is\ not\ CG.Destinations$ **do**
18: $\quad$ **if** $edge(v, v- > next)\ is\ backward$ **then**
19: $\quad\quad CG.removeEdge(v, v- > next)$
20: $\quad$ **end if**
21: $\quad v = v- > next$
22: **end while**
23: **return** $CG$

### 4.2.1 Compatible Graph Generating Module

Algorithm 4.1 illustrates the pseudocode of our proposed CGGM process. In the CGGM process, we use the source services (from Fig 4.3) as the initial services, which are added to a queue $Q$ of unvisited services (lines 3-5). At each step, a service from the top of queue $Q$ is selected (line 7). We need to search the service catalog to find compatible services with the selected service (line 8). To perform the search, we can use the dictionary (key-value) search algorithm. There are two situations if the selected service is not listed in the service catalog: 1- If the selected service is one of the source or destination services, there is no output service graph. 2- If the selected service is not one of the source or destination services, we output service graphs that do not contain this selected service. After finding this service in the service catalog, we add to $Q$ all the services whose input interface is compatible with the output interface of this service (lines 9-14). Let us define the Compatible Graph (CG) as a graph where services are the nodes and relationships between services are the links. We add the selected service along with its compatible services (acquired from the service catalog) to CG (line 13). The process of selecting a service from the top of the Q, searching for its compatible services in the service catalog, adding new services to queue $Q$, and updating CG are repeated until queue $Q$ becomes empty (lines 6-15). At this point, we start from the source services in CG and follow the links toward the destination services (from domain ontology) and remove backward links to prevent having a cycle in the graph (lines 16-22). The flowchart for the CGGM part also is depicted in Fig. 4.5.

According to the RDF of our example, we have one source service (i.e., Oculus-Sensors) and two destination services (i.e., Oculus-Player and Oculus-Audio). We add "Oculus-Sensors" to queue $Q$ and CG. The status of queue $Q$ then becomes Q=["Oculus-Sensors"]. We then select a service from the top of queue Q, i.e., "Oculus-Sensors". By

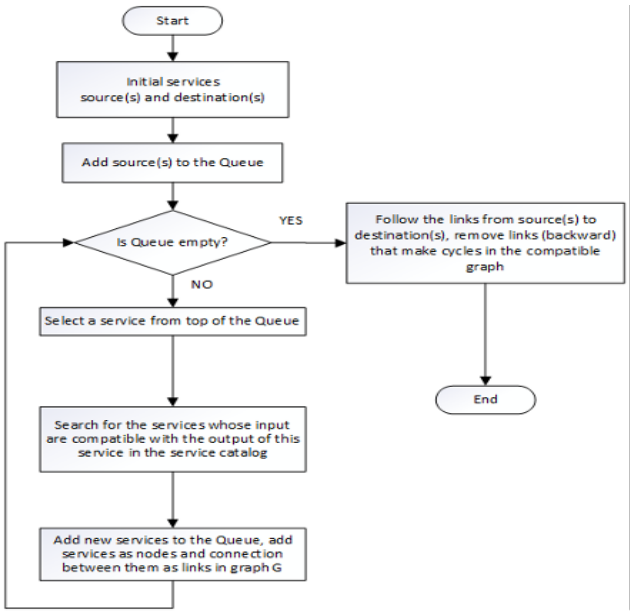



Figure 4.5: Flowchart of proposed Compatible Graph Generation Module (CGGM).

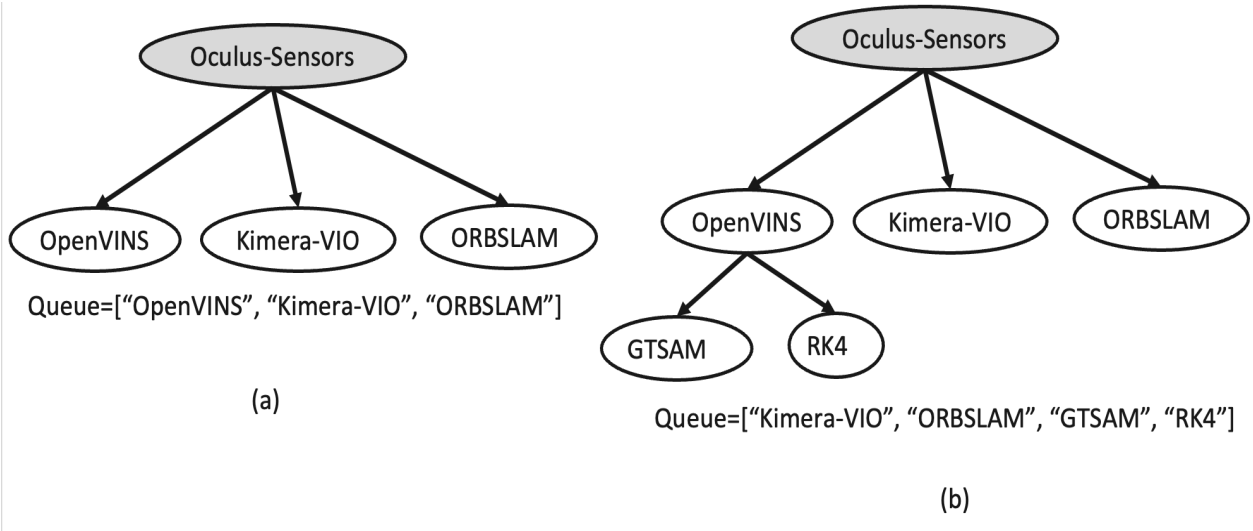



Figure 4.6: Updated CG and queue $Q$: (a) after adding the compatible services of "Oculus-Sensors", (b) after adding the compatible services of "OpenVINS".

exploring row 1 of Table 4.1 for "Oculus-Sensors", we notice that it does have any input interface (see column 4). On the other hand, it has "imu, stereo_cam, rgb_depth" as its output interface (see column 5). Thus, we need to look for the services in the service catalog to be compatible with "imu, stereo_cam, rgb_depth" as an input interface. These compatible services are "OpenVINS" (in row 5), "Kimera-VIO" (in row 6), "ORBSLAM" (row 9). We add these services to queue $Q$ and update CG. The updated Q and CG are depicted in Figure 4.6-(a).

Next, we select the next service from the top of queue $Q$, "OpenVINS". The status of queue $Q$ and CG after processing "OpenVINS" is shown in Figure 4.6-(b).

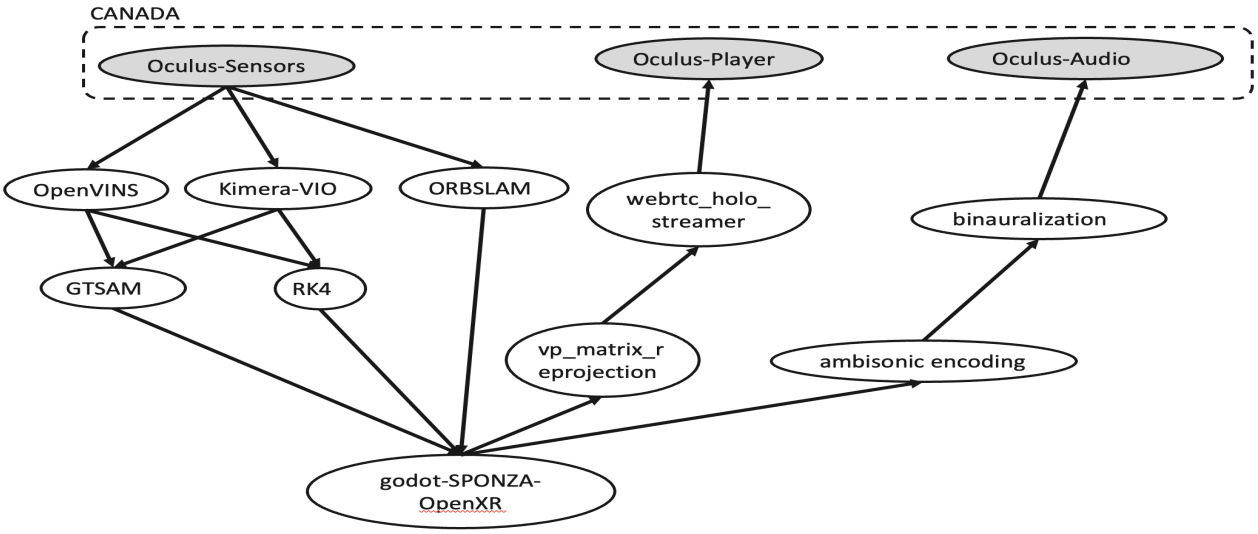



Figure 4.7: Final Compatible Graph (CG) after adding the last service in queue $Q$

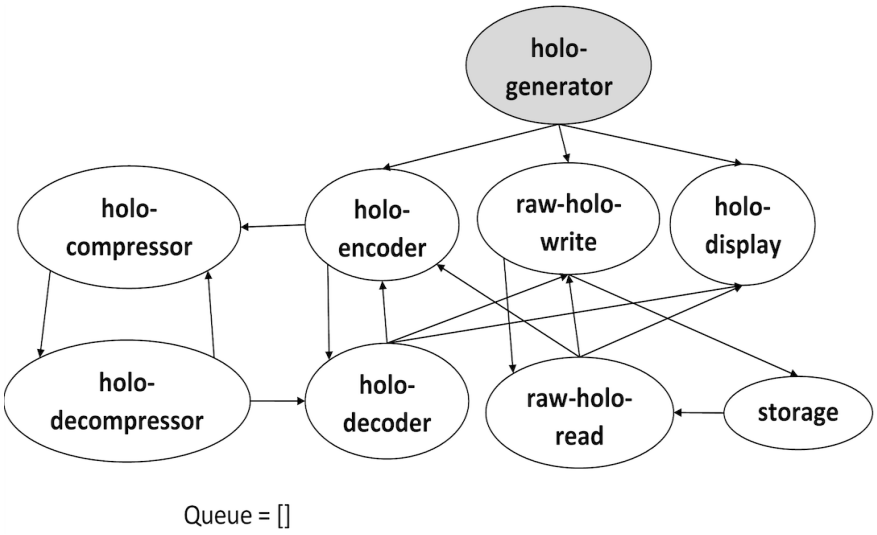



Figure 4.8: Graph CG for our motivating scenario of holographic communication use case after adding the last service in queue $Q$.

We continue this approach for all the services in the queue until it is empty. After processing all services in the Q, the status of graph CG is depicted in Figure 4.7. Next, we remove backward links (if they exist) by starting from the source services in CG and following the links toward the destination services. For example, Fig. 4.8 depict the graph CG for our motivating scenario of holographic communication (Chapter 3.1) after adding the last services in queue $Q$. In this scenario, there is one source ”holo-generator” and two destinations ”holo-display” and ”storage”. We start from the source, ”holo-generator”, and go forward through the links and nodes to the destinations, i.e., ”holo-display” and ”storage”. When we start from “holo-generator”, the next move can be “holo-encoder”. From “holo-encoder”, we can go to “holo-decoder” and from “holo-decoder” we can get back to “holo-encoder”, which is a backward link that

**Algorithm 4.2** Service Graphs Generating Module (SGGM)

**Input:** $CG$ //compatible graph
**Output:** $SGs$ //service graphs
1: **for** $src\ in\ CG.Sources$ **do**
2:  **for** $dst\ in\ CG.Destinations$ **do**
3:  $paths = CG.paths(src, dst)$
4:  **end for**
5: **end for**
6: $SGs = find\ all\ combinations\ of\ paths$
7: **for** $g\ in\ SGs$ **do**
8:  **for** $src\ in\ g.Sources$ **do**
9:  **for** $dst\ in\ g.Destinations$ **do**
10:  **if** $len(g.paths(src, dst)) > 1$ **then**
11:  $SGs.remove(g)$
12:  **end if**
13:  **end for**
14:  **end for**
15: **end for**
16: **return** $SGs$

causes a cycle in the graph. To avoid this, in our CG, we just consider the "holo-encoder" to "holo-decoder" which is a forward move starting from "holo-generator". After processing all nodes and links from the source to destination services, the output for HXR SPONZA Game is the CG which is shown in Figure 4.7.

### 4.2.2 Service Graphs Generating Module

The second part of our solution is using CG to generate Service Graphs (SGs). The pseudocode of our proposed SGGM is shown in Algorithm 4.2. In this part, we find all the paths from source to destination services in CG, then we discover all combinations of these paths which contain source and destination services (lines 1-6). When the process of finding all combinations is finished, we may have some combinations that have the same nodes and links, thus these combinations are duplicates and should be removed (lines 7-15). In each service graph, we need to show one of the ways that the source can be connected to the destination, thus if in any combination there is more than one path from source to destination, these combinations will be removed. Finally, these selected combinations are the ultimate SGs. Also, The followchart of this algorithm is depicted in Fig. 4.9. In our example, for finding all possible SGs, we first need to find all paths from each source to each destination. In Figure 4.7, we look for all paths from "Oculus-Sensors" to "Oculus-Player", and from "Oculus-Sensors" to "Oculus-Audio". All paths from "Oculus-Sensors" to "Oculus-Player" are as follows:

(1) *Oculus-Sensors* $->$ *ORBSLAM* $->$ *godot-SPONZA-OpenXR* $->$ *vp-matrix-reprojection* $->$ *webrtc-holo-streamer* $->$ *Oculus-Player*

(2) *Oculus-Sensors* $->$ *OpenVINS* $->$ *GTSAM* $->$ *godot-SPONZA-OpenXR* $->$ *vp-matrix-reprojection* $->$ *webrtc-holo-streamer* $->$ *Oculus-Player*

(3) *Oculus-Sensors* $->$ *OpenVINS* $->$ *RK4* $->$ *godot-SPONZA-OpenXR* $->$ *vp-matrix-reprojection* $->$

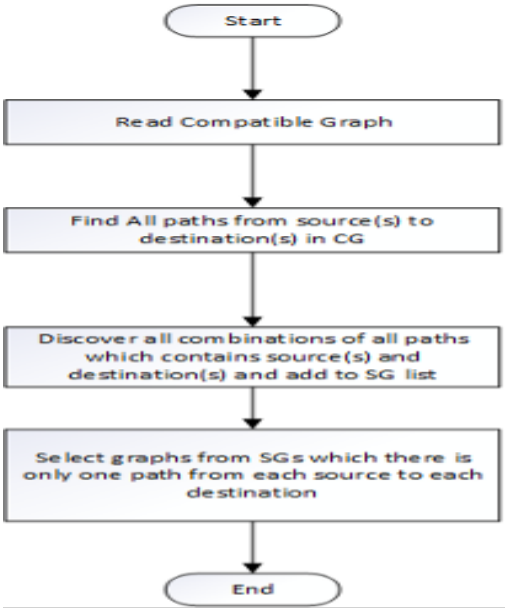



Figure 4.9: Flowchart of proposed Service Graph Generation Module (SGGM).

*webrtc-holo-streamer* $->$ *Oculus-Player*

(4) *Oculus-Sensors* $->$ *Kimera-VIO* $->$ *GTSAM* $->$ *godot-SPONZA-OpenXR* $->$ *vp-matrix-reprojection* $->$ *webrtc-holo-streamer* $->$ *Oculus-Player*

(5) *Oculus-Sensors* $->$ *Kimera-VIO* $->$ *RK4* $->$ *godot-SPONZA-OpenXR* $->$ *vp-matrix-reprojection* $->$ *webrtc-holo-streamer* $->$ *Oculus-Player*

All paths from "Oculus-Sensors" to "Oculus-Audio" are as follows:

(1) *Oculus-Sensors* $->$ *ORBSLAM* $->$ *godot-SPONZA-OpenXR* $->$ *ambisonic-encoding* $->$ *binauralization* $->$ *Oculus-Audio*

(2) *Oculus-Sensors* $->$ *OpenVINS* $->$ *GTSAM* $->$ *godot-SPONZA-OpenXR* $->$ *ambisonic-encoding* $->$ *binauralization* $->$ *Oculus-Audio*

(3) *Oculus-Sensors* $->$ *OpenVINS* $->$ *RK4* $->$ *godot-SPONZA-OpenXR* $->$ *ambisonic-encoding* $->$ *binauralization* $->$ *Oculus-Audio*

(4) *Oculus-Sensors* $->$ *Kimera-VIO* $->$ *GTSAM* $->$ *godot-SPONZA-OpenXR* $->$ *ambisonic-encoding* $->$ *binauralization* $->$ *Oculus-Audio*

(5) *Oculus-Sensors* $->$ *Kimera-VIO* $->$ *RK4* $->$ *godot-SPONZA-OpenXR* $->$ *ambisonic-encoding* $->$ *binauralization* $->$ *Oculus-Audio*

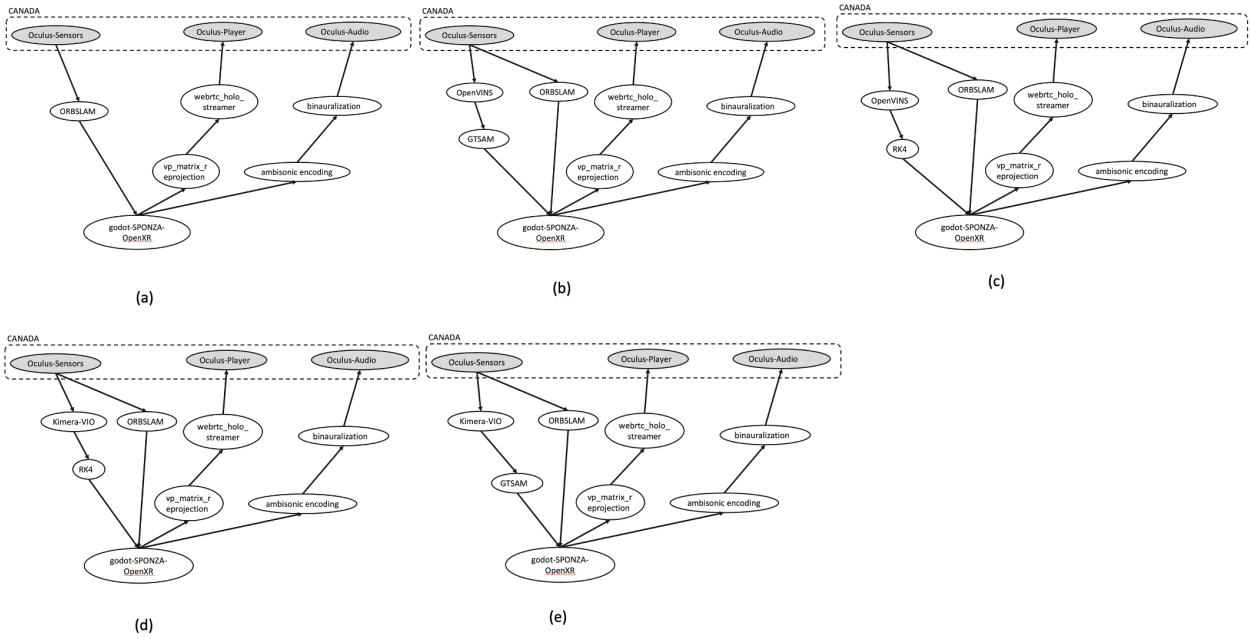



Figure 4.10: Examples of generated service graphs

Figure 4.10 shows the combination of the first path from "Oculus-Sensors" to "Oculus-Player" with each path from "Oculus-Sensors" to "Oculus-Audio". We do it for all the paths in the "Oculus-Sensors" to "Oculus-Player" paths.

Since each SG presents one of the ways that can connect the source to destination services, then we select combinations that there is only one path from each source to each destination. As shown in Figure 4.10-(b), there are two paths from "Oculus-Sensors" to "Oculus-Player". Also, for "Oculus-Sensors" to "Oculus-Audio", there are two paths. Therefore, this SG will be removed from the final SGs. Similarly, SGs (c),(d), and (e) will be removed from final SGs. The final service graphs for our example are shown in Figure 4.11.

## 4.3. Service Graph Selection with minimum Deployment Cost

In this section, we propose a solution for selecting a cost-effective service graph from all possible service graphs while considering deployment constraints. As discussed in the previous section, multiple service graphs can satisfy the high-level intent FRs. To select a single service graph, NFRs must also be considered. Therefore, this section first presents our system model and problem formulation, followed by a discussion of our proposed solution for identifying the cost-effective service graph.

### 4.3.1 System Model and Problem Formulation

In this subsection, we present our system model followed by problem formulation.

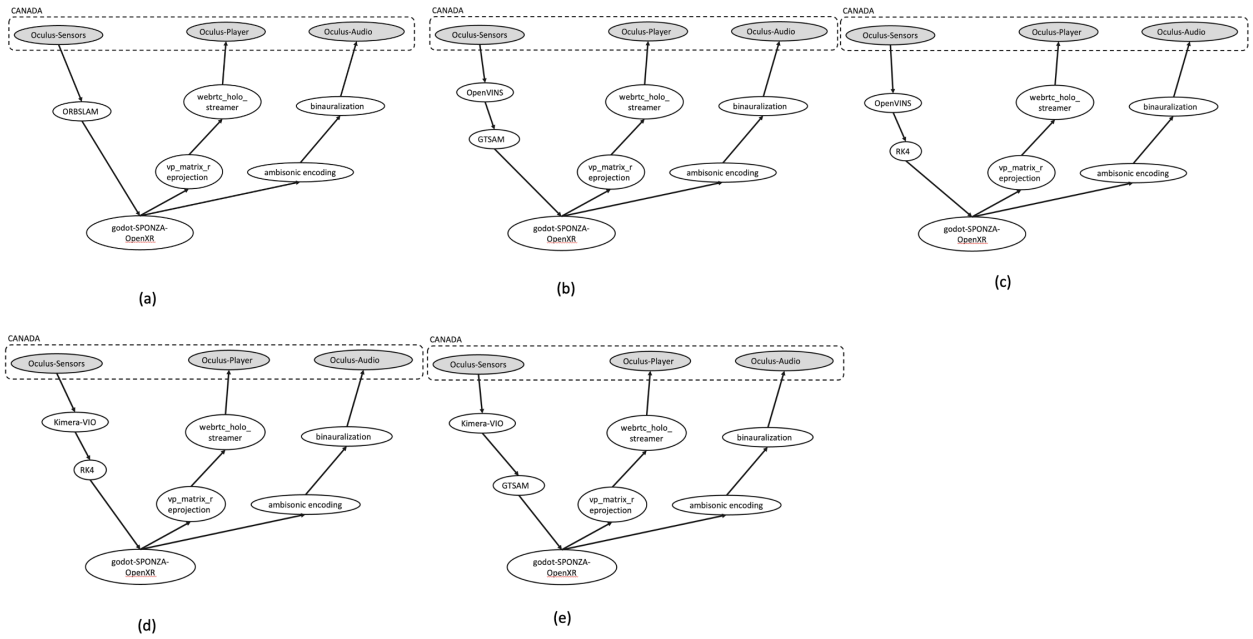



Figure 4.11: 5 Service graphs with one source "Oculus-Sensors" and two destinations "Oculus-Player" and "Oculus-Audio".

### System Model

Our system model defines the framework for selecting a service graph based on cloud consumer requirements with minimum cost. We define the low-level intent to includes NFRs extracted from intent defined by a cloud consumer. The compatible graph (CG) encompass all possible service graphs that can meet the FRs of user requests. The service catalog contains information about each service, such as its name, CPU usage, GPU usage, required operating system (OS), input/output interface, license cost, and more. This information helps in identifying the suitable data centers that can host the services. The model includes multiple data centers, which serve as the infrastructure for deploying various cloud services. Each data center has specific capabilities and resources, such as CPU, GPU, OS, etc. Additionally, the data center network comprises multiple data centers connected through links with specific bandwidth capacities. For each SG, the system evaluates the information of its services against the data center information to output one SG with the minimum deployment cost. The inputs of our system are defined as follows:

(1) **Service Graphs:** There are $N$ service graphs in compatible graph $CG = \{SG_1, \ldots, SG_N\}$, where each service graph modeled as a directed graph $SG = (V_{SG}, E_{SG})$, where $V_{SG} = \{v_1, \ldots, v_p\}$ is the set of services and $E_{SG} = \{e_1, \ldots, e_q\}$ is the set of directed links between services. Each service graph contains $K$ paths from sources to destinations $P_{SG} = \{p_1, \ldots, p_K\}$, where each path represents one way to send data from source to destination.

(2) **Low-level Intent (NFRs):** It includes NFRs of the user intent such as bandwidth, latency, reliability, security,

etc. Here, we define bandwidth as $R_{BW}$, representing the required bandwidth of user request. Latency is defined as the total time it takes for data to be send from source to destination in path $p$ as $R^p_{LAT}$.

(3) **Service Catalog:** There are $m$ services in the service catalog $SC$. Each service $s \in SC$ has specific characteristics that must be satisfied when selecting a data center. These characteristics are $R^s_{CPU}$, $R^s_{GPU}$, $R^s_{RAM}$ and $R^s_{OS}$. Moreover, there is an hourly license cost for each service $C^s_{lic}$, the service runtime latency $W^s_{LAT}$ and the required bandwidth for links $e \in E_{SG}$ denoted as $W^e_{BW}$, which is derived from the estimated input/output bandwidth of the service listed in the service catalog. The service catalog is dynamic, meaning it is continuously updated to reflect the availability of services in real-time. New services are added as they become available, while outdated or unavailable services are removed.

(4) **Data Center Network:** Our data center network is modeled as an undirected graph $DC = (N, L)$, where $N = \{n_1, \ldots, n_R\}$ is the set of data centers and $L = \{l_1, \ldots, l_s\}$ is the set of link between them. Each link $l \in L$ has a bandwidth capacity $W^l_{BW}$ and a latency $W^l_{LAT}$. Each data center $n \in N$ has different flavors. Let $N.F$ denote the set of all flavors associated with the set $N$, and let $n.F$ indicate the set of flavors available at data center $n$, hence $n.F \subseteq N.F$. Each flavor of a data center $n.F$ has various capacity characteristics such as $CAP^{n.F}_{CPU}$, $CAP^{n.F}_{GPU}$, $CAP^{n.F}_{RAM}$ and $CAP^{n.F}_{OS}$. Additionally, there is an hourly usage cost for the data center flavor $C^{n.F}_{DC}$ and $C^l_{BW}(d)$ represent the bandwidth cost for transferring $d$ units of data between data centers, where the cost is calculated based on the amount of data transferred in one hour. Each service in the service catalog can be deployed in any data center that meets its requirements.

### 4.3.2 Problem Formulation

In the following, we formulate the service graph selection problem as an ILP by defining the following decision variables:

$$X_{v,n.F} = \begin{cases} 1, & \text{if service } v \in V_{SG} \text{ is mapped to flavor} \\ & f \text{ of data center } n \text{ where } n.F \subseteq N.F, \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

and

$$X_{e,l} = \begin{cases} 1, & \text{if } e \in E_{SG} \text{ is mapped to link } l \in L, \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

Next, we explain our cost model followed by the objective function of our ILP.

(1) **Cost:** Total cost is the summation of the following three components.

- **Data Center Cost** $(C^{dc})$**:** $C^{dc}$ represents the total hourly cost of using data center flavors $n.F \subseteq N.F$ for running each service $v \in N_{SG}$ mapped to data centers given by:

$$C^{dc} = \sum_{v \in V_{SG}} \sum_{n.F \subseteq N.F} C^{n.F}_{DC} \, X_{v,n.F} \, . \tag{4.3}$$

- **Service License Cost** $(C^{lic})$**:** $C^{lic}$ represents the hourly cost of using service $v \in V_{SG}$, as specified in the service catalog. We estimate $C^{lic}$ as follows:

$$C^{lic} = \sum_{v \in V_{SG}} C^{v}_{lic}. \tag{4.4}$$

- **Bandwidth Cost** $(C^{bw})$**:** $C^{bw}$ represents the total hourly cost of transferring data between data centers when mapping links $e \in E_{SG}$ between services in the SG to data center links $l \in L$. We obtain $C^{bw}$ as follows:

$$C^{bw} = \sum_{e \in E_{SG}} \sum_{l \in L} C^{l}_{BW}(d) \, X_{e,l} \, . \tag{4.5}$$

- **Total Cost** $(C^{tot})$**:** Total hourly cost $(C^{tot})$ is the summation of the above mentioned three cost components:

$$C^{tot} = C^{dc} + C^{lic} + C^{bw}. \tag{4.6}$$

(2) **Objective Function and Constraints:** In our problem, we aim to select the service graph with the minimum deployment cost among possible service graphs while respecting latency and bandwidth requirements of the intent. We define our objective function as follows:

$$\underset{SG \in CG}{arg\,min} C^{tot}, \tag{4.7}$$

which aims to minimize the total costs of data center usage, service licenses, and bandwidth.

**Constraints:** First, the latency and bandwidth requirements of the intent must be satisfied. Each path from source to destination in a service graph has a latency requirement. The total latency of a path is the summation of the service runtime latency and the latency incurred during data transfer between data centers. This constraint is imposed as follows:

$$\sum_{s \in p} W^{s}_{LAT} + \sum_{e \in p} \sum_{l \in L} W^{l}_{LAT} \, X_{e,l} \leq R^{p}_{LAT} \quad \forall \, p \in P_{SG} \tag{4.8}$$

To meet the bandwidth requirement $R_{BW}$, the total bandwidth associated with the mapping between the service

graph and the data centers must exceed this specified requirement:

$$\sum_{l \in L} W^{l}_{BW} \, X_{e,l} \geq R_{BW} \quad \forall \, e \in E_{SG}, l \in L \tag{4.9}$$

Moreover, a service graph can be mapped to a data center network if all the following constraints in equations (4.10) and (4.11) be satisfied:

$$\forall \, v \in V_{SG}, n.F \subseteq N.F \tag{4.10}$$

$$R^{v}_{CPU} \cdot X_{v,n.F} < CAP^{n.F}_{CPU},$$

$$R^{v}_{GPU} \cdot X_{v,n.F} < CAP^{n.F}_{GPU},$$

$$R^{v}_{RAM} \cdot X_{v,n.F} < CAP^{n.F}_{RAM},$$

$$R^{v}_{OS} \cdot X_{v,n.F} \subset CAP^{n.F}_{OS}$$

which show that to map a service to a data center flavor, all the required CPU, GPU, RAM, and OS specifications of the service must be matched by the corresponding resource capacities in the data center.

Equation (4.11) ensures that when mapping link $e$ in the service graph $E_{SG}$ to link $l$ between data centers in $L$, the bandwidth requirement $W^{e}_{BW}$ of the service graph link does not exceed the available bandwidth $W^{l}_{BW}$ on the corresponding data center link.

$$W^{e}_{BW} \, X_{e,l} \leq W^{l}_{BW} \quad \forall \, e \in E_{SG}, l \in L \tag{4.11}$$

### 4.3.3 Proposed Solution for Service Graph Selection with Minimum Deployment Cost

In this subsection, we first describe our proposed solution using flowcharts. Next, we discuss the pseudocode of the solution. Finally, we explain each step of the solution in the context of our HXR SPONZA Game use case.

#### Proposed Solution Flowchart

Here, we present our proposed solution for the service graph selection. We will illustrate our solution in 5 flowcharts and will explain them in detail. Figure 4.12 shows the main steps of our proposed solution. The description for each step is as follows:

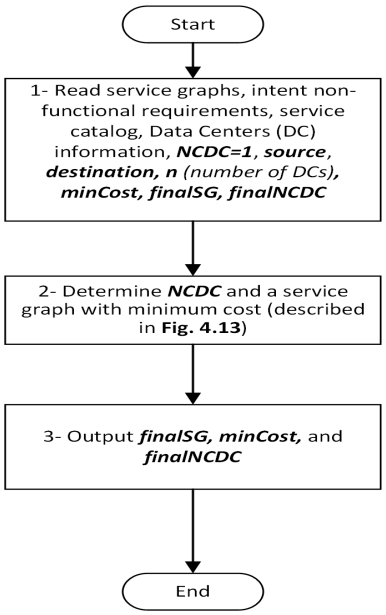



Figure 4.12: Flowchart of main steps of the proposed solution.

(1) The first step is reading the inputs that are:

- The intent non-functional requirements (e.g., latency, bandwidth, etc.).
- The service catalog includes information about services (e.g., CPU, GPU, OS usage, and license cost) and input/output interfaces.
- The infrastructure information contains datacenter resources capacities (e.g., CPU, GPU, OS, etc.), cost, bandwidth, and latency of links between data centers.
- The Number of Candidate Data Centers (NCDC) is an input parameter specifying the number of possible datacenters where a service could be mapped.
- The variable N is the number of all available data-centers.
- The set of input service graphs.

(2) Using the input data, the solution determines NCDC and find a service graph with minimum cost. We will discuss the details in Figure 4.13.

(3) The output of our solution is the minCost, finalSG, and finalNCDC. The minCost, finalSG, and finalNCDC are variables that store the minimum cost, final service graph, and the required value for NCDC to get the minimum cost, respectively.

Figure 4.13 shows the steps of determining NCDC and finding a service graph with minimum cost. The description for each step is as follows:

(1) The inputs are the set of service graphs, the values of NCDC and N. Moreover, the minCost, finalSG, and finalNCDC are variables that store the final output. Before starting, our solution identifies the required value of "n" for NCDC.

(2) Next, it adds all services in the set services graph to a list L.

(3) It finds NCDC candidate data centers for all services in L. The details of the steps are described in Figure 4.14.

(4) It finds a service graph with minimum cost. The details of the steps are described in Figure 4.15.

(5) For different values of NCDC, it compares the cost of the obtained service graph with the minCost, if it is smaller than minCost, update minCost, finalSG, and finalNCDC.

(6) Finally, it outputs minCost, finalSG, and finalNCDC.

Figure 4.14 illustrates the steps followed by our solution to find the number of candidates data-centers to be checked for services in the service graphs. The description for each step is as follows:

(1) The inputs are all services in the service graphs stored in list L, intent non-functional-requirements, service catalog, and data center information. The output is the value NCDC to be checked for service graph selection. Our solution starts by initializing the variable "i" with 0. Next, it keeps iterating until "I" is less than the size of L, and it repeats steps 2-4.

(2) It finds all data centers that can meet the requirements of service "L[i]" (e.g., CPU, GPU, OS, etc.) mentioned in the service catalog.

(3) Among these data centers, it selects NCDC of them which have the lowest cost. Then, it adds the selected data centers for L[i] to a ServiceToDC dictionary where L[i] is key and data centers are values.

(4) When i is bigger than the size of L, it outputs ServiceToDC.

Figure 4.15 shows the steps of finding a service graph with minimum cost. The description for each step is as follows:

(1) The inputs are ServiceToDC, the input service graphs (alongside with their corresponding source and destination services). For each service graph, it repeats steps 2 and 3. Our solution starts by initializing variables j and k with 0. Next, it iterates while "j is less than the size of source" and "k is less than the size of destination" and it does steps 2-3.

(2) Then, it finds all paths from source[j] to destination[k] in service graphs and adds them to Paths variable.

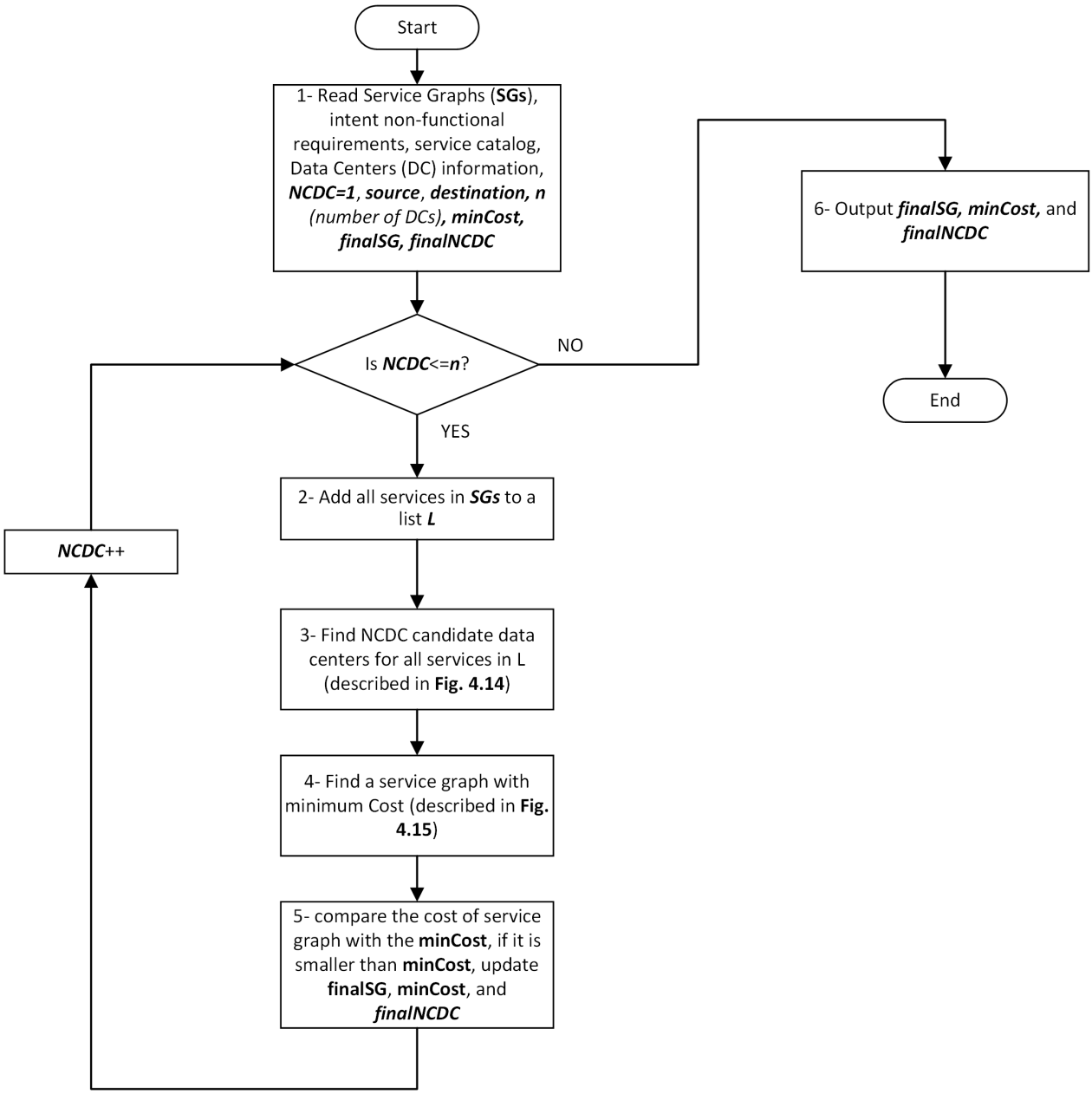



Figure 4.13: Flowchart of determining NCDC and finding a service graph with minimum cost.

(3) It uses SelectLowestCostPath function (described in detail in Figure 4.16) to select one path in Paths variable with the lowest cost. It adds selected path to FinalPaths variable.

(4) When k is bigger than the size of destination and j is bigger than the size of source, our solution combines all paths in FinalPaths to generate a service graph and add up the cost of each path to get the total cost.

(5) It outputs the service graph and the total cost.

Figure 4.16 shows the steps of SelectLowestCostPath function. The description for each step is as follows:

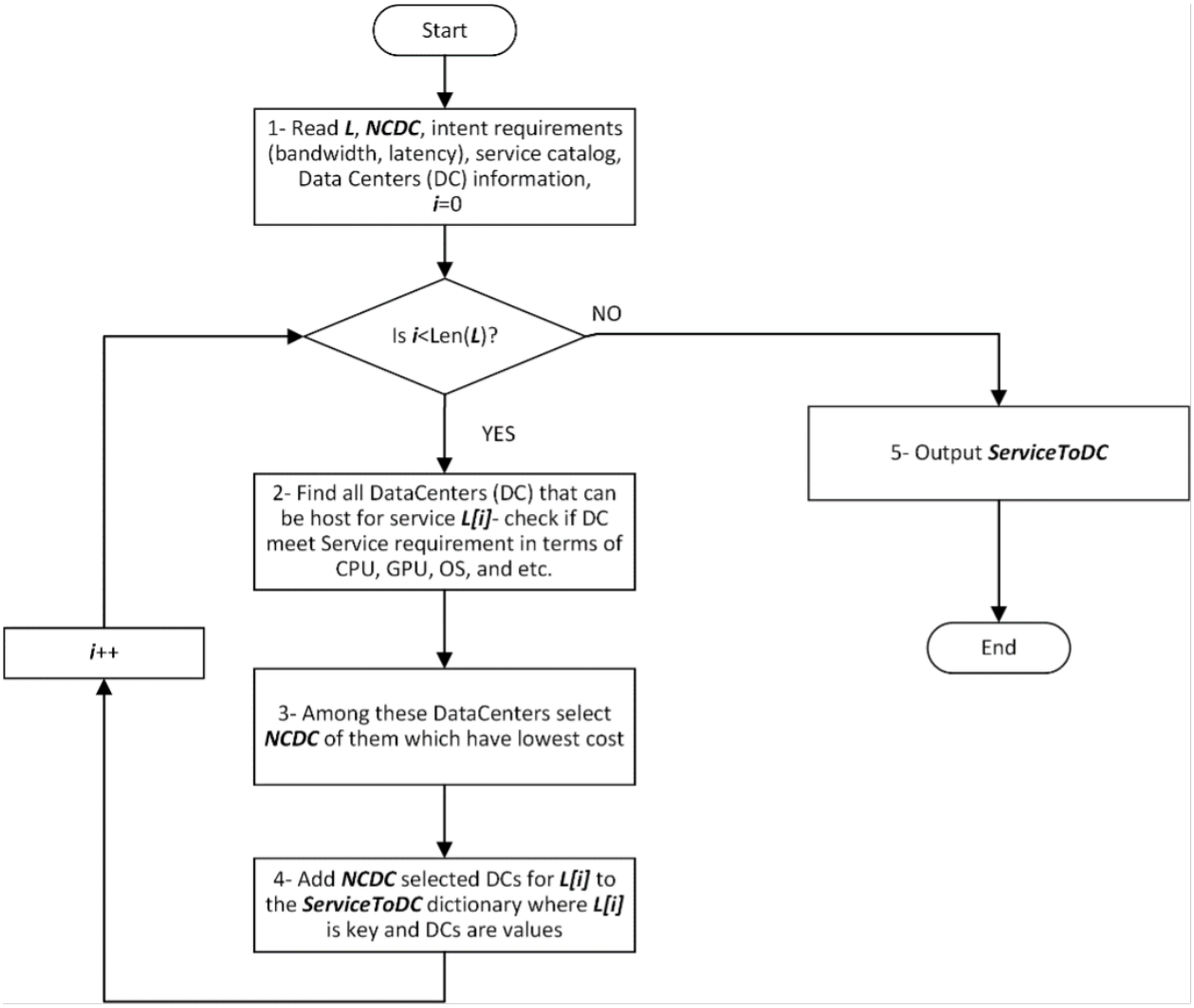



Figure 4.14: Flowchart of finding candidate data centers for services in the service graphs.

(1) The inputs are Paths, ServiceToDC, intent non-functional-requirements, service catalog, and data centers information. Our solution initializes variable i with 0 and a Decision Tree (DT) with null. While i is less than the size of Paths and for all services (variable j) in Path[i], it repeats steps 2-3.

(2) Find all data centers from ServiceToDC where Path[i][j] is the key and add them to DT.

(3) Connect these new data centers to path[i][j-1] added data centers to DT (if exist in DT), check they meet the intent requirements (latency, bandwidth, etc.) and update the cost of the path (cost = data center cost + service license cost + bandwidth cost between each two data centers in the path).

(4) When i is bigger than the size of Paths, it selects a path with minimum cost among all combinations of services in DT.

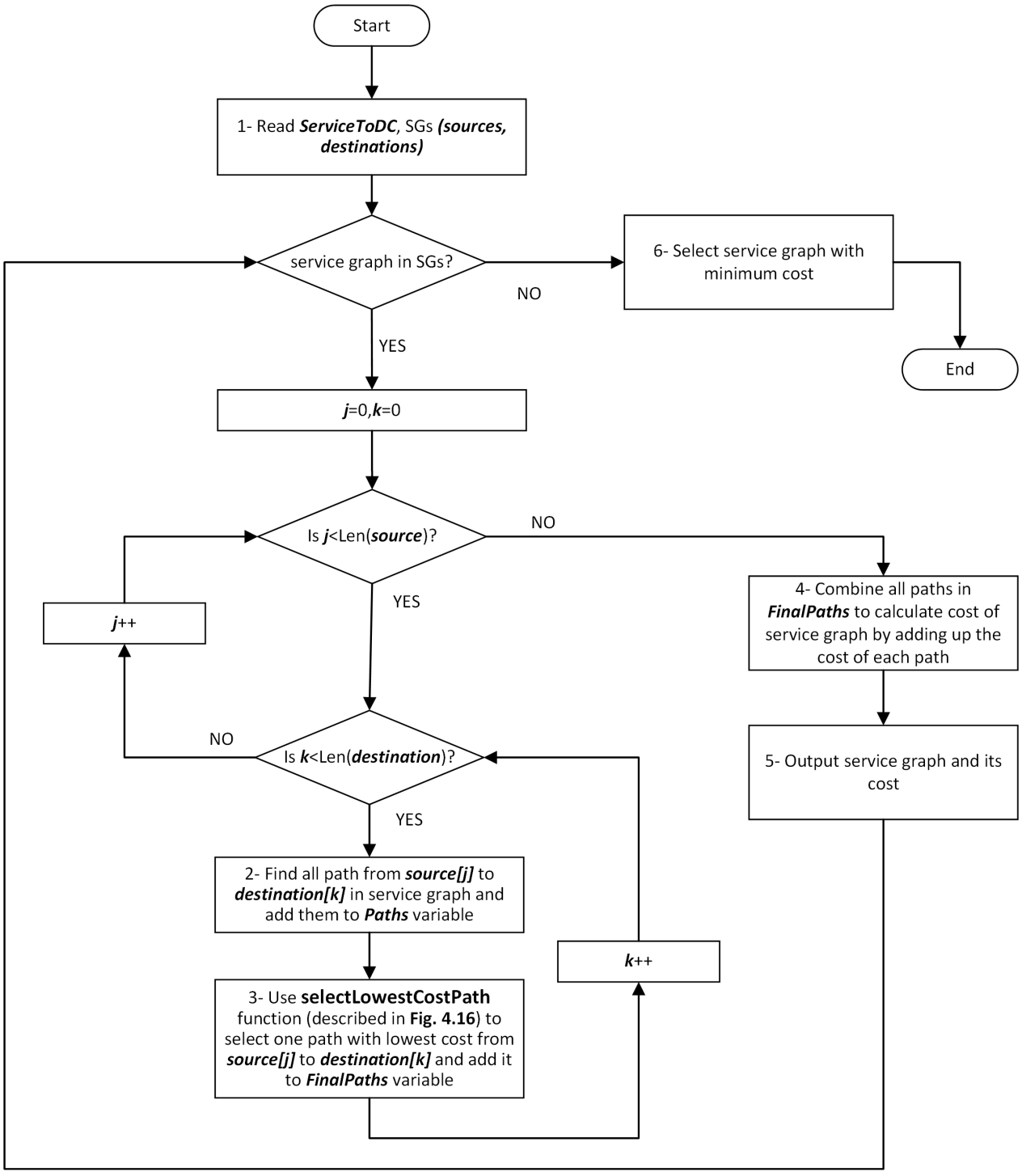



Figure 4.15: Flowchart of finding a service graph with minimum cost.

**Proposed Solution Pseudocode**

Figure 4.17 depicts an overview of our proposed solution. As shown in Fig. 4.17, our proposed Service Graph Selection (SGS) algorithm receives the NFRs along with the compatible graph, service catalog, and data center network

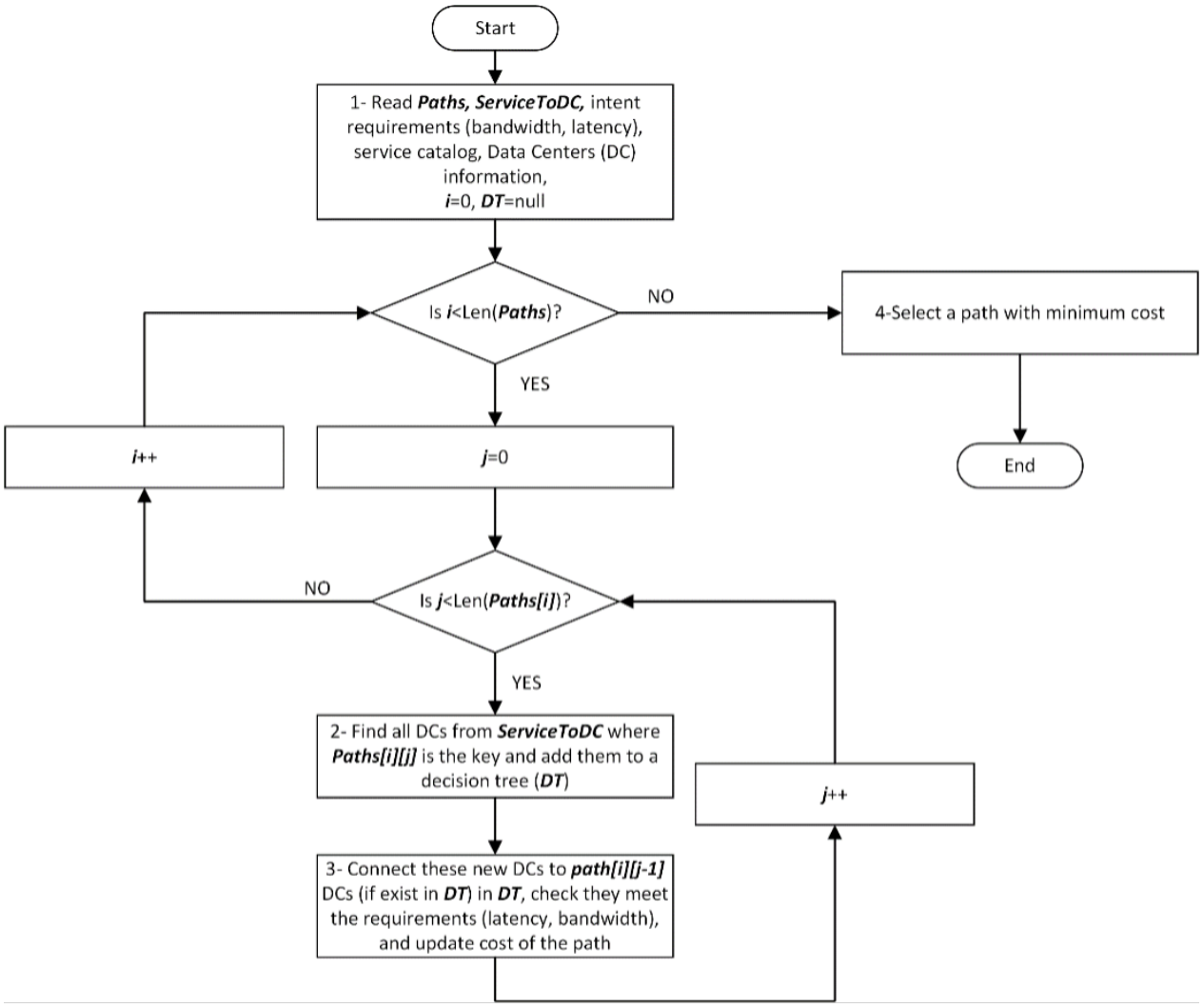



Figure 4.16: Flowchart of SelectLowestCostPath function.

as the input. The output of our proposed SGS algorithm is a service graph with the minimum deployment cost along with its cost and placement within the distributed data center network. Algorithm 4.3 illustrates the pseudocode of our proposed SGS algorithm. Let us define the number $N_D$ of candidate data centers to represent the number of potential data centers where a service can be mapped. It can range from 1 up to the total number of data centers available in the network. We will further elaborate on the role of $N_D$ in Chapter 5. Algorithm 4.3 begins by identifying $N_D$ candidate data centers for all services in the compatible graph using the $findCandDCs()$ function (lines 5-7). It then finds all the paths from each source to destination in the compatible graph and adds them to a list $P$ (lines 10-15). The cost and placement of each path in $P$ are calculated using the $calcPathCost()$ function, and the path with the minimum cost and potential placement are added to list $listMinPaths$ and list $listPotPlacements$, respectively (lines 16-27). Finally, all the paths in $listMinPaths$ are merged to generate the SG, summing the cost of each path while avoiding duplicates in services or links to compute the total cost (lines 28-29). Similarly, all the placements in

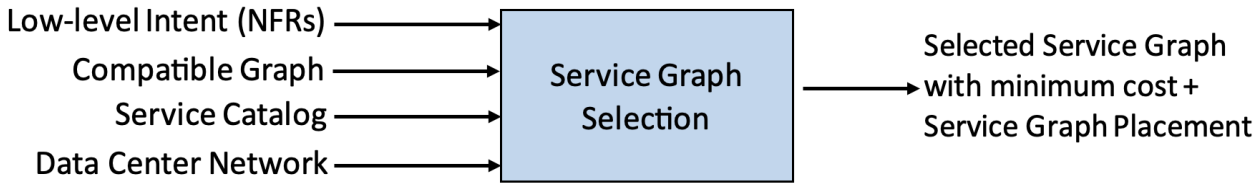



Figure 4.17: Overview of our proposed solution.

**Algorithm 4.3** Service Graph Selection (SGS)

**Input:** $CG, DC, SC, NFRs, N_D, Sources, Destinations$
**Output:** $minSG, minCost, finalPlacement$ //service graph with minimum deployment cost and placement in distributed data centers network

1: $minSG = \{\}$
2: $minCost = \infty$
3: $finalPlacement = \{\}$
4: $candDCs = \{\}$
5: **for** $s$ *in* $CG$ **do**
6: $\quad candDCs[s] = findCandDCs(s, N_D)$ //Algorithm 2
7: **end for**
8: $listMinPaths = \{\}$
9: $listPotPlacements = \{\}$
10: **for** $src$ *in* $Sources$ **do**
11: $\quad$ **for** $dst$ *in* $Destinations$ **do**
12: $\quad\quad minPathCost = \infty$
13: $\quad\quad minPath = \{\}$
14: $\quad\quad potPlacement = \{\}$
15: $\quad\quad P = P_{CG}.search(src, dst)$
16: $\quad\quad$ **for** $p$ *in* $P$ **do**
17: $\quad\quad\quad cost, placement = calcPathCost(p, candDCs)$ //Algorithm3
18: $\quad\quad\quad$ **if** $cost < minPathCost$ **then**
19: $\quad\quad\quad\quad minPathCost = cost$
20: $\quad\quad\quad\quad minPath = p$
21: $\quad\quad\quad\quad potPlacement = placement$
22: $\quad\quad\quad$ **end if**
23: $\quad\quad$ **end for**
24: $\quad\quad listMinPaths.add(minPath)$
25: $\quad\quad listPotPlacements.add(potPlacement)$
26: $\quad$ **end for**
27: **end for**
28: $minSG = combine\ all\ paths\ in\ listMinPaths$
29: $minCost = sum\ cost\ of\ all\ paths\ in\ listMinPaths$ //avoid duplicate summation
30: $finalPlacement = combine\ all\ placements\ in\ listPotPlacements$
31: **return** $minSG, minCost, finalPlacement$

$listPotPlacements$ are merged to generate the final placement (line 30).

The pseudocode of function $findCandDCs()$ is shown in Algorithm 4.4. The required resources of service $s$ (e.g., CPU, GPU, RAM, OS) are examined against the capacities of each data center flavor, and those that satisfy the resource requirements of the service are added to list $candDCs$ (lines 2-6). List $candDCs$ is then sorted in ascending order of data center cost, and the first $N_D$ data centers are selected (lines 7-8).

The pseudocode of function $calcPathCost()$ is shown in Algorithm 4.5. We build a Decision Tree (DT) to select the best mapping of path $p$ to the data center network with the minimum cost. The decision tree is a tree structure used for making decisions based on data. Each node of the tree represents a decision point based on the value of a specific attribute, each branch represents the outcome of that decision, and each leaf node represents a final decision. For each

**Algorithm 4.4** findCandDCs(): Find Candidate Data Centers

**Input:** $s, N_D$
**Output:** $candDCs$ //top $N_D$ data centers with lower cost for service s
1: $candDCs = \{\}$
2: **for** $n.F$ *in* $N.F$ **do**
3: **if** $constraints\ in\ equation\ (10)\ is\ satisfied$ **then**
4: $candDCs.add(n.F)$
5: **end if**
6: **end for**
7: $candDCs.sort(asc)$ // sort in ascending order based on data center cost
8: **return** $candDCs.slice(N_D)$ //select first $N_D$ items

**Algorithm 4.5** calcPathCost(): Calculate the Path Cost

**Input:** $p, candDCs$
**Output:** $pathCost, placement$
1: $pathCost = 0$
2: $placement = \{\}$
3: $DT = \{\}$ //Decision Tree
4: **for** $s$ *in* $p$ **do**
5: **for** $n.F$ *in* $candDCs[s]$ **do**
6: **if** $check\ constraints\ (8)\ and\ (11)$ **then**
7: $DT.add(s, n.F)$
8: $DT.cost(s, n.F) = update\ cost\ using\ equation\ (6)$
9: $DT.placement(s, n.F) = store\ all\ placements\ till\ this\ node$
10: **end if**
11: **end for**
12: **end for**
13: $pathCost = findMin(DT.cost)$ //find minimum cost among destination nodes in DT
14: $placement = DT.placement$ //find DT.placement that corresponds to the minimum cost node in DT
15: **return** $pathCost, placement$

service in path $p$, all its possible deployment data centers in $candDCs$ along with the link between services are added to DT while respecting user bandwidth and latency requirements (lines 4-12). If a branch does not satisfy user NFRs, it will be pruned, and the costs of the remaining branches will be updated using Eq. 4.6. Finally, we find the minimum cost among the leaf nodes, which represent the cost and possible placement for path $p$ (lines 13-14). Consequently, the minimum $pathCost$ and $placement$ are returned (line 15).

### Illustrative Example

Here, we explain the proposed solution using our example, the HXR SPONZA Game. Based on Fig. 4.17, our solution has four inputs: NFRs, a compatible graph, a service catalog, and the data center network.

From the requested high-level intent, we know that the HXR SPONZA Game should operate in ”low interaction” and ”medium quality” modes. After translating the high-level intent using the domain ontology, ”interaction” and ”quality” are mapped to latency and bandwidth requirements. Therefore, our NFRs, as shown in the RDF in Fig. 4.3, are a latency of less than 120 ms and a bandwidth of 5 MB/s or greater. Additionally, the location for playing the HXR SPONZA Game is Ericsson Canada, which translates to the data center locations for the source and destination services.

Now, we have both the FR and NFR requirements extracted from the user’s high-level intent. To proceed with the

solution, we need the service catalog and data center information. Table 4.1 presents some of the services available in the service catalog.

Table 4.2 presents data centers information. Each data center has some flavors with different resources. For each flavor, there is available capacity in terms of CPU, CPU Type, GPU, RAM, and OS. Moreover, each flavor has a price, and each data center has a price for consumed bandwidth for output data.

Table 4.2: Data Center Catalog.

| Row Number | Data Center Name | Flavor Name | CPU | CPU Type | GPU | RAM | OS | Flavor Price | Output Bandwidth Price/GB |
|---|---|---|---|---|---|---|---|---|---|
| 1 | DCJapan | flavor1 | 16 | Intel Xeon | 0 | 64 | Windows | 4.1 $ | 0.09 $ |
| 2 | DCJapan | flavor2 | 32 | Intel Xeon | 0 | 128 | Windows | 8.01 $ | 0.09 $ |
| 3 | DCCanada | flavor1 | 48 | Intel Xeon | 0 | 192 | Windows | 2.47 $ | 0.02 $ |
| 4 | DCCanada | flavor2 | 16 | Intel Xeon | 8 | 32 | Linux | 2.1 $ | 0.02 $ |
| 5 | DCCanada | flavor3 | 12 | Intel Xeon | 6 | 16 | Linux | 1.2 $ | 0.02 $ |
| 6 | DCGermany | flavor1 | 16 | AMD | 0 | 32 | Windows | 0.64 $ | 0.02 $ |
| 7 | DCGermany | flavor2 | 48 | AMD | 0 | 192 | Windows | 1.7 $ | 0.02 $ |
| 8 | DCSweden | flavor1 | 16 | Intel Xeon | 8 | 32 | Linux | 1.3 $ | 0.02 $ |
| 9 | DCSweden | flavor2 | 128 | Intel Xeon | 12 | 512 | Linux | 2.1 $ | 0.02 $ |
| 10 | DCUSA | flavor1 | 4 | AMD | 8 | 16 | Linux | 0.4 $ | 0.02 $ |
| 11 | DCUSA | flavor2 | 4 | AMD | 8 | 8 | Linux | 0.3 $ | 0.02 $ |
| 12 | DCUSA | flavor3 | 16 | AMD | 6 | 32 | Linux | 0.5 $ | 0.02 $ |

Figure 4.18 shows the data centers network. It shows the connection between data centers and the latency and bandwidth of links between them.

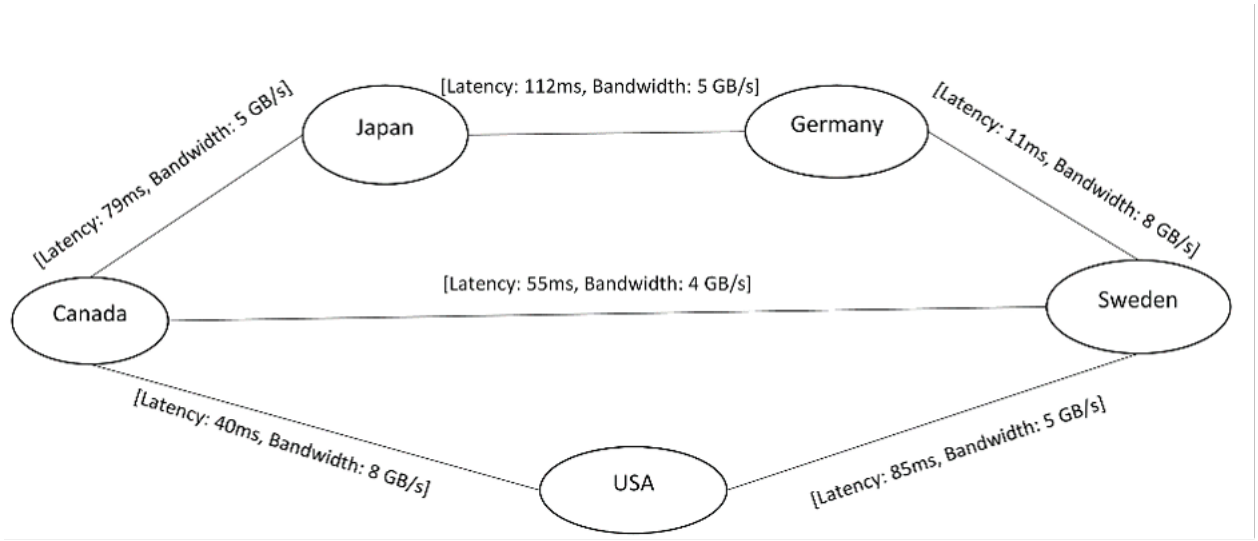



Figure 4.18: Data Centers Network.

Finally, we have the compatible graph generated in section 4.2. Figure 4.19 shows the compatible graph for HXR SPONZA game. Oculus-Sensor is source and Oculus-Player and Oculus-Audio are destinations. Each link has a

number which implies the required bandwidth for sending data.

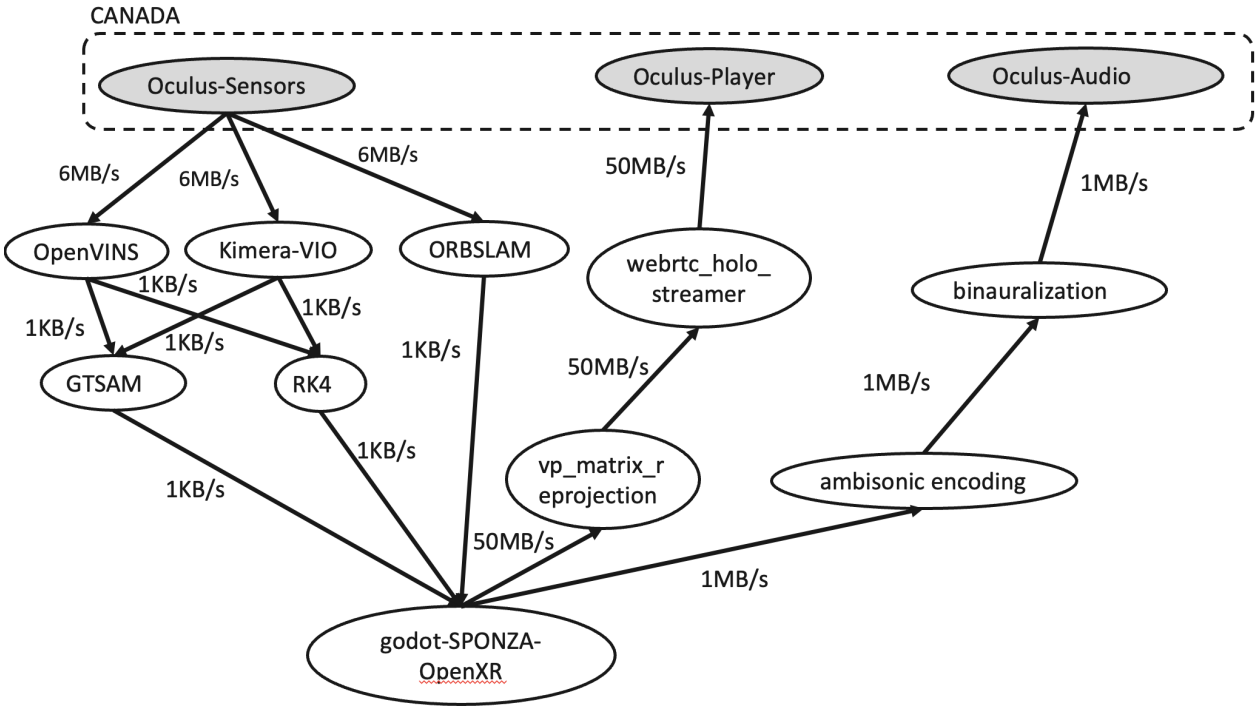



Figure 4.19: HXR SPONZA Game Compatible Graph.

Now, we have all the required information to start our proposed solution. We consider $N_D = 3$. Next, we find candidate data centers for services in the compatible graph, calculate the cost of each service graph, and output a minimum-cost service graph. The steps for finding candidate data centers are depicted in Algorithm 4.4. We compare the required resources (CPU, GPU, RAM, OS) for each service of the compatible graph with available resources in the data centers catalog (Table 4.2) and select data centers that can satisfy the required resources. Among those selected data centers, we select $N_D = 3$ data centers with the lowest cost. Note that we find data centers for service, not physical devices. For example, the source and destinations in this example are physical devices, thus they do not need to be mapped. The 3 candidate data centers for each service in the compatible graph are shown in Table 4.3.

Next, we should calculate the cost of each service graph in compatible graph. There are 5 service graphs in the compatible graph depicted in Figure 4.19. These service graphs are shown in Figure 4.20

We explain the cost calculation for the service graph in Figure 4.20-(e). The same approach can be used for the other service graphs in Figure 4.20. First, we find all the paths between source and destination. There is one path from “Oculus-Sensors” to “Oculus-Player” and one path from “Oculus-Sensors” to “Oculus-Audio”. We calculate the cost for each path using a decision tree and consider all possible combinations of data centers while ensuring the requirements are satisfied. Part of the decision tree for the path from ”Oculus-Sensors” to ”Oculus-Player” is shown in Fig. 4.21. Based on domain ontology, the source and destination are located in Canada, the Latency is lower than 120

Table 4.3: Candidate Data Centers for Service in Compatible Graph.

| Service Name | Data Center 1 | Data Center 2 | Data Center 3 |
|---|---|---|---|
| OpenVINS (Table 4.1 row 5) | DCCanada, flavor 3 (Table 4.2 row 5) | DCSweden, flavor 1 (Table 4.2 row 8) | DCUSA, flavor 2 (Table 4.2 row 11) |
| Kimera-VIO (Table 4.1 row 6) | DCCanada, flavor 3 (Table 4.2 row 5) | DCSweden, flavor 1 (Table 4.2 row 8) | DCUSA, flavor 2 (Table 4.2 row 11) |
| GTSAM (Table 4.1 row 7) | DCCanada, flavor 3 (Table 4.2 row 5) | DCUSA, flavor 2 (Table 4.2 row 11) | DCGermany, flavor 1 (Table 4.2 row 6) |
| RK4 (Table 4.1 row 8) | DCCanada, flavor 3 (Table 4.2 row 5) | DCUSA, flavor 2 (Table 4.2 row 11) | DCGermany, flavor 1 (Table 4.2 row 6) |
| ORBSLAM (Table 4.1 row 9) | DCCanada, flavor 3 (Table 4.2 row 5) | DCSweden, flavor 1 (Table 4.2 row 8) | DCUSA, flavor 2 (Table 4.2 row 11) |
| godot-SPONZA-OpenXR (Table 4.1 row 4) | DCCanada, flavor 2 (Table 4.2 row 4) | DCSweden, flavor 1 (Table 4.2 row 8) | DCUSA, flavor 2 (Table 4.2 row 11) |
| vp˙matrix˙reprojectin (Table 4.1 row 10) | DCCanada, flavor 3 (Table 4.2 row 5) | DCSweden, flavor 1 (Table 4.2 row 8) | DCUSA, flavor 2 (Table 4.2 row 11) |
| webrtc˙holo˙streamer (Table 4.1 row 11) | DCCanada, flavor 3 (Table 4.2 row 5) | DCSweden, flavor 1 (Table 4.2 row 8) | DCUSA, flavor 1 (Table 4.2 row 10) |
| ambisonic encoding (Table 4.1 row 12) | DCCanada, flavor 3 (Table 4.2 row 5) | DCUSA, flavor 2 (Table 4.2 row 11) | DCGermany, flavor 1 (Table 4.2 row 6) |
| Binauralization (Table 4.1 row 13) | DCCanada, flavor 3 (Table 4.2 row 5) | DCUSA, flavor 2 (Table 4.2 row 11) | DCGermany, flavor 1 (Table 4.2 row 6) |

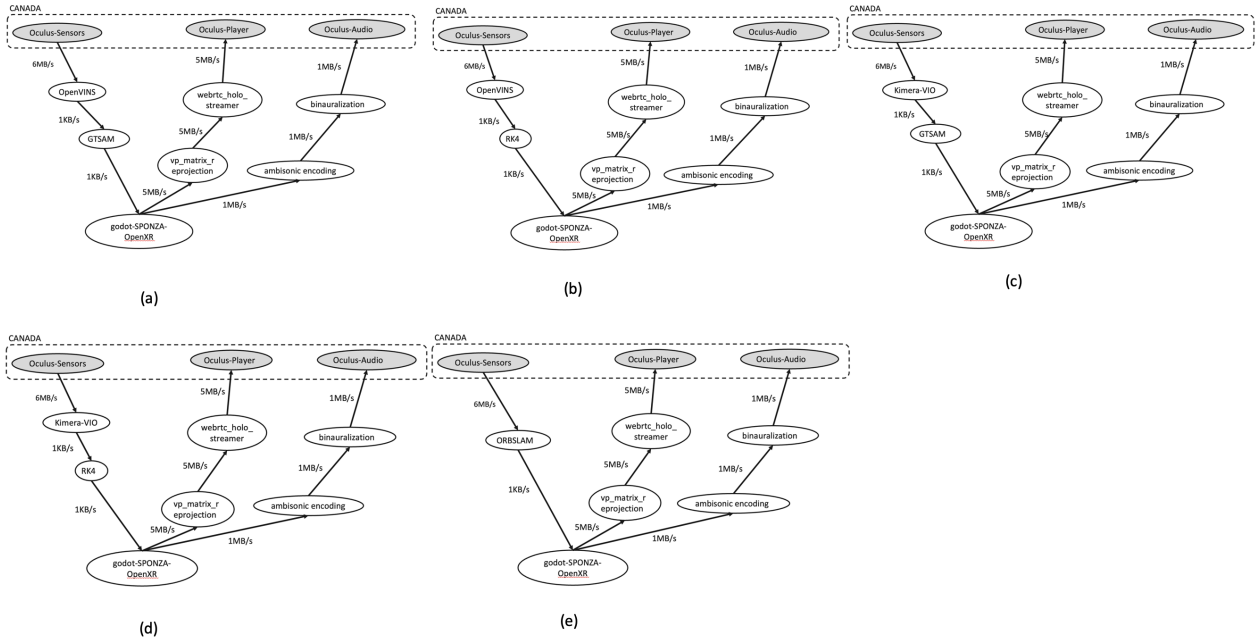



Figure 4.20: 5 Service Graphs of the HXR SPONZA Game Compatible Graph.

ms, and the Bandwidth is at least 50 MB/s.

The numbers beside the nodes in the decision tree represent the latency and cost of the path to the node. The Latency and cost of mapping the ORBSLAM to DCCanada, DCSweden and DCUSA are as follows:

- **DCCanada:** *latency* = **0.11 ms** (service latency) , *cost* = 1.2 $ (flavor cost) + 0.2 $ (license cost) = **1.4 $**

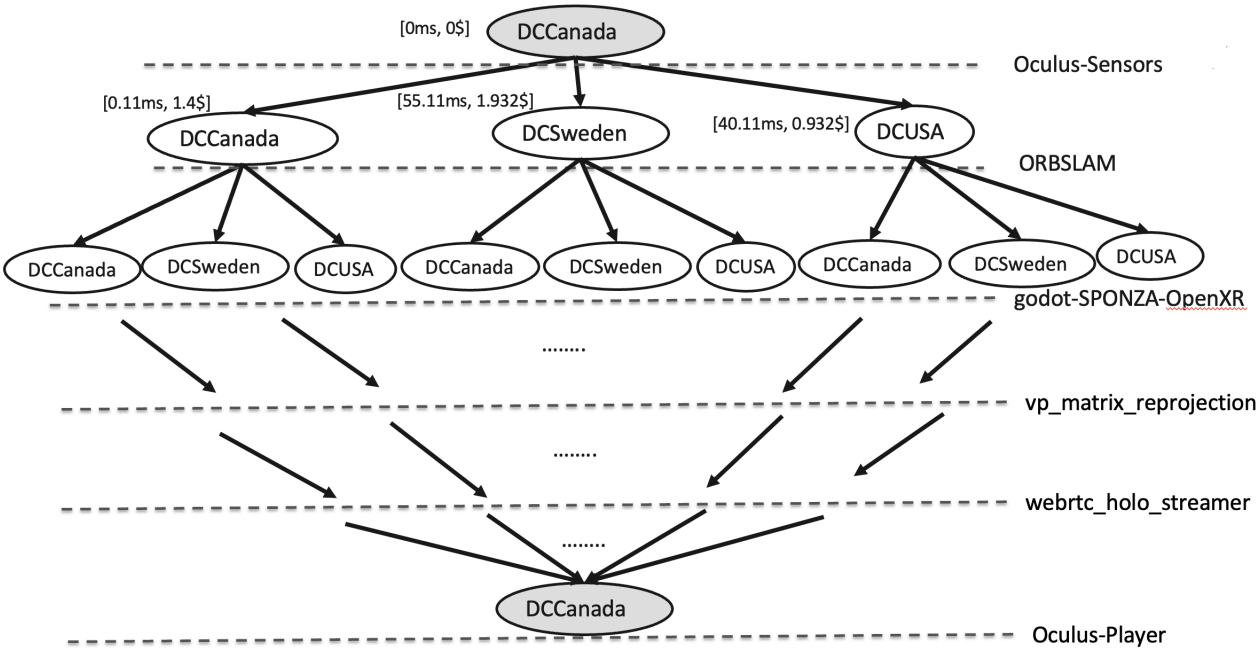



Figure 4.21: Decision Tree for path from ”Oculus-Sensors” to ”Oculus-Player”.

- **DCSweden:** *latency* = 55 ms (link latency) + 0.11 ms (service latency) = **55.11 ms** , *cost* = 1.3 $ (flavor cost) + 0.2 $ (license cost) + (0.006 * 0.02 * 3600) $ (bandwidth cost) = **1.932 $**

- **DCUSA:** *latency* = 40 ms (link latency) + 0.11 ms (service latency) = **40.11 ms** , *cost* = 0.3 $ (flavor cost) + 0.2 $ (license cost) + (0.006 * 0.02 * 3600) $ (bandwidth cost) = **0.932 $**

Note that, for DCCanada, since we still are in the same data center, there is no link latency or bandwidth cost. By continuing this approach for all services the mapping with the minimum cost can be reached. We do the same for the other path and add up the cost of both paths to achieve the minimum cost for this service graph. We already calculated the minimum cost for each service graph in Fig. 4.20. The minimum cost for Fig. 4.20-(a), (b), (c), (d), and (e) are 12.52 $, 12.54 $, 12.52 $, 12.54 $, and **12.17 $**, respectively. Thus, the service graph Fig. 4.20-(e) has the minimum cost and satisfies the requirements. The latency for this service graph is a maximum of **94 ms** which is lower than the requirement which is 120 ms. The data centers for this service graph are shown in Table 4.4:

Note That if the user asks for a medium interaction (latency lower than 80 ms), to meet this requirement, all services should be mapped in DCCanada which leads to a higher cost. In this case, the final service graph is the same as the above-mentioned example, however, the minimum cost is **14.22 $** and latency is a maximum of **14.11 ms**. The data centers for this service graph are shown in Table 4.5:

Table 4.4: Data Centers for Service graph in Fig. 4.20-(e) with latency lower than 120 ms.

| Service Name | Data Center |
|---|---|
| Oculus-Sensors | DCCanada, flavor3 |
| ORBSLAM | DCUSA, flavor2 |
| godot-SPONZA-OpenXR | DCCanada, flavor2 |
| vp_matrix_reprojection | DCCanada, flavor3 |
| webrtc_holo_streamer | DCCanada, flavor3 |
| Oculus-Player | DCCanada, flavor3 |
| ambisonic_encoding | DCUSA, flavor2 |
| binauralization | DCUSA, flavor2 |
| Oculus-Audio | DCCanada, flavor3 |

Table 4.5: Data Centers for Service graph in Fig. 4.20-(e) with latency lower than 80 ms.

| Service Name | Data Center |
|---|---|
| Oculus-Sensors | DCCanada, flavor3 |
| ORBSLAM | DCCanada, flavor3 |
| godot-SPONZA-OpenXR | DCCanada, flavor2 |
| vp_matrix_reprojection | DCCanada, flavor3 |
| webrtc_holo_streamer | DCCanada, flavor3 |
| Oculus-Player | DCCanada, flavor3 |
| ambisonic_encoding | DCCanada, flavor3 |
| binauralization | DCCanada, flavor3 |
| Oculus-Audio | DCCanada, flavor3 |

## 4.4. Conclusion

In this chapter, we presented a detailed explanation of our proposed solution for intent-based service graph generation and selection with minimum deployment cost, demonstrating its effectiveness through the HXR SPONZA Game use case. All the requirements mentioned in Chapter 3 are satisfied. Requirement 1 is addressed by enabling cloud consumers to express their intent using natural language without requiring technical knowledge. To satisfy Requirement 2, the solution employs semantic analysis techniques using domain ontology to translate high-level intent into low-level intent. For example, cloud consumer requests such as ”low interaction” or ”medium quality” are accurately mapped to specific non-functional requirements (NFRs), such as latency and bandwidth constraints. Requirement 3 is fulfilled by automating the service graph generation and selection process. Once the high-level intent is translated into low-level intent, all possible service graphs are generated automatically without human intervention. The solution also meets Requirement 4 by abstracting the complexity of service selection. Cloud consumers are not required to specify individual services; instead, the solution leverages domain ontology and the service catalog to determine the necessary services and their relationships based on the intent. Finally, Requirement 5 is addressed by considering deployment constraints during the service graph selection process. The solution evaluates computational resources, geographic distribution, and network-related constraints, such as latency and bandwidth, to ensure that the selected service graph meets the specified NFRs while optimizing deployment costs. By integrating these capabilities, the proposed solution

successfully satisfies all five requirements.

# Chapter 5

# Evaluation and Result

In this chapter, we evaluate the performance of our proposed service graph generation and selection solutions using the HXR SPONZA game and real data center data from AWS. We begin by describing the evaluation setup and criteria for each solution. Then, we present and analyze the evaluation results across various datasets.

## 5.1. Service Graphs Generation

We implement our service graphs generation solution in a Windows system with 8 core CPUs and 16 GB RAM. Moreover, we use python 3.9.13 and Networkx [45] library to work with graphs. In this section, we present the performance evaluation of our proposed solution. We first present our evaluation criteria followed by the obtained results.

### 5.1.1 Evaluation Criteria

To evaluate our proposed service graphs generation solution, we generate service catalogs in different sizes. Each service catalog contains some use-cases, and each use-case contains services that are needed for its implementation. We generate different use-cases in random sizes. Use-case sizes are generated using an exponential distribution with mean $\frac{1}{\lambda}$. Exponential distributions are memoryless models that can be used to model our uses-case sizes as they are independent of one another. For example, in [46] exponential distributions are used to model the number of incoming service requests. Moreover, we define content diversity for service catalogs. Content diversity is the ratio of unique interfaces in a service catalog to its size. For example, in a service catalog with a size of 100, 50% content diversity means that all 100 services consume/produce 50 unique interfaces that connect these services together.

We evaluate our work on service catalogs of size 100 with content diversity from 20% to 90%. The mean use-case size is set from 5 to 25. We compared our proposed CGGM algorithm with a recent work [9], where service

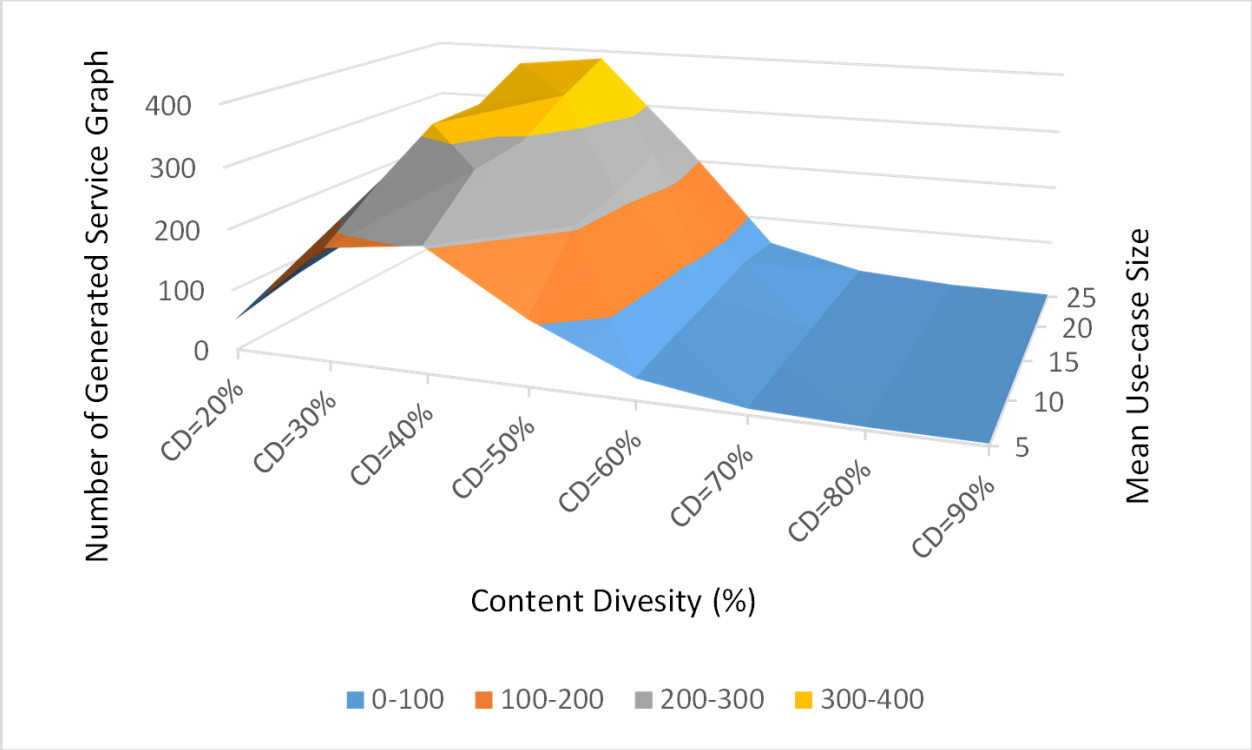



Figure 5.1: Number of generated service graphs for a service catalog of size 100.

function chains (SFC) are constructed and then combined to generate service function graphs. Given that our work considers service graphs, it is different from the existing literature, which mostly consider service function chains. Nevertheless, the method presented in [9] is the most relevant work to our work as it generates a service function graph that is similar to the generated compatible graph in CGGM. The inputs of the SFGC algorithm are the dependency constraint graph (containing dependency of VNFs) and SFC requests containing required VNFs for each request. SFC is similar to a service graph with one source and destination in our work. Besides, since there is no approach in the literature to compare the whole process of service graph generation, we implemented a naive algorithm, which, instead of generating compatible graphs by starting from source services, generates them based on finding the compatibility between all services in the service catalog and repeating it for every incoming use-case. To be more specific, it finds the compatibility between all services in the service catalog, removes redundant services starting from source towards destination services, and generates the compatible graph. After generating the compatible graph, the remaining steps are similar to SGGM. Finally, we evaluate the execution time of our solution by considering a situation where, instead of having one source and destination service in each use-case, we may have multiple sources and/or destinations, similar to our motivating scenario.

### 5.1.2 Evaluation Results

As described in Chapter 4, the generated service graphs using our proposed solution should meet the cloud consumer intent in terms of its FRs. Figure 5.1 presents the number of generated service graphs for a service catalog of size 100 considering content diversity of 20% to 90% and mean use-case size from 5 to 25. In Figure 5.1, we observe that by increasing content diversity from 20% to 40%, the number of generated service graphs increments, and there is a peak at 40%. We also found out that the number of generated service graphs gradually decreases at 40% of content diversity. When we generate use-cases in a service catalog, we deduct the number of source and destination services from use-case size because their location in service graphs is constant. For the remaining services, we need to distribute them between source and destination services, based on the number of existing contents. In other words, we have $n$ services and wants to equally distribute them between $m$ locations (number of contents). Thus, since $n$ is constant and $m$ is variable, the number of generated service graphs increases until 40% of content diversity. At this point it starts to decrease because when $m$ increases cause a smaller number of services located in each location which leads to generating a smaller number of service graphs. For example, for generating a service catalog of the size of 10 and content diversity of 30%, there are 10 services and 3 different contents. Also, we have one source and one destination. Thus, the number of generated graphs will be $1 \times 4 \times 4 \times 1 = 16$. For content diversity of 40%, we have $1 \times 3 \times 3 \times 2 \times 1 = 18$. And for content diversity of 50%, we have $1 \times 2 \times 2 \times 2 \times 2 \times 1 = 16$. If we continue this for different service catalog sizes, we can see that there is always a peak at 40%. In fact, in service catalogs that contain services with high input/output contents diversity, we have less variety in generated service graphs.

Figure 5.2 shows a comparison of the average execution time (in ms) of each use-case in the compatible graph generation module of our proposed CGGM algorithm and the SFGC algorithm [9] in service catalogs with a size of 200 to 1000 and a content diversity of 20%, 40%, 60%, 80%, and a mean use-case size of 15. We observe from Figure 5.2 that the execution time of our proposed CGGM algorithm is smaller than the SFGC algorithm for any given content diversity and service catalog size. More specifically, our proposed CGGM algorithm achieves upto 67% improvement of execution time compared the SFGC algorithm. The reason behind this result is that our CGGM algorithm generates compatible graphs directly by finding compatible services starting from source to destination, whereas the SFGC algorithm first generates SFCs and then combines them to generate the service function graph. Thus, our proposed CGGM algorithm visits each service once for generating compatible graphs, whereas the SFGC algorithm uses a topological sorting for ordering the VNFs in SFCs, thus visiting each VNF more than once. By adding the service function graph generation step, the overall execution time of the SFGC algorithm eventually becomes larger than the proposed CGGM algorithm.

Given that the SFGC algorithm does not include the generation of service graphs, we were not able to make a comparison between it and our proposed SGGM algorithm. Thus, we present a comparison of the logarithmic average

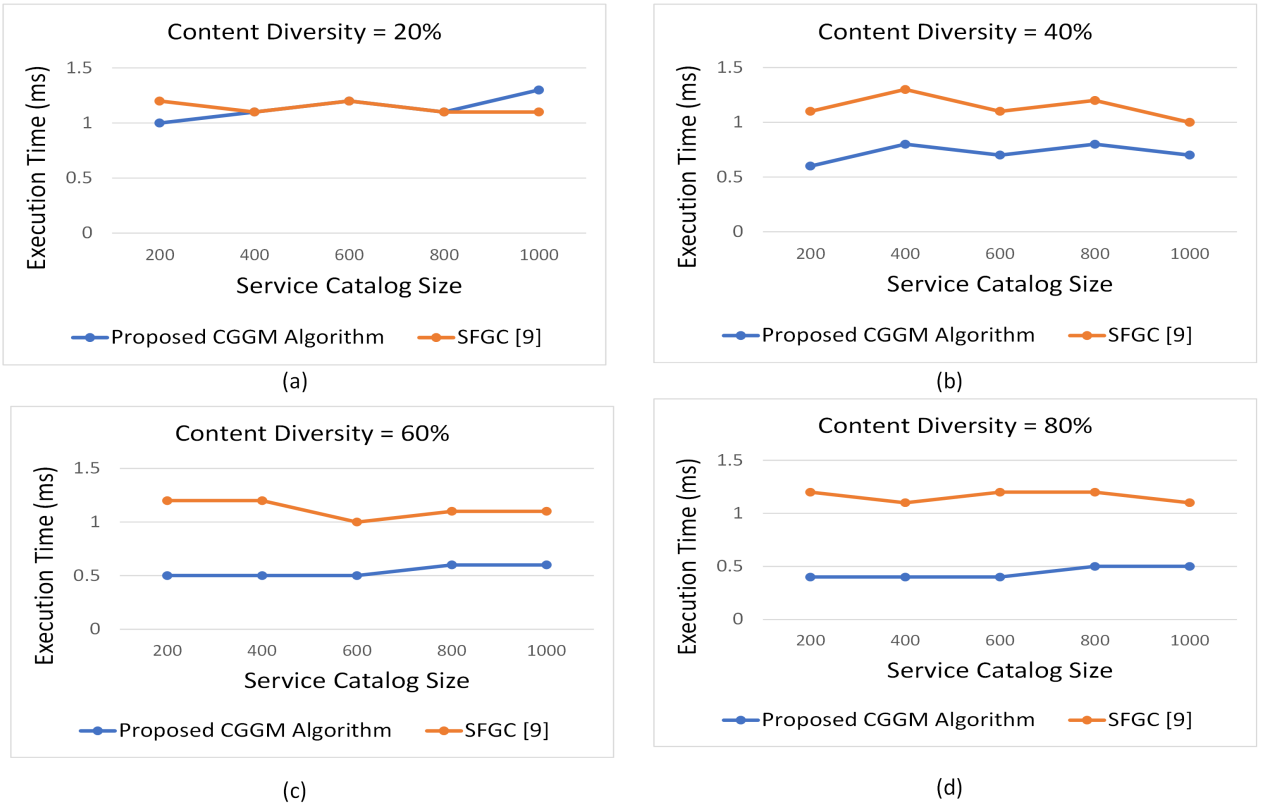



Figure 5.2: Comparison of the average execution time (ms) of each use-case of our proposed CGGM algorithm with the SFGC algorithm [9].

execution time (ms) of each use-case in our proposed algorithm (i.e., both CGGM and SGGM) and a naive algorithm (described in subsection 5.1.1). We observe from Figure 5.3 that by increasing the service catalog size, the execution time for the naive algorithm increases exponentially. Thus, our proposed algorithm generates service graphs with a smaller execution time compared to the naive algorithm. More specifically, we observe that our proposed algorithm improves the execution time by at least 84%. In Figure 5.3-(d), the execution time of our proposed algorithm remains constant for any given service catalog size. This happens because the mean use-case size in all service catalog sizes is 15. On the other side, because of content diversity of 80%, the compatible graph has fewer links between services. Thus, there are fewer paths between source and destination, which leads to the same execution times for different service catalog sizes.

Table 5.1 illustrates the average execution time and the number of service graphs of each use-case with a different number of sources and destinations in a service catalog with a size of 100 and content diversity of 80%, and mean use-case size of 10. As shown in Table 5.1, if there are 1 source and less than 7 destinations, our proposed solution can generate service graphs in less than 1 second, while for more than 7 destinations, it takes more than 1 second. When the number of source and destination services exceeds 3, it takes over a second to find all possible service graphs. The reason is that by expanding sources and destinations, finding all possible routes from sources to destinations may take

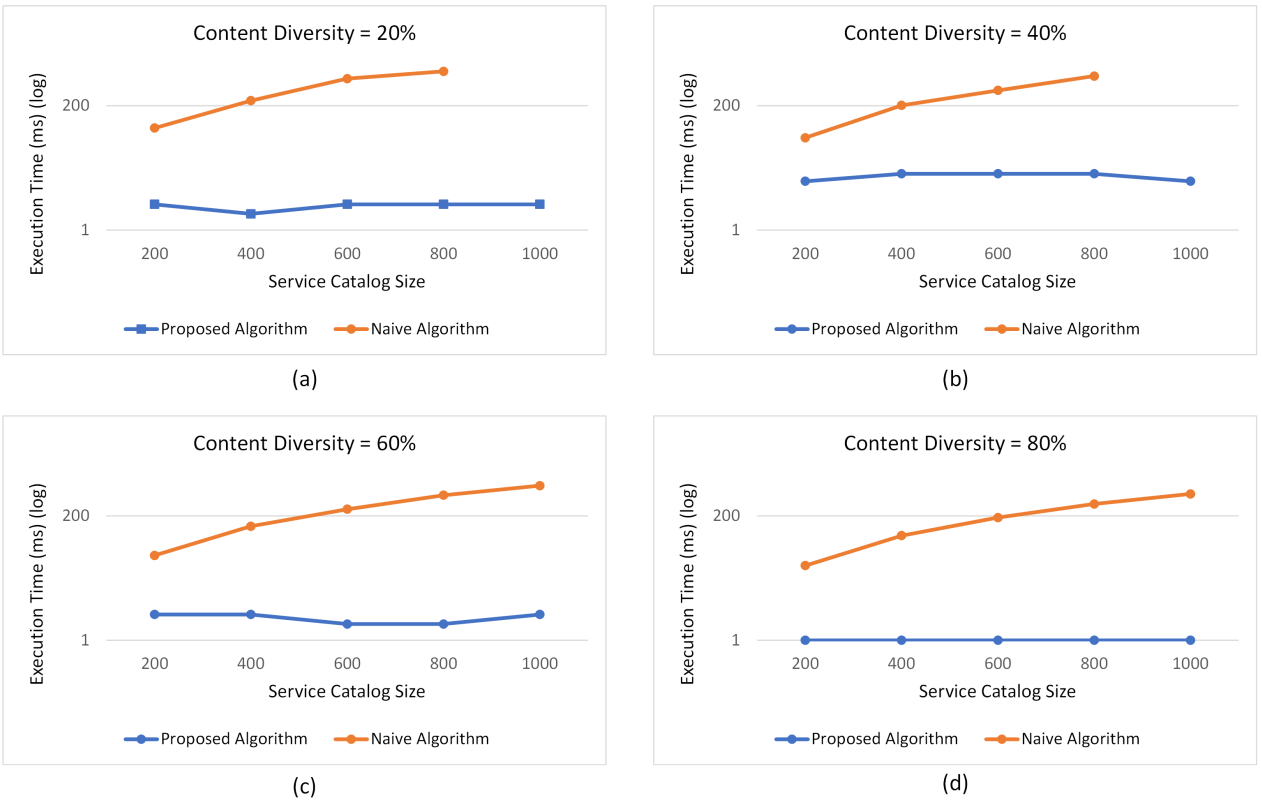



Figure 5.3: Comparison of the average execution time (ms) of each use-case of our proposed algorithm with a naive algorithm.

Table 5.1: Average execution time and number of service graphs for different use-cases.

| Number of Sources | Number of Destinations | Execution Time (ms) | Number of Service Graphs |
|---|---|---|---|
| 1 | $< 7$ | $< 400$ | $< 750$ |
| 1 | $>= 7$ | $> 1s$ | $> 1000$ |
| 2 | $< 4$ | $< 400$ | $< 50$ |
| 2 | $>= 4$ | $> 1s$ | $> 100$ |
| $>= 3$ | $>= 3$ | $> 1s$ | $> 50$ |

longer time. Furthermore, the number of service graphs increases by expanding the number of source and destination services.

## 5.2. Service Graph Selection with Minimum Deployment Cost

In this section we present our evaluation setup followed by the obtained results for service graph selection with the minimum deployment cost solution.

### 5.2.1 Evaluation Setup

We implement our proposed service graph selection solution in an environment (MacBook laptop) with 10 core CPUs, 16 core GPUs, and 16 GB RAM. We use python 3.10.9 and Networkx 2.8 [45] library to work with graphs. To evaluate our proposed solution, we use the Holographic Extended Reality (HXR) SPONZA game as our case study. The services and relationships within the game are derived from ILLIXR [47]. In this use-case, a user wants to play a holographic game by using a head mounted virtual reality device (e.g., Oculus). A typical example of a cloud consumer intent can be as follows: "I would like to have a **Holographic Extended Reality SPONZA Game** using Oculus device at *Ericsson Canada* with *low interaction*, and *medium quality* aiming to have *the minimum deployment cost*." The process of generating compatible graph using FRs captured from intent is discussed in detail in Chapter 4.2. Figure 4.2 illustrates the domain ontology used to translate NFRs from high-level intent into low-level intent suitable as input for our solution. For example, *low interaction* is mapped to a latency requirement of smaller than 120 ms, while *medium quality* corresponds to a bandwidth requirement greater than 50 MB/s. Figure 4.19 depicts the compatible graph of HXR SPONZA game use-case, which is based on a real scenario. The bandwidth values between services in the figure are encoded. However, in our evaluation, since our deployment cost is based on hours, we assume that the data will be transferred over one hour. For example, bandwidth between 'godot-SPONZA-OpenXR' and 'vp-matrix-reprojection' is considered as 5GB/hour, and the bandwidth between 'Oculus-Sensors' and 'ORBSLAM' is treated as 600MB/hour. From the intent, the interaction and quality are translated to latency and bandwidth, respectively. Moreover, for each NFRs, there are three levels of quality, i.e., low, medium, and high, whose translation based on each use-case can be different.

For the data center network, we use real data from Amazon Elastic Compute Cloud (EC2) on-demand pricing[1], which provides scalable computing capacity on demand within the Amazon Web Services (AWS) Cloud. We use data from five regions: USA, Germany, Sweden, Canada, and Japan, with each region potentially hosting multiple data centers. Based on this data, we generate the datasets and, for simplicity, assume that each data center in these regions has 3 flavors. For instance, a total data center size of 50 implies that the data centers are equally split across 5 regions, meaning that each region has 10 data centers, and each data center contains 3 flavors. For each data center size, we generate 50 instances, and flavors for each data center are randomly selected from the EC2 on-demand pricing pool. The bandwidth cost for data transfer into EC2 regions from the internet is $0, while the cost for data transfer out of all regions is $0.02 per GB, except for Japan, which is $0.09 per GB. We set the bandwidth capacity between data centers randomly between 5 and 15 GB, based on the use-case requirements. This range allows for variability in bandwidth, ensuring that while many links will meet the application's needs, some may not, introducing realistic constraints into the model. For latency of transferring data between data centers, we used information from the *cloudping* website[2].

[1] https://aws.amazon.com/ec2/pricing/on-demand/
[2] https://www.cloudping.co/grid

We evaluate our work on the number of data centers ranging from 25 to 100, with each data center offering 3 flavors of different resource capacities and distributed equally across 5 regions. Parameter $N_D$, which determines the number of candidate data centers for each service, is set to approximately 10% of the data center size. We will discuss the impact of $N_D$ on cost and execution time in the next subsection. We compare the deployment cost achieved by our proposed solution with the lower-bound cost. The lower-bound cost is the deployment cost of the service graph when there are no limitations on bandwidth and latency requirements, and the bandwidth price between data centers is zero, making it the minimum deployment cost achievable. By comparing our results to this theoretical minimum cost, we can assess how close our solution gets to the optimal solution. Finally, we evaluate the execution time of our solution by considering changes in data center size and $N_D$.

### 5.2.2 Evaluation Results

Figure 5.4 presents a comparison of the average deployment cost (per hour) for the HXR use-case of our proposed solution vs. the lower-bound cost across different data center sizes (ranging from 25 to 100) with varying latency requirements (from 100 ms to 400 ms). We observe from Figure 5.4 that when the latency requirement increases, the deployment cost of our proposed solution approaches the lower-bound cost for a given data center size. This is because increasing the latency requirement expands the solution space, allowing the algorithm to identify data centers with lower costs for hosting services. More specifically, our proposed solution achieves a deployment cost with a maximum gap of 4% compared to the lower-bound cost, while satisfying user NFRs and deployment constraints. Additionally, the success rate, indicated at the top of each column, represents the percentage of service graphs for which the solution successfully identified a suitable data center network while satisfying NFRs. This success rate reaches 100% as latency requirements increase. Furthermore, as the number of data centers increases, the average cost per hour decreases, which can be attributed to the larger pool of possible data centers capable of hosting services based on their requirements. It is important to note that not only we achieve a minimum deployment cost, but our solution also ensures that all requirements are met and deployment constraints are rigorously satisfied.

Figure 5.5 shows a comparison of the average execution time (seconds) in logarithmic scale for the HXR use-case of our proposed solution in different data center sizes with varying latency requirements. We observe from Figure 5.5 that as the latency requirement increases, the execution time increases for any data center size. The reason behind this is that large latency requirements increase the solution space by providing more possible host data centers to be checked for lower deployment costs. Additionally, the increasing data center size from 25 to 100, increases the execution time exponentially, because the solution checks all possible combinations of data centers to minimize the deployment cost, consequently, as the number of data centers increases, the number of combinations grows exponentially.

Next, we evaluate the impact of bandwidth and latency requirements on deployment cost. Figure 5.6 shows the cost

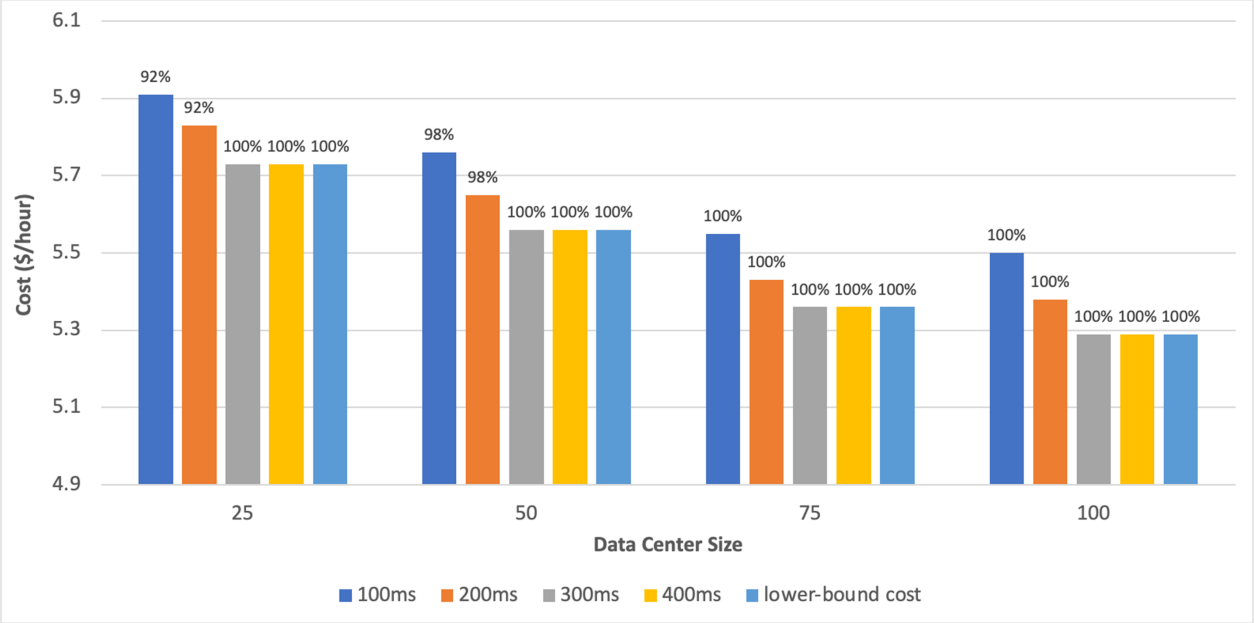



Figure 5.4: Comparison of average cost/hour of changing data center size from 25 to 100 and latency requirement from 100 ms to 400 ms with the lower-bound cost. The percentage at the top of each column indicates the success rate.

per hour for the HXR use-case of our proposed solution as bandwidth requirements vary from 0 to 5 GB/s, considering scenarios with and without latency constraints in a data center size of 25. From Figure 5.6, we observe that as the bandwidth requirement increases, the deployment cost per hour decreases. When there is zero bandwidth, services must reside in the same data center because there is no capacity to transmit data between data centers. Consequently, once the first service is assigned to a data center, all subsequent services must follow, potentially being assigned to more expensive flavors, thus increasing the deployment cost. For bandwidths between 1 and 5 GB/hour, the cost decreases because our HXR-compatible graph (e.g., Fig. 4.19) includes links within this range, then services can be assigned to different data centers, thus reducing overall costs. Furthermore, we notice that for bandwidths of 1 and 4 GB/hour, the costs remain the same regardless of the latency constraints. This occurs because for small bandwidth requirements (e.g., 1 MB/s bandwidth requirement between "OpenVNS" and "GTSAM" ), the cost of bandwidth between data centers is high, making it more economical to keep all services in the same data center. However, for a bandwidth of 5 GB/hour, we observe a cost reduction when there is no latency limitation. In this case, the cost of transmitting data (e.g., the 5 GB/hour bandwidth requirement between "godot-SPONZA-OpenXR" and "vp-matrix-reprojection" ) is smaller than the cost of keeping all services in the same data center.

Figures 5.7 and 5.8 illustrate the impact of the number $N_D$ of candidate data centers in our solution on the deployment cost and execution time. The goal is to determine the impact on the solution's result if an exhaustive search approach was applied. As shown in Figure 5.7, increasing $N_D$ to 25 reduces the deployment cost, with the minimum cost achieved when $N_D$ equals the data center size. This reduction occurs because a larger $N_D$ provides a broader

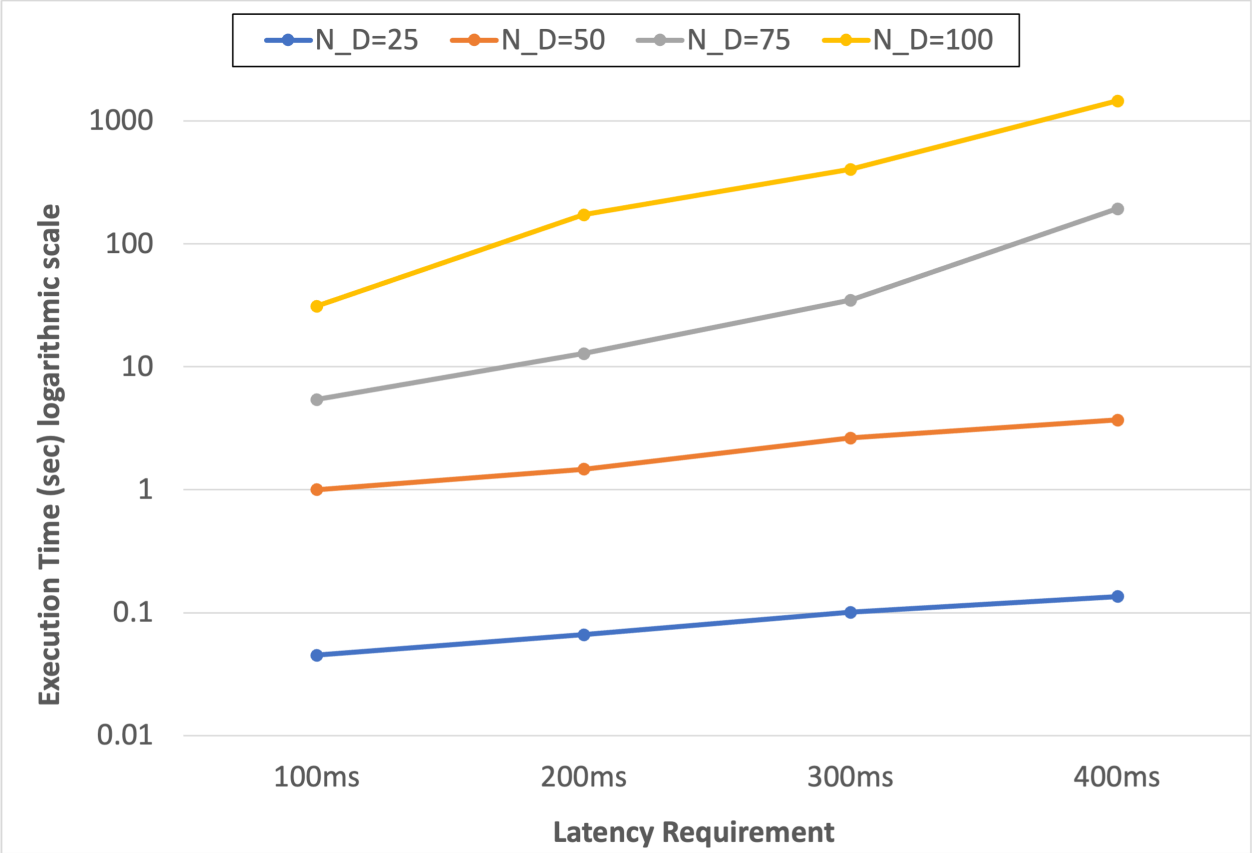



Figure 5.5: Execution time (in logarithmic scale) of our proposed algorithm vs. latency requirement (in ms) for different values of $N_D$.

solution space, enabling the identification of lower-cost data centers. However, increasing the $N_D$ also affects execution time. Figure 5.8 demonstrates the execution time of the algorithm with different $N_D$ parameters in a data center size of 25. As the $N_D$ increases, the execution time grows exponentially. This is because the algorithm must search through all possible data center flavors and their combinations to find the minimum deployment cost. To balance cost and execution time, we keep the $N_D$ to a low value, approximately 10% of the data center size.

### 5.2.3 Discussion

Our results indicate that the proposed solution effectively select a service graph with minimum deployment cost while respecting to cloud consumer intent requirements. Specifically, we achieve a deployment cost with a maximum gap of 4% compared to the lower-bound cost, while ensuring that NFRs and deployment constraints are satisfied. This is crucial as it demonstrates that our solution does not compromise on quality to achieve cost savings. Moreover, a critical observation is the impact of the $N_D$ parameter on deployment cost and execution time. Increasing the $N_D$ expands the solution space, allowing for the identification of lower-cost data centers, however, it also exponentially increases the execution time. By examining the impact of an exhaustive search, we find that maintaining the $N_D$ at

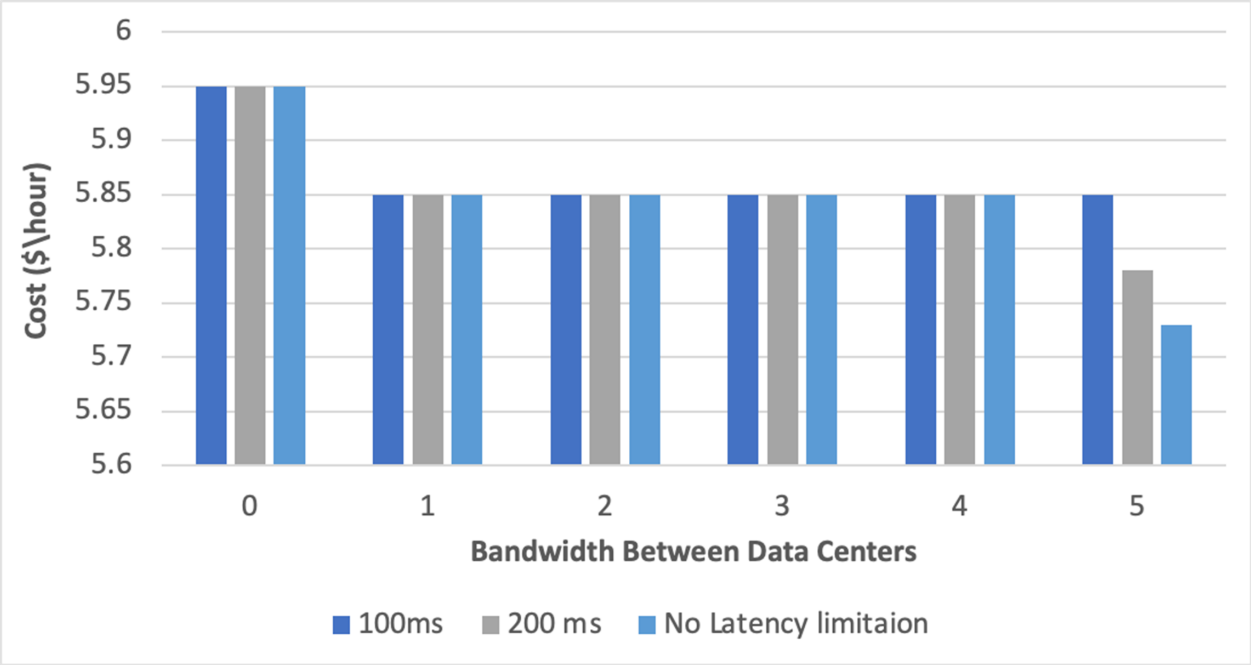



Figure 5.6: Cost/hour of our proposed solution vs. bandwidth requirement (in GB/hour) for different latency requirements of 100 ms, 200 ms, and no latency limitation ($N_D$=25).

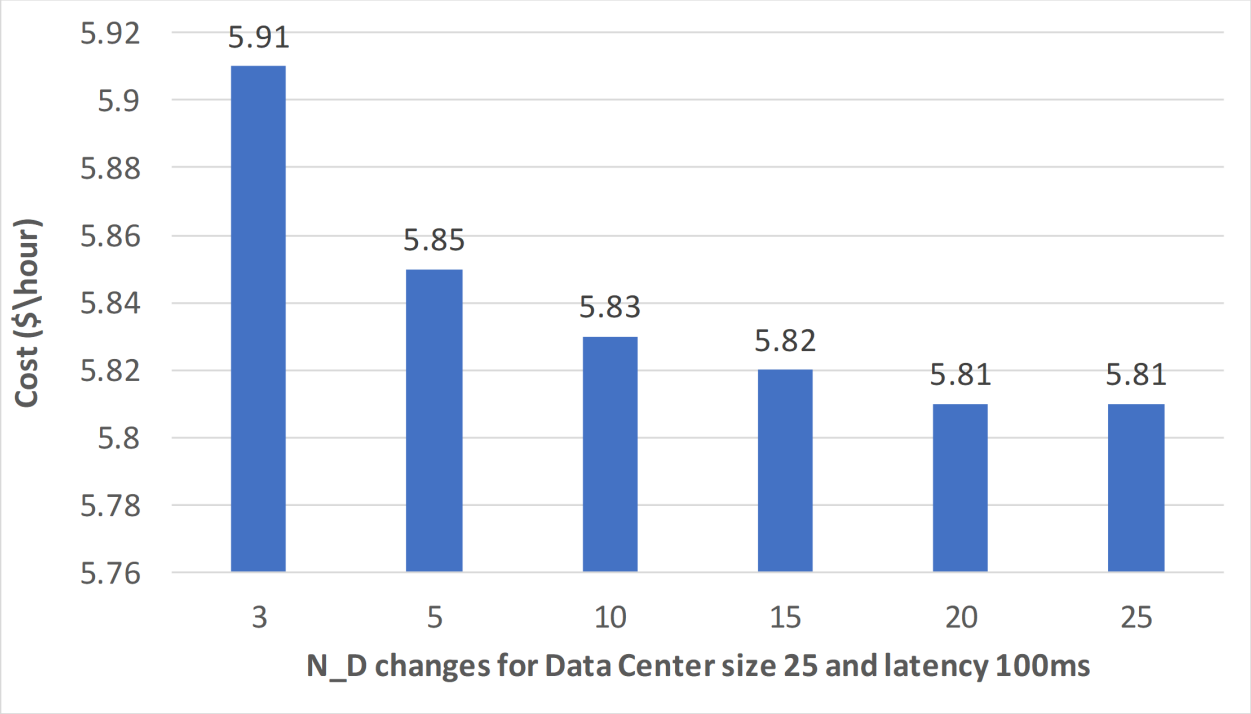



Figure 5.7: Cost/hour vs. $N_D$ ranging from 3 to 25 in data center size 25.

10% of the total data center size provides a balanced trade-off between computational effort and the optimality of the solution. Our solution has been tested with a service catalog containing up to 100 data centers. We note that as the

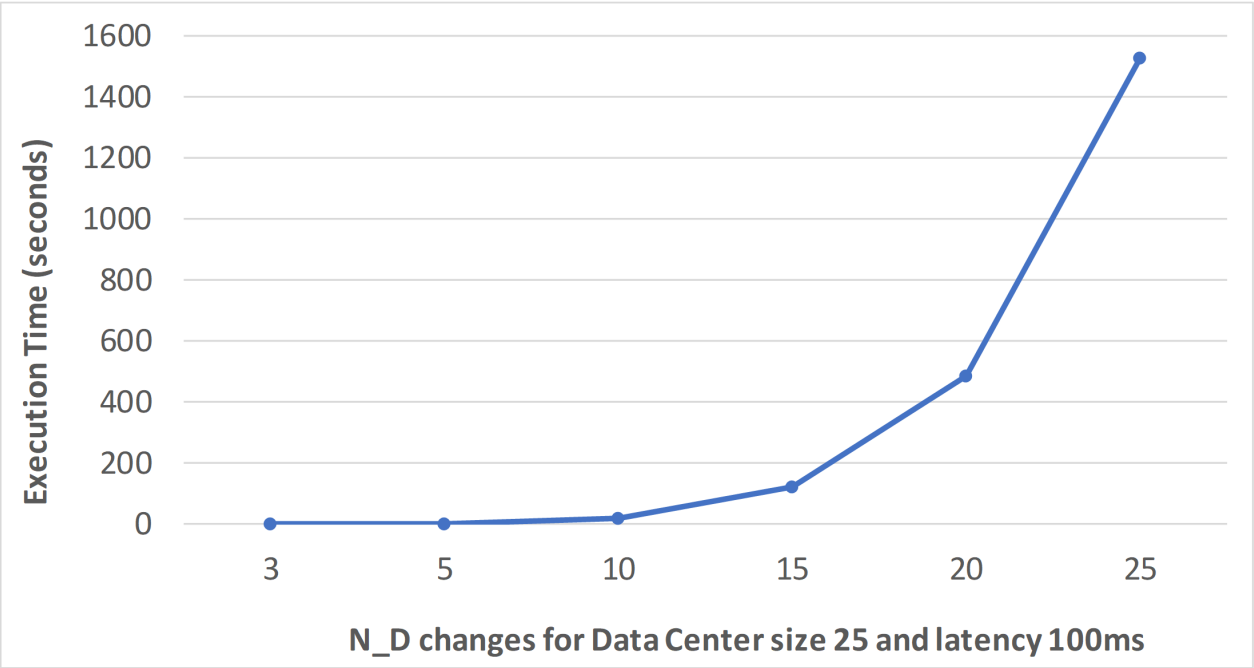



Figure 5.8: Execution time changes of our proposed solution for changing $N_D$ from 3 to 25 in data center size 25.

size of the data centers increases, the execution time is expected to grow exponentially due to the need to evaluate all possible combinations of data centers to minimize deployment costs. Additionally, our implementation showed that when the bandwidth cost between data centers is reduced, the solution tends to assign services to data centers with lower prices, even if staying within the same data center would have no bandwidth cost. This is because the solution prioritizes minimizing the overall deployment cost, and may choose to distribute services across different data centers if it leads to a lower total cost.

# Chapter 6

# Conclusions and Future Works

## 6.1. Conclusions

Cloud computing has transformed the way services are provisioned by enabling scalable and on-demand access to computing, storage, and networking resources. It allows cloud consumers to deploy services based on specific requirements of their applications. These applications are composed of independent microservices, which communicate with each other to deliver a specific functionality. However, selecting and composing services to generate a service graph that meet both functional and non-functional requirements of cloud consumers requirements remains a complex challenge, particularly in dynamic cloud environments. This thesis tackled the service graph generation and selection problem by introducing an intent-based solution that allows cloud consumers to specify their requirements in natural language.We began by identifying key challenges in service graph generation and selection through a motivating scenario in holographic communication, where a cloud consumer with limited network/cloud knowledge, specifies high-level intents. To enable automated service generation and selection, we established five essential requirements: (1) natural language support, (2) semantic analysis, (3) automated service graph generation and selection, (4) no prior knowledge of required services, and (5) minimizing service deployment cost while considering infrastructure deployment constraints. Our analysis of related work revealed that while some approaches address certain aspects-such as natural language support or deployment constraints- none provide a comprehensive solution that satisfies all five requirements. To bridge this gap, we proposed a solution that extracts functional requirements (FRs) and non-functional requirements (NFRs) from high-level intent, generates all possible service graphs, and selects a service graph with minimum deployment costs while considering infrastructure deployment constraints. We extract FRs and NFRs from high-level intent using a domain ontology. Based on the extracted FRs, all possible service graphs that satisfy the intent are generated. Among these, one service graph is selected. The selection problem is formulated as an Integer Linear Programming (ILP) model to ensure that the chosen service graph minimizes deployment cost while satisfying

the NFRs and infrastructure deployment constraints.Our solution eliminates the need for manual human intervention, making service graph generation and selection automated. Through performance evaluations, we demonstrated that our approach significantly improves execution time and minimize deployment cost compared to existing solutions. Our service graphs generation algorithm achieves up to 67% reduction in execution time compared to the SFGC algorithm, proving its efficiency in handling large-scale service catalogs. Furthermore, our solution achieves a deployment cost within 4% of the lower-bound optimal cost, ensuring that cost savings do not compromise service quality or intent requirements. Additionally, our results highlight the trade-off between solution space exploration and computational effort, where limiting the number of candidate data centers to 10% of the total size, balances execution time and cost deployment. We also observed that bandwidth cost significantly influences service placement, as the algorithm dynamically assigns services to different data centers to achieve the lowest overall cost. By automating the transition from high-level intent to cost-effective service deployment, our work contributes to more flexible and user-friendly cloud service management. This thesis provides a foundation for future advancements in intent-based cloud management, simplifying service graph generation and selection for cloud consumers while ensuring cost-effective service deployments in the cloud.

## 6.2. Future Works

Future research can focus on validating the proposed solution using real-world cloud service deployments and cloud consumers intents to assess its practical applicability and performance. Additionally, incorporating AI and machine learning techniques to help translating intent, predict future user requirements and optimize the service graph selection process proactively. Beyond optimizing the service graph itself, future work could explore customizing individual functions within the graph to better align with specific application needs. Furthermore, the current approach does not explicitly address some functionalities of Intent-Based Networking (IBN), such as verification and assurance, which are essential for a fully automated and reliable intent-driven system. Finally, while cost minimization has been a primary focus, future research should also consider other critical factors like energy consumption and fault tolerance, which are becoming increasingly important in sustainable and resilient cloud deployments.

# Chapter 7

# Publications

- Sabour S, Ebrahimzadeh A, Soualhia M, Wuhib F, Glitho RH. Intent- based Service Graph Selection for Cost-Effective Cloud Deployment. In 2025 IEEE 28th Conference on Innovation in Clouds, Internet and Networks (ICIN).
- Sabour S, Ebrahimzadeh A, Wuhib F, Soualhia M, Glitho RH. Service Graphs Generation in Intent-Based Networks. In IEEE 21st Consumer Communications Networking Conference (CCNC) 2024 Jan 6 (pp. 90-97).
- W O2024201109A1, Method for generating service graphs from high-level intents, Sasan Sabour, Fetahi Wuhib, Amin Ebrahimzadeh, Roch Glitho, publication date: 03 October 2024.

# Bibliography


[1] A. Leivadeas and M. Falkner, "A survey on intent-based networking," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 625–655, 2022.

[2] Y. Ouyang, C. Yang, Y. Song, X. Mi, and M. Guizani, "A brief survey and implementation on refinement for intent-driven networking," *IEEE Network*, vol. 35, no. 6, pp. 75–83, 2021.

[3] C. Li *et al.*, "Intent classification," *IETF, Fremont, CA, USA, Mar. 2021*.

[4] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, J. Yin, S. Dustdar, and A. Y. Zomaya, "Cloud-native computing: A survey from the perspective of services," *Proceedings of the IEEE*, vol. 112, no. 1, pp. 12–46, 2024.

[5] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," 2008.

[6] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "Above the clouds: A berkeley view of cloud computing," *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, vol. 28, no. 13, p. 2009, 2009.

[7] M. De Donno, K. Tange, and N. Dragoni, "Foundations and evolution of modern computing paradigms: Cloud, iot, edge, and fog," *IEEE Access*, vol. 7, pp. 150936–150948, 2019.

[8] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet computing*, vol. 13, no. 5, pp. 14–22, 2009.

[9] Y. Wang, L. Zhang, P. Yu, K. Chen, X. Qiu, L. Meng, M. Kadoch, and M. Cheriet, "Reliability-oriented and resource-efficient service function chain construction and backup," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 240–257, 2021.

[10] P. Horn, "Autonomic computing: Ibm's perspective on the state of information technology," 2001.

[11] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[12] M. H. Behringer, M. Pritikin, S. Bjarnason, A. Clemm, B. E. Carpenter, S. Jiang, and L. Ciavaglia, "Autonomic Networking: Definitions and Design Goals," 2015.

[13] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura, "Intent-based networking - concepts and definitions." https://www.rfc-editor.org/info/rfc9315, 2022.

[14] A. Clemm, L. Ciavaglia, L. Granville, and J. Tantsura, "Intent-based networking-concepts and overview," *Internet Engineering Task Force, Internet-Draft*, 2019.

[15] J. Li, A. Sun, J. Han, and C. Li, "A survey on deep learning for named entity recognition," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 50–70, 2022.

[16] M. Kiran *et al.*, "Enabling intent to configure scientific networks for high performance demands," *Elsevier Future Generation Computer Systems*, vol. 79, pp. 205–214, 2018.

[17] A. S. Jacobs *et al.*, "Deploying natural language intents with lumi," in *Proc. ACM SIGCOMM Conference Posters and Demos*, pp. 82–84, 2019.

[18] M. Jain *et al.*, "Intent-based, voice-assisted, self-healing sdn framework," *Journal of Network Communications and Emerging Technologies (JNCET)*, vol. 10, no. 2, 2020.

[19] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville, "Refining network intents for self-driving networks," in *Proc. the Afternoon Workshop on Self-Driving Networks*, pp. 15–21, 2018.

[20] P. Mell, "The nist definition of cloud computing," *Recommendations of the National Institute of Standards and Technology*, 2011.

[21] W. Voorsluys, J. Broberg, and R. Buyya, "Introduction to cloud computing," *Cloud computing: Principles and paradigms*, pp. 1–41, 2011.

[22] "What is aws?." https://aws.amazon.com/what-is-aws/. Accessed: July 22, 2025.

[23] "What is amazon ec2?." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. Accessed: July 22, 2025.

[24] "Regions and zones." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html. Accessed: July 22, 2025.

[25] J. Gil Herrera and J. F. Botero, "Resource allocation in nfv: A comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[26] G. ETSI, "Network functions virtualization (nfv); architectural framework," *ETsI Gs NFV*, vol. 2, p. V1, 2014.

[27] "What are microservices?." https://aws.amazon.com/microservices/. Accessed: July 22, 2025.

[28] N. Gritli, F. Khendek, and M. Toeroe, "Decomposition and propagation of intents for network slice design," in *Proc. IEEE 5G World Forum (5GWF)*, pp. 165–170, 2021.

[29] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing?," *International journal of human-computer studies*, vol. 43, no. 5-6, pp. 907–928, 1995.

[30] N. F. Noy, D. L. McGuinness, *et al.*, "Ontology development 101: A guide to creating your first ontology," 2001.

[31] "What is service catalog?." https://docs.aws.amazon.com/servicecatalog/latest/adminguide/introduction.html. Accessed: July 22, 2025.

[32] A. Clemm, M. T. Vega, H. K. Ravuri, T. Wauters, and F. De Turck, "Toward truly immersive holographic-type communication: Challenges and solutions," *IEEE Communications Magazine*, vol. 58, no. 1, pp. 93–99, 2020.

[33] M. Kiran, E. Pouyoul, A. Mercian, B. Tierney, C. Guok, and I. Monga, "Enabling intent to configure scientific networks for high performance demands," *Elsevier Future Generation Computer Systems*, vol. 79, pp. 205–214, 2018.

[34] C. Kim, Y. Oh, and J. Lee, "Latency-based graph selection manager for end-to-end network service on heterogeneous infrastructures," in *Proc. International Conference on Information Networking (ICOIN)*, pp. 534–539, 2018.

[35] D. C. Verma, "Service level agreements on ip networks," *Proceedings of the IEEE*, vol. 92, no. 9, pp. 1382–1388, 2004.

[36] A. S. da Silva, H. Ma, and M. Zhang, "A graph-based particle swarm optimisation approach to qos-aware web service composition and selection," in *Proc. IEEE Congress on Evolutionary Computation (CEC)*, pp. 3127–3134, 2014.

[37] A. Sailer, M. R. Head, A. Kochut, and H. Shaikh, "Graph-based cloud service placement," in *Proc. IEEE International Conference on Services Computing*, pp. 89–96, 2010.

[38] J. Pei, P. Hong, K. Xue, D. Li, D. S. Wei, and F. Wu, "Two-phase virtual network function selection and chaining algorithm based on deep learning in SDN/NFV-enabled networks," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 6, pp. 1102–1117, 2020.

[39] N. Nazarzadeoghaz, F. Khendek, and M. Toeroe, "Automated design of network services from network service requirements," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pp. 63–70, IEEE, 2020.

[40] W. Chao and S. Horiuchi, "Intent-based cloud service management," in *Proc. IEEE Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pp. 1–5, 2018.

[41] E. J. Scheid *et al.*, "INSpIRE: Integrated NFV-based intent refinement environment," in *Proc. IFIP/IEEE Symposium on Integrated Network and Service Management*, pp. 186–194, 2017.

[42] A. Leivadeas and M. Falkner, "Vnf placement problem: A multi-tenant intent-based networking approach," in *Proc. 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pp. 143–150, 2021.

[43] T. Di Riccio, J. Massa, S. Forti, and A. Brogi, "Sustainable placement of vnf chains in intent-based networking," in *Proc. IEEE/ACM 16th International Conference on Utility and Cloud Computing*, pp. 1–10, 2023.

[44] M. Avgeris, A. Leivadeas, N. Athanasopoulos, I. Lambadaris, and M. Falkner, "Model predictive control for automated network assurance in intent-based networking enabled service function chains," in *Peoc. IEEE/IFIP Network Operations and Management Symposium*, pp. 1–7, 2023.

[45] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[46] R. Mohamed, A. Leivadeas, I. Lambadaris, T. Morris, and P. Djukic, "Online and scalable virtual network functions chain placement for emerging 5G networks," in *Proc. IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, pp. 255–260, 2022.

[47] M. Huzaifa, R. Desai, S. Grayson, X. Jiang, Y. Jing, J. Lee, F. Lu, Y. Pang, J. Ravichandran, F. Sinclair, *et al.*, "Illixr: Enabling end-to-end extended reality research," in *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 24–38, 2021.