INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Uma Extensão ao Kubernetes para Cargas de Trabalho de Telepresença em XR

SIMÃO PEDRO RIBEIRO DOS SANTOS
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Extending Kubernetes for XR Telepresence Workloads

Simão Santos

A dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science, Specialisation Area of Software Engineering

Advisor: Dr. Nuno Pereira

Porto, June 29, 2025

Statement of Integrity

I hereby declare that I have conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore, the work presented in this document is original and authored by me, having not previously been used for any other end. The exceptions are explicitly recognised in the section "Ethical considerations" of the first chapter. This section also states how AI tools were used and for what purpose.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, June 29, 2025

Dedicatory

Dedicated to the loving memory of my grandmother, Maria Arminda Leite Ribeiro. I carry her spirit forward in this work and in all that I do. This is for you, Avó...

Abstract

While conventional 2D telepresence platforms like Zoom and Microsoft Teams have become commonplace, they lack key real-world interaction cues. To bridge this gap, Extended Reality (XR) telepresence has emerged, using immersive wearables for more natural collaboration. However, the intensive Graphics Processing Unit (GPU) processing required by XR often exceeds the limited compute power and battery life of client devices, leading to poor frame rates. Therefore, the common architectural solution is to offload these demanding workloads to the network edge to satisfy strict low-latency requirements.

Kubernetes, the standard platform for orchestrating and deploying containerized applications, was originally designed for cloud-centric stateless architectures with load-based scaling. This model conflicts with the demands of XR telepresence systems, which require session-aware scaling for user-shared sessions and stable network identities for persistent, stateful workloads. This need for dynamic orchestration is further complicated by low-latency Quality of Experience (QoE) requirements that force computation to the network edge. While existing research addresses enabling Kubernetes at the edge, these studies are often generic, crucially lacking the session-awareness required for telepresence.

This dissertation's primary objective is to bridge the gap between Kubernetes and the demands of XR telepresence. The goal is to design a system that abstracts infrastructure complexity and automates session lifecycle management. This will allow an application's runtime to dynamically schedule, scale, and place compute resources to optimize for user latency and session demands.

This dissertation follows a design and creation research strategy, structured in key phases: design, implementation, and evaluation. The initial phase was informed by a state of the art review, which was conducted with a systematic methodology. This review was driven by two primary research questions: the first investigated the architectural design of XR telepresence systems, while the second explored how these applications could be deployed to Kubernetes.

The contributions of this work are twofold. First, a state of the art review of 28 studies identified a significant gap in current research: while tools exist for generic multi-cluster management, they lack the specific session-awareness required for XR telepresence workloads. Second, to fill this gap, the main contribution is a framework that extends Kubernetes using the Operator Pattern, introducing custom controllers that make the platform session-aware. The evaluation of this framework demonstrated significant performance improvements when compared against baseline implementations. With the proposed solution, nearly 100% of clients connect to a session in under 15 seconds, whereas clients in baseline cases often take several minutes. Furthermore, the framework showcases inferior memory utilization under dynamic client loads, reducing consumption by up to 50% due to its garbage collection mechanisms.

Keywords: Telepresence, Extended Reality, Kubernetes, Edge Computing

Resumo

A notória diferença entre a comunicação no mundo real e a que se experiencia em plataformas de telepresença convencionais, como o Zoom e o Microsoft Teams, impulsiona a transição para uma era de interações mais imersivas. Estas tiram partido das interfaces de realidade estendida *(XR)*, suportadas por dispositivos como o Microsoft HoloLens e o Meta Quest, que visam uma colaboração mais próxima do natural. Este formato de comunicação permite que aspetos como o olhar, a linguagem corporal e o áudio espacial sejam replicados no ambiente de interação, superando assim as limitações dos formatos tradicionais, onde a atenção dos utilizadores se cinge a uma grelha de vídeos. A telepresença em *XR* pode, contudo, adotar múltiplas configurações, que a literatura posiciona ao longo do chamado contínuo da realidade-virtualidade. Estas modalidades vão desde Ambientes Virtuais Colaborativos *(CVEs)* totalmente imersivos, onde os utilizadores interagem através de avatares num mundo gerado por computador, até sistemas de Realidade Mista *(MR)* assimétricos, que fundem os espaços físico e virtual.

O Kubernetes, a plataforma de referência para a implantação de aplicações que tiram partido de conteinerização, foi inicialmente desenvolvido para arquiteturas *stateless* orientadas para a *cloud*, nas quais a provisão de recursos responde ao aumento da carga. Por outro lado, os sistemas de telepresença requerem um modelo de orquestração distinto. No seu núcleo reside o conceito de sessão, que define um espaço de interação para um grupo de utilizadores. Deste modo, a gestão dos recursos da infraestrutura deve ser realizada em função da atividade das sessões, necessitando que a plataforma gira dinamicamente os Pods à medida que os utilizadores se juntam ou abandonam uma sessão. Adicionalmente, estas cargas de trabalho, cujo estado é relevante (*stateful*), exigem endereços de rede fixos para garantir a persistência das ligações, um requisito que entra em conflito com o funcionamento padrão do Kubernetes. A necessidade de uma orquestração dinâmica e orientada às sessões torna-se ainda mais complexa pela infraestrutura exigida para cumprir os requisitos de baixa latência, que impõe que a computação seja colocada na *edge*. Embora a investigação existente aborde o desafio de estender o Kubernetes para a *edge*, estes estudos são, na sua maioria, genéricos. De forma crucial, não consideram o contexto específico da telepresença, onde a noção de sessão tem de ser integrada diretamente no plano da infraestrutura.

O objetivo primordial desta dissertação é colmatar a lacuna identificada entre as capacidades do Kubernetes e os requisitos específicos das cargas de trabalho de telepresença em *XR*. A meta consiste na conceção e implementação de um conjunto de mecanismos capazes de abstrair a complexidade inerente aos serviços e à infraestrutura. Este trabalho visa, assim, criar um sistema que promova a gestão automatizada da infraestrutura necessária. Tal sistema permitirá que o tempo de execução de uma aplicação possa, de forma dinâmica, alocar recursos computacionais, ajustá-los em função da atividade da sessão e distribuí-los por localizações que otimizem a latência para o utilizador.

A presente dissertação segue uma metodologia de investigação composta pelas fases de conceção, implementação e avaliação. A fase inicial de conceção foi fundamentada por uma análise do estado da arte, conduzida através de uma metodologia sistemática. Esta

análise foi orientada por duas questões de investigação primordiais: a primeira incidiu sobre a arquitetura dos sistemas de telepresença em *XR*, enquanto a segunda explorou de que forma é que estas aplicações podem ser implantadas no Kubernetes.

Este trabalho apresenta duas contribuições principais. Em primeiro lugar, uma análise ao estado da arte que abrangeu 28 estudos identificou uma lacuna na investigação atual: embora existam ferramentas para a gestão genérica de múltiplos clusters, estas carecem de suporte para aplicações com cargas de trabalho centradas no conceito de sessão.

Para colmatar esta lacuna, o principal contributo consiste num *framework* que estende o Kubernetes através do *Operator Pattern*, introduzindo controladores especializados no domínio da telepresença que permitem à plataforma sensibilidade às sessões. A avaliação deste framework revelou melhorias de desempenho significativas quando comparado com implementações de referência. Com a solução proposta, perto de 100% dos clientes acedem a uma sessão em menos de 15 segundos, enquanto que nos cenários de referência este processo chega a demorar vários minutos. Adicionalmente, o framework apresenta um menor consumo de memória sob cargas dinâmicas de clientes, alcançando uma redução de até 50% devido aos seus mecanismos de *garbage collection*.

Acknowledgement

I would like to express my sincere gratitude to all who have played a pivotal role in my development throughout this academic journey.

First and foremost, I am profoundly grateful to my parents, grandparents, and my entire family for their unconditional support and the opportunities they have provided me. Their collective influence has been fundamental to my growth and success.

I extend my special and heartfelt thanks to my girlfriend, Filipa, for her invaluable comfort and support during this entire process. Her presence was my anchor in the most challenging moments, making the successful completion of this dissertation possible.

I owe an enormous thank you to my advisor, Professor Nuno Pereira, for his expert guidance and incredible dedication. I could always count on his assistance and collaboration. His willingness to schedule calls, answer my messages promptly, and provide advice of the highest quality to my progress is deeply appreciated.

My gratitude also extends to my friends, particularly Rui Neto and Nuno Ribeiro. They were exemplary classmates and companions, whose support during group projects and constant motivation were invaluable.

Contents

List of Figures

List of Tables

List of Acronyms

6DoF	Six Degrees of Freedom.
AKS	Azure Kubernetes Service.
API	Application Programming Interface.
AR	Augmented Reality.
AV	Augmented Virtuality.
AWS	Amazon Web Services.
CAPI	ClusterAPI.
CCDF	Complementary Cumulative Distribution Function.
CDF	Cumulative Distribution Function.
CIDR	Classless Inter-Domain Routing.
CNFs	Container Network Functions.
CNI	Container Networking Interface.
CPU	Central Processing Unit.
CRD	CustomResourceDefinition.
CRUD	Create, Read, Update and Delete.
CVE	Collaborative Virtual Environment.
DT	Digital Twin.
ECS	Entity Component System.
FPS	Frames Per Second.
GC	Garbage Collection.
GPU	Graphics Processing Unit.
HTTP	Hypertext Transfer Protocol.
ICE	Interactive Connectivity Establishment.
JPEG	Joint Photographic Experts Group.
JWT	JSON Web Token.
MLLM	Multimodal Large Language Model.
MMO	Massively Multiplayer Online.
MR	Mixed Reality.
MS	Mapping Study.

NAT	Network Address Translation.
NHPP	Non-Homogeneous Poisson Process.
OKG	Open Kruise Game.
OODA	Observe-Orient-Decide-Act.
QoE	Quality of Experience.
RBAC	Role-based Access Control.
REST	Representational State Transfer.
SDK	Software Development Kit.
SDP	Session Description Protocol.
STUN	Session Traversal Utilities for NAT.
TURN	Traversal Using Relays around NAT.
URL	Uniform Resource Locator.
UUID	Universally Unique Identifier.
VM	Virtual Machine.
VMI	VirtualMachineInstance.
VR	Virtual Reality.
WASM	WebAssembly.
XR	Extended Reality.

Chapter 1

Introduction

Kubernetes workload management primitives, such as Deployment [1], ReplicaSet [2], and StatefulSet [3], do not suit the characteristics of Extended Reality (XR) telepresence workloads. These native resources were primarily designed for traditional, cloud-centric architectures where stateless microservices and stateful databases predominate. In contrast, XR telepresence workloads, ranging from complex 3D graphics rendering to real-time object detection, impose far more complex orchestration behaviour and stricter network requirements than typical web applications. To maintain an acceptable level of user experience, computational throughput must be maximized, which is heavily affected by network latency. Consequently, the consensus in the researched literature is that XR computation should be offloaded to the network edge to satisfy the low-latency requirements of user Quality of Experience (QoE). Although some studies address the deployment of XR workloads on Kubernetes [4], they focus on enabling Kubernetes at the edge. Crucially, immersive telepresence applications involve a notion of a session where several users meet, requiring the management of infrastructure according to the state of each telepresence session. This particular aspect of immersive telepresence is not yet adequately addressed in existing work.

Bridging the identified gap between standard Kubernetes and the demands of XR telepresence requires an in-depth understanding of the platform's internal mechanisms. Based on this understanding, this work proposes a direct extension to Kubernetes itself. This extension provides a high-level framework that encapsulates the complexities of compute scheduling, service discovery, and networking across sliced edge environments into a simple interface designed to be integrated directly into an application's runtime, enabling it to communicate with the infrastructure and schedule compute resources according to session demands. Compared with baseline implementations that use standard Kubernetes primitives, this solution demonstrates a significant performance improvement. With the proposed framework, nearly 100% of clients can connect to a session in under 15 seconds, whereas clients in the baseline cases often take several minutes. This efficiency extends to resource utilization during periods of high load variance, where the framework consumes up to 50% less memory than conventional baseline implementations. This superior performance, however, represents a notable trade-off. The same advanced, session-aware features that enable these efficiencies introduce a minor Central Processing Unit (CPU) and memory overhead during periods of sustained client load.

1.1 Context and Problem

As XR wearables become increasingly affordable, their rate of adoption accelerates. A market analysis by Oberlo [5], illustrated in Figure 1.1, projects that the number of Virtual Reality

(VR) users in the United States alone is estimated to reach 333.5 million by 2028. This growing embrace of XR technologies opens the door for new and more immersive forms of communication. From this growing adoption emerges XR telepresence: a communication model that aims to enhance the sense of co-presence among remote users and offers a significant improvement over traditional 2D video-conferencing.

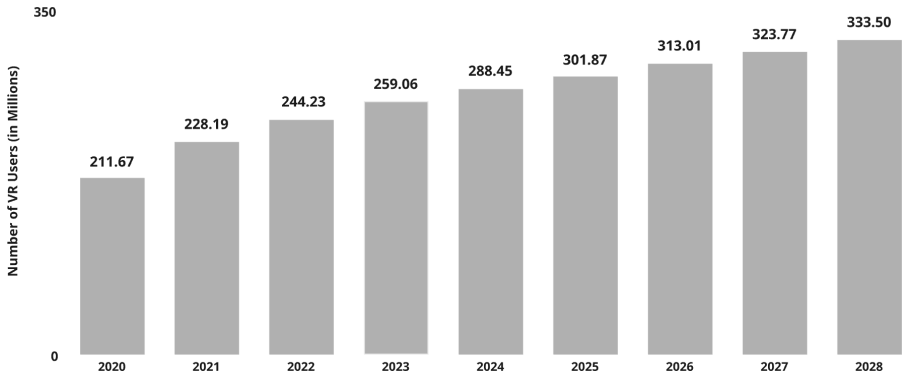


Figure 1.1: Number of VR users in the United States from 2020 to 2028, with projections from 2024 onwards [5].

XR workloads, such as real-time object detection and high-resolution 3D rendering, are computationally intensive and rely on powerful processing resources [6–9]. Attempting to execute these tasks locally on the constrained Graphics Processing Units (GPUs) of client devices is often infeasible, as it exceeds their processing capabilities and leads to rapid battery depletion [7, 8]. In response to these limitations, computational offloading has emerged as a prevailing strategy in the research community. This approach involves transferring demanding XR workloads from client devices to more powerful remote servers. Prior studies suggest leveraging remote compute resources for techniques such as remote rendering [7, 9] and assisted object detection [6, 8, 10]. These methods bridge the gap between the computational needs of XR applications and the limited resources of client devices. While computational offloading effectively solves the problem of limited on-device processing power, it introduces a critical new consideration: network latency. Research shows that XR applications are highly sensitive to latency, requiring end-to-end latencies below 15 ms to provide a smooth user experience and avoid motion sickness [8, 11]. To meet theses low-latency requirements, studies suggest that workloads must be placed in a distributed manner across edge infrastructure, positioning computational power closer to users [6, 8, 10–12]. While this approach achieves latency reduction, it results in a cumbersome infrastructure and escalates the deployment complexity of these applications.

While Kubernetes [4] is widely regarded as the standard platform for orchestrating and deploying containerized distributed applications [12], its capabilities for managing XR services have remained largely unexplored [6]. Typical cloud-native workloads are structured around stateless Representational State Transfer (REST) architectures [13], which differ significantly from the demands of XR telepresence. XR telepresence workloads are characterized by stateful, session-bound remote instances that perform computation for users participating in a specific session. Furthermore, their communication model is not based on discrete requests; instead, it relies on continuous data streams, often using protocols like WebRTC [14]. This stateful, stream-based interaction creates fundamental challenges for standard Kubernetes patterns. First, because each workload instance is non-fungible (i.e., tied to specific users and session), clients require a stable network identity to maintain a persistent

connection. This directly conflicts with the conventional practice of using a Kubernetes Service [15] to load-balance traffic randomly across a set of interchangeable replicas, a model that is unsuitable here.

Second, the scaling behaviour is fundamentally different. Unlike REST services that scale based on generic metrics like CPU or request load, XR telepresence must scale in response to application-level session events, such as users joining or leaving. Standard primitives like the Deployment and StatefulSet resources are ill-suited for this task. This limitation is evident during both scale-up and scale-down operations. When scaling up, these controllers simply increment a replica count; they create a generic Pod [16] and have no mechanism to associate it with the specific client that triggered the scaling event. This forces the client into a complex and inefficient discovery process to find its allocated resource. When scaling down, the problem persists. A Deployment treats its Pods as fungible replicas and terminates them non-deterministically, which could remove a Pod serving an active client. A StatefulSet imposes a strict ordinal scaling policy, which prevents the targeted removal of a specific idle Pod from the middle of the set.

In summary, a fundamental conflict exists between the architectural requirements of XR telepresence and the design of conventional Kubernetes. The need for stateful, session-bound workloads with stable network identities, deployed across a distributed edge infrastructure, is at odds with the platform's native primitives, which were designed for stateless, fungible services. This work, therefore, proposes a framework to extend Kubernetes, enabling the session-aware lifecycle management of XR telepresence workloads.

1.2 Objectives

This work is guided by a central question: *How can Kubernetes support the deployment of XR telepresence workloads?*

The main objective is to enable the deployment of XR telepresence applications by establishing a set of mechanisms that abstract and automate infrastructure management. This integration enables the application runtime to directly manage infrastructure by providing endpoints to query session state and orchestrate services, including their scheduling and placement, based on session demands and user locality. To systematically explore and address this main objective, the following sub-objectives have been developed:

1) **Establish a state of the art on XR telepresence architectures and their deployment on Kubernetes**
   This objective was addressed by conducting a comprehensive state of the art review to build a foundational understanding of the problem space. The review analysed existing XR telepresence architectures, the necessity of computational offloading to the edge, and the inherent limitations of standard Kubernetes for this domain.

2) **Design and implement a framework for deploying XR telepresence on Kubernetes**
   Building upon the insights from the literature review, this objective was fulfilled through the design and implementation of a new framework. The solution is based on extending Kubernetes to manage the stateful, session-based lifecycle of XR workloads.

3) **Evaluate the performance of the proposed solution and extract conclusions**
   The final objective was met by conducting a performance evaluation of the proposed framework. A comparative analysis was performed by benchmarking the solution

against two conventional baseline architectures, using both synthetic and real-world load.

The remainder of this dissertation is structured to address these objectives sequentially. Chapter 2 presents the state of the art review. Chapters 3 and 4 detail the design and implementation of the proposed framework, respectively. Finally, Chapter 5 presents the evaluation and discusses the results.

1.3 Ethical Considerations

Ethical Considerations as a Student of Polytechnic Institute of Porto As a student of Polytechnic Institute of Porto, this work has been developed in accordance with the ethical principles outlined in the institution's Code of Good Practices and Conduct [17]. In line with the declaration of commitment, included at the beginning of this document, this work is entirely original, free from plagiarism, and all sources are properly cited, upholding the required standards of academic integrity. While portions of the state of the art review are adapted from a prior academic report, they have been substantially revised and expanded for this dissertation. AI-powered tools were utilized exclusively for grammatical correction and language refinement; all content presented in this dissertation is the original work of the author.

Ethical Considerations During Research For ethical considerations during research, the *Code of Good Practices and Conduct of P.Porto* [17] served as a guiding framework for ethical judgment. Article 10 of the code outlines best practices for research, emphasizing key principles such as integrating the latest knowledge (principle 1-a), distinguishing prior research (principle 1-e), and ensuring transparency (principle 1-f) in presenting findings. The following considerations outline how this work adheres to these principles:

- This study includes a comprehensive state of the art analysis that incorporates the most recent advancements in the field, ensuring that the findings are grounded in the latest knowledge and reflect the current understanding of the subject matter.
- The research carefully distinguishes the innovative aspects of this study from the existing body of work by appropriately citing related studies.
- Transparency is prioritized in the presentation of research findings, allowing others to evaluate the methodology and build upon the insights generated.

1.4 Document Structure

The present document is organized into the following six chapters:

- Chapter 1, the current chapter, introduces the problem this work aims to address, outlines its main objectives, and discusses the ethical considerations taken during the research and development process.
- Chapter 2 delves into the state of the art, beginning with the research methodology used to conduct the literature review. It then presents findings on the architectural characteristics of XR telepresence applications and explores how they can be deployed and managed on Kubernetes.

- Chapter 3 details the system's design. It starts by outlining the requirements derived from the state of the art review and then explores two potential Kubernetes extension patterns. The chapter concludes by presenting the final proposed system architecture.
- Chapter 4 shifts from design to the concrete implementation of the framework's components. It provides an in-depth look at the framework inner workings.
- Chapter 5 is dedicated to the evaluation of the proposed framework. This chapter details the experimental setup, the generation of load, and the baseline scenarios used for comparison. It presents a comparative analysis of resource utilization and latency, followed by a multi-cluster performance benchmark.
- Chapter 6 provides the conclusion, summarizing the work's contributions in relation to its objectives. It also discusses potential architectural enhancements and outlines directions for future work.

Chapter 2

State of the art

This chapter explores the state of the art, reviewing the latest advancements relevant to the work. It begins by outlining the research methodology and proceeds to present the findings in two main sections: the first focuses on the architectural aspects of XR telepresence applications, while the second explores the deployment of these to Kubernetes cloud-edge environments.

2.1 Research Methodology

To build a comprehensive and evidence-based understanding of the existing body of knowledge, this research implemented a structured literature review. This approach adopted common practices from systematic methodologies to effectively identify and synthesize foundational studies relevant to the research questions.

The methodology for this structured review is guided by the principles of a Systematic Mapping Study (MS). An MS is designed to provide a broad overview of a research area by identifying the quantity and type of available research [18]. Accordingly, the process adopted in this dissertation involved several key steps outlined by Petersen et al. [18]: defining research questions, conducting searches for primary studies, screening papers using inclusion and exclusion criteria, classification, and data extraction. This section details the execution of the first four steps, while the final one is presented in subsequent sections (Section 2.2 and 2.3).

For the present study, two research questions were defined as follows:

- **RQ1** - What are the architectural characteristics of XR telepresence applications?
- **RQ2** - How can XR telepresence applications be deployed on Kubernetes?

While *RQ2* is closely aligned with the overarching goal of this work, it is crucial to first understand the underlying architecture of the targeted applications. *RQ1* was therefore designed to explore the architectural components of XR telepresence systems, providing the foundational knowledge needed to address *RQ2*. By establishing this groundwork, the study ensures a more informed approach to understanding how such applications can or should be managed. Insights gained from the studies reviewed under *RQ1* contributed to the refinement and development of *RQ2*.

Critically, the knowledge required to answer *RQ1* proved to be exceptionally broad, spanning a wide spectrum from studies targeting XR telepresence systems to research more directly focused on XR offloading. A single, rigid search query was found to be impractical for

covering this diverse landscape. This realization necessitated a flexible, exploratory approach for *RQ1*, while the more focused nature of *RQ2* permitted a stricter, systematic process.

For the broad inquiry of *RQ1*, an exploratory review was conducted. This process followed a two-phase approach using the *ACM Digital Library* and *IEEE Xplore*. The first phase, discovery, involved executing a series of queries to identify the core concepts across the different domains. From these results, a foundational body of literature was established through selecting the most relevant studies. The second phase, expansion, then pivoted to snowballing and author-based searches.

In contrast, for *RQ2*, a systematic MS was performed. A single search query was formulated and executed on the same digital libraries. Subsequently, all articles returned by the search were systematically screened to identify primary studies for inclusion. To ensure comprehensiveness, this initial set of studies was then augmented through a snowballing cycle, which involved reviewing the references of included papers to identify relevant research not captured by the database search.

All studies identified through both the exploratory review for *RQ1* and the MS for *RQ2* were subject to the same filtering protocol. This protocol was guided by the inclusion and exclusion criteria defined in Table 2.1.

Inclusion Criteria	**Exclusion Criteria**
(IC1) A study that provides insights into the architectural design of XR telepresence systems and/or aspects related to deploying these to Kubernetes.	(EC1) A study not written in English. (EC2) A study published as an abstract or book chapter. (EC3) A study published before the year of 2020.

Table 2.1: Study inclusion/exclusion criteria.

Studies were analysed using the 3-Pass Approach. In an initial stage, only the first pass was conducted, focusing on the title, abstract, and introduction of each study to evaluate their relevance. Studies deemed irrelevant were excluded, thereby avoiding the need for a detailed review of every paper. This approach significantly reduced the pool of studies, leaving only those regarded as relevant for a more comprehensive review.

Regarding database searches, the presence of search keywords in the abstract was set as a key inclusion rule. Studies lacking these keywords in their abstracts were excluded, as they were unlikely to address the research questions and would further be eliminated during the first-pass analysis. This 'fail-fast' approach allowed for the early elimination of irrelevant studies, ensuring that only the most pertinent papers were retained for further review.

2.1.1 RQ1: Selected Studies

Following the exploratory review process detailed in the preceding section, a total of 19 studies were selected as relevant for answering *RQ1*. These studies form the evidence base for understanding the architectural characteristics of XR telepresence applications. Table 2.2 provides a comprehensive list of these included studies. Each entry is presented with its *Publication Year*, its *Class* (XR telepresence, XR offloading, or both), the *Library* from which it was sourced, and the *Method* used for its discovery (query in library, author-based search, or snowballing).

To better comprehend the overall XR telepresence environment, the studies were classified into two groups representing different architectural perspectives. The first class, "XR telepresence", consists of papers focused on describing the overall XR telepresence scenario, often presenting the architecture from a high-level perspective with less technical granularity. The second, "XR offloading", addresses the more specific technical challenge of computational offloading, a common requirement for these systems. The selected literature shows a division between these classes, with a larger portion of studies focusing specifically on XR offloading (9 studies) and a substantial number addressing XR telepresence (7 studies). Notably, only three studies were identified that explicitly integrate both concepts.

Reference	Year	Class	Library	Method
[19]	2025	XR telepresence	ACM DL	Query
[20]	2024	XR telepresence and XR offloading	ACM DL	Query
[21]	2024	XR offloading	IEEE Xplore	Author
[22]	2024	XR telepresence	ACM DL	Query
[23]	2024	XR telepresence	ACM DL	Query
[7]	2024	XR telepresence and XR offloading	ACM DL	Query
[24]	2023	XR telepresence	ACM DL	Query
[9]	2023	XR offloading	IEEE Xplore	Author
[25]	2023	XR offloading	ACM DL	Query
[26]	2023	XR telepresence	ACM DL	Snowball
[27]	2023	XR telepresence	IEEE Xplore	Author
[8]	2023	XR offloading	IEEE Xplore	Query
[28]	2023	XR telepresence and XR offloading	IEEE Xplore	Query
[6]	2023	XR offloading	ACM DL	Query
[10]	2023	XR offloading	ACM DL	Query
[11]	2022	XR offloading	IEEE Xplore	Snowball
[29]	2022	XR offloading	IEEE Xplore	Snowball
[30]	2022	XR offloading	IEEE Xplore	Snowball
[13]	2021	XR telepresence	IEEE Xplore	Snowball

Table 2.2: List of all included studies for answering RQ1.

2.1.2 RQ2: Selected Studies

The search query for RQ2 was developed through an iterative refinement process, culminating in the final string shown in Figure 2.1. Preliminary searches that directly incorporated keywords from the research question, such as "telepresence" or "XR", proved ineffective, yielding a very limited number of relevant results. To bridge this gap, insights from the architectural analysis in *RQ1* were leveraged. It became apparent that the challenges of deploying XR applications often manifest as issues related to multi-cluster setups. This term was therefore adopted as a more effective proxy to identify relevant studies. The final query was constructed around the "multi-cluster" keyword to guarantee the retrieval of papers addressing the core technical challenges pertinent to *RQ2*.

The study selection process for the *RQ2* MS, visually summarized in Figure 2.2, followed a structured funnelling approach. The initial database search yielded 37 candidate studies. After applying the inclusion and exclusion criteria from Table 2.1 to filter for aspects like publication type and publication year, 23 studies remained. These were then subjected to a

kubernetes
AND
(multi-cluster **OR** cross-cluster **OR** inter-cluster **OR** federated-cluster)
AND
(management **OR** manage **OR** orchestration **OR** orchestrate **OR** coordination **OR** coordinate)

Figure 2.1: Search string for RQ2.

manual screening of their titles, abstracts, and introductions, which identified 7 studies as highly relevant. To ensure complete coverage, a snowballing cycle was performed on this core set, which contributed 2 additional primary studies. This resulted in a final selection of 9 reviewed papers for *RQ2*.

In addition to the systematic review of research literature, the analysis for *RQ2* was complemented with grey literature. This was especially crucial for understanding the practical implementations and architectural patterns of two foundational open-source projects: Agones [31] and Open Kruise Game (OKG) [32].

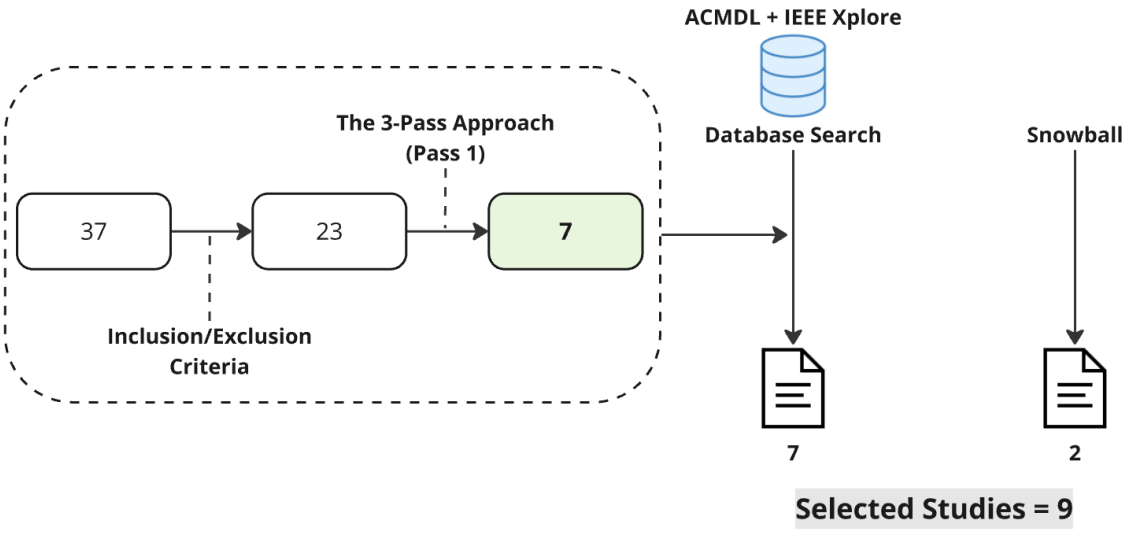


Figure 2.2: Study selection process for RQ2.

The reviewed papers selected for *RQ2* are catalogued in Table 2.3. While this table follows the same structure as that for *RQ1*, a different classification scheme was adopted. The studies are classified into two categories. The first, "XR workloads", includes papers that specifically discuss the deployment of XR applications on Kubernetes. The second and more general category, "Edge workloads", encompasses research on the Kubernetes edge computing paradigm, which was identified as a foundational topic for deploying XR systems. A key insight from this classification is that all selected studies address the challenges of edge workloads. Of these, only two papers focus on XR workloads explicitly.

Reference	Year	Class	Library	Method
[12]	2024	XR workloads and Edge workloads	IEEE Xplore	Query
[33]	2024	Edge workloads	ACM DL	Query
[34]	2024	Edge workloads	IEEE Xplore	Query
[35]	2024	Edge workloads	IEEE Xplore	Query
[36]	2024	Edge workloads	IEEE Xplore	Query
[37]	2023	XR workloads and Edge workloads	IEEE Xplore	Snowball
[38]	2023	Edge workloads	IEEE Xplore	Query
[39]	2023	Edge workloads	IEEE Xplore	Query
[40]	2021	Edge workloads	ACM DL	Snowball

Table 2.3: List of all included studies for answering RQ2.

2.2 Mixed-Reality Telepresence

The desire to communicate remotely without travelling or commuting has propelled the development of telepresence systems. Such systems have become an integral part of daily activities, achieving significant usage in both professional and casual sectors [19, 24]. On desktops and mobile platforms, 2D telepresence applications such as Zoom[1], Facetime[2], or Microsoft Teams[3] have proven to be a successful form of communication among distant users. However, researchers argue that the lack of gaze [19, 24, 27], body orientation [19, 24, 27], and spatial audio [19, 27] inherent in these systems limits communication compared to real-world interactions. Because users typically view all participants on a grid of videos, it is difficult to discern who is addressing them, as gaze and head direction are aimed at a screen rather than at individual users. For example, environments such as remote education requiring hands-on training are much limited in 2D video-conferencing as physical demonstrations are difficult to capture [26]. Furthermore, side conversations are extremely rare since audio lacks spatial meaning [19], further distancing this form of communication from face-to-face conversations.

As XR interfaces (e.g., Microsoft HoloLens [41], Meta Quest [42]) demonstrate great promise for improving the sense of co-presence [26] and become increasingly affordable [24], new modalities for remote human collaboration have emerged. Capitalizing on this, 3D telepresence has become a longstanding area of research [23], explored in many forms, spanning the full Milgram's virtuality-reality continuum [43]. The conceptual framework for XR—an umbrella term for immersive technologies—was introduced by Paul Milgram in 1994 through the Reality-Virtuality continuum, which categorizes technologies based on their balance of real and virtual elements [30]. Positioned at one end of this continuum is VR, which involves minimal real-world content and immerses users entirely within a computer-rendered world. On the opposite end, Augmented Reality (AR) maintains the physical environment as the user's main perspective while layering computer-generated elements onto their real-world view. Augmented Virtuality (AV) takes an approach inverse to that of AR, embedding real-world elements within primarily virtual environments. Together, AR and AV form the subcategory of Mixed Reality (MR), spanning the spectrum between AR and AV, merging real and virtual elements.

[1]Zoom: `https://www.zoom.com/en` (Accessed 28-06-2025)

[2]Apple Facetime: `https://support.apple.com/en-us/105088` (Accessed 28-06-2025)

[3]Microsfot Teams: `https://www.microsoft.com/en-us/microsoft-teams/free?market=pt` (Accessed 28-06-2025)

Various efforts have been made to reduce the communication gap between 2D telepresence systems and real-world interactions [19, 27]. A key focus has been on supporting side conversations, with prominent examples including Zoom's breakout rooms[4] and the 2D spatial audio environments of SpatialChat[5]. However, researchers argue that these approaches still struggle to preserve natural aspects of conversation, such as maintaining eye contact and conveying who is facing whom [19, 27].

In contrast to the flattened arrangement of user attention across a grid of videos, XR telepresence offers an enhanced approach to remote communication and collaboration. It preserves spatial meaning in conversations and makes body orientation, attention, and body language continuously observable. Some popular forms of XR telepresence unfold as Collaborative Virtual Environments (CVEs), including platforms like Meta Horizon[6] and VRChat[7]. Several other works, also leveraging avatar representations (e.g., video-cubes, Meta Avatars), bring remote users into VR [13, 20, 22, 24, 27] or AR environments [19]. Other collaborative configurations involve co-located users interacting with virtual objects in an AR scene [13, 28]. Alternative approaches use 3D reconstructions or 360° video streaming to immerse remote users into the physical spaces of co-located users through asymmetric AR/VR collaboration [13, 23, 25]. Furthermore, some methods, particularly in hands-on remote education, transmit hand gestures and verbal communication through a human surrogate [26].

Previous studies have demonstrated a wide range of benefits from these immersive settings across several domains. These include lab training [25], gaming [13, 28], education [23, 26], social meetings [13, 19, 24, 27], sports training [22], and robotic training through Digital Twins (DTs) [30], among others.

Figure 2.3 illustrates some of the aforementioned approaches, showcasing different ways in which remote users can be represented and interact within shared digital or physical spaces. The following sections detail several key works that exemplify these diverse XR telepresence scenarios.

(a) ArenaXR [27] (b) MiniMates [19] (c) VirtualNexus [23]

Figure 2.3: Extended reality telepresence examples: In (a), users are represented as video-cube avatars while interacting in a VR scene. In (b), the AR user sees others represented as miniature avatars, which are projected onto his real-world perspective. Combining both AR and VR, (c) demonstrates MR telepresence, allowing a VR user, projected as an avatar, to experience the same physical location as an AR user.

[4]Zoom breakout rooms: `https://support.zoom.com/hc/en/article?id=zm_kb&sysparm_article=KB0061583` (Accessed 28-06-2025)

[5]SpatialChat: `https://spatial.chat/` (Accessed 28-06-2025)

[6]Meta Horizon: `https://horizon.meta.com/` (Accessed 28-06-2025)

[7]VRChat: `https://hello.vrchat.com/` (Accessed 28-06-2025)

ArenaXR Pereira et al. [13] introduced ArenaXR, a platform conceived to simplify the development and hosting of collaborative XR applications on web browsers. Leveraging the adoption and support of XR standards (WebXR [44]) in the web browser space, this approach aims to accelerate the construction of cross-platform XR applications, thereby making them accessible through various devices such as XR headsets, desktops and other browser-capable devices.

ArenaXR enables XR collaboration through scenes. These scenes abstract groups of users and virtual content (3D objects), accommodating remote VR users in virtual worlds and/or co-located AR users in physical locations. Scenes are loaded similarly to web applications; however, it is possible to simultaneously view a composition of multiple scenes without switching between tabs. For users interacting in a scene, synchronization of graphical updates, sounds, and I/O events across all participants is ensured. This is all enabled by a fundamental architectural component: a message broker that disseminates information between data producers (publishers) and consumers (subscribers). For instance, when a user moves their camera or modifies an object's color, these actions are transmitted as messages via the broker, enabling real-time updates and maintaining consistency in the shared environment. Additionally, ArenaXR's architecture includes a persistence data store to provide users with the latest state of 3D objects upon connection. Authentication is performed using OAuth [45] and user permissions are managed via JSON Web Tokens (JWTs) [46]. It also provides a web server for delivering static content such as 3D models and sounds.

As a development framework, ArenaXR provides a distributed WebAssembly (WASM) [47] runtime environment capable of hosting developers' applications across various distributed compute elements. These elements include browsers or headless devices (e.g., sensors), with the latter operating a stand-alone WASM runtime. A Python library [48] facilitates the development of these applications, enforcing the Entity Component System (ECS) architectural pattern. Interfacing with the message broker, this library allows developers to embed scene logic (such as manipulating 3D objects or reacting to event callbacks) directly into programs that can dynamically start and exit in a hot-pluggable manner anywhere on the network.

As part of the use cases, ArenaXR was leveraged to build a MR telepresence application, by integrating the video conferencing stack Jitsi [49] into the platform. This enabled VR and AR users to experience the same physical location, by volumetrically capturing the physical local and then uploading the resulting 3D model into an arena scene. With this setup, AR users joining the scene from the physical location were able to view and interact with remote VR users, while experiencing the same place, and represented as video-cube avatars (see Figure 2.3a). Later, Dasari et al. [27] built a VR chat system on top of ArenaXR with users also presented as video-cube avatars. His work aimed at the challenges of scaling VR telepresence for thousands of users in the same virtual world, highlighting the differences to scaling traditional 2D video content delivery.

MiniMates Kiuchi et al. [19] introduces MiniMates (Figure 2.3b), a new approach designed to address the challenges of scaling AR videoconferencing and effectively accommodating all remote users within each participant's physical space. This method facilitates remote collaboration through the use of miniature 3D avatars in AR, thereby overcoming issues stemming from the heterogeneity and limitations of physical space layouts. This prevents common problems like avatar occlusion or interference with furniture and walls that often occur when using life-sized avatars. Furthermore, by overlaying these miniature avatars onto

the user's real-world environment, MiniMates preserves situational awareness and prevents isolation from the surroundings. Beyond spatial challenges, MiniMates also addresses key limitations of traditional 2D videoconferencing tools, such as the lack of gaze, head direction, and the ability for parallel conversations.

ViGather Qiu et al. [24] contributes to maintaining a consistent level of perceived presence between MR users and those using traditional screen devices. To achieve this, the authors propose ViGather, a platform-agnostic telepresence system that normalizes the experience for hybrid users through avatars in a first-person VR scene. Because the user representation (avatar) is platform-independent, participants joining from either a flat screen or an XR device cannot perceive the platform of other users, thereby standardizing behaviour among all involved peers.

Unlike other hybrid XR telepresence systems, such as Meta Horizon, where participants joining with a screen device (e.g., a laptop) are shown on a virtual display within the VR scene, ViGather enables flat-screen participants to establish co-presence by relaying their non-verbal communication and behaviour to an avatar representation. This approach allows these traditional users to express non-verbal cues such as body language (e.g., turning to another person), gestures (e.g., pointing at objects in the VR scene), and gaze direction (eye contact), making communication more identical to real-world interactions.

VirtualNexus Huang et al. [23] presents VirtualNexus (Figure 2.3c), an asymmetric telepresence system that delivers physical locations to VR users through 360° live video streaming. In contrast to 3D reconstruction [25], the authors argue that 360° video approaches are much more cost-effective, as streaming high-quality full-scene reconstructions in real-time requires high-end sensors, computing, and network infrastructure. Despite these benefits, 360° telepresence limits remote users to viewing the physical location from a stationary perspective, whereas movement in 3D environments is much more straightforward. Also hindering from the lack of depth information, virtual object manipulation becomes challenging as objects can only float in the environment without any physicality, thereby breaking the illusion of physical presence.

To overcome these challenges, VirtualNexus introduces an approach that embeds a spatially aligned 3D reconstruction beneath the video layer. Leveraging spatial meshes generated by the AR user's Microsoft HoloLens, VirtualNexus transmits these meshes in real-time to the remote VR user, aligning them precisely with the 360° video. This alignment allows virtual objects to react physically with surfaces in the environment, enabling interactions such as drawing on walls or placing objects on desks with realistic collision behaviour. In addition, to mitigate the limitations of a stationary camera viewpoint, the system implements environment cutouts, interactive miniatures of physical regions that the VR user can extract, scale, and reposition for manipulation. This mechanism allows VR users to 'move' within the space by bringing distant elements into reach, while their virtual avatar is rendered near the original physical location in the AR user's view, preserving spatial coherence.

PanoCoach Kang, Pfister, and Lin [22] addresses the disconnect between tactical instruction and physical execution in team sports, particularly football. Traditional tools like whiteboards and video analysis often fall short in conveying dynamic spatial relationships and coordinated team movements. PanoCoach introduces a XR telepresence system combining a 2D interface for coaches with an immersive 3D VR environment for players. This setup

enables real-time tactical demonstration, allowing players to experience scenarios from a first-person perspective while receiving synchronized feedback. Unlike conventional methods, the VR environment supports nuanced elements such as player orientation, motion cues, and velocity, therefore enhancing spatial understanding.

Robotic Training Distinctively to the previous cases, Kaarlela et al. [30] demonstrated how XR environments can be applied to robotic training by incorporating DTs to create immersive, risk-free training environments. In industries where training in real-world conditions would expose trainees to potentially hazardous situations, these simulated environments allow trainees to control virtual models of actual equipment and observe realistic outcomes within a virtual setting. DTs enable bidirectional communication between virtual environments and real hardware. This setup allows for real-time data exchange, such as speed, position, and pose from the physical equipment to its digital counterpart. The approach proposed in the study involves integrating DTs that permit trainees to interact directly with real-world robot hardware via a virtual interface, providing a high-fidelity, responsive training experience.

The training scenario involves a three-dimensional AR model of an industrial robot, complete with a gripper, 3D-printed objects, and a wireframe representation of the robot's workspace. This setup presents trainees with a hands-on task where they must use AR to program the robot to stack a series of 3D-printed shapes: a cube, triangle, and cylinder. Trainees can project this AR model onto any physical setting, creating an environment in which they program the robot's actions, including approach, pick-up, placement, retraction, and gripper control. Trainees interact with this setup via a mobile-device interface, which allows them to manipulate the robot's tool center point by dragging it to specified positions. Buttons overlaid on the AR display provide gripper control, enabling users to open and close the gripper with ease. Trainees can also observe a WebRTC [14] live video feed of the physical robot, synchronizing the virtual and real environments.

2.2.1 Offloading Computation

Systems leveraging XR are known to be extremely computationally demanding, as they often rely on intensive GPU processing [11]. Workloads such as pose estimation [24, 28], 3D rendering [9, 20, 25], object detection [8, 10, 29], and Multimodal Large Language Model (MLLM) inference [21] present significant challenges for execution on mobile devices (e.g., mobile XR headsets, smartphones). Such devices are typically constrained in terms of compute power, memory, and battery life, which limits their ability to perform under these demanding conditions [9, 10].

A critical evaluation metric in immersive applications is *motion-to-photon latency*, which refers to the delay between a user's action and the corresponding graphical update in the display [10]. Since XR applications demand real-time interaction, they are extremely sensitive to latency [11, 13]. When not minimized, latency can significantly impact user QoE [9, 10]. Tasks such as human keypoint detection [28], pose estimation [24], and the rendering of complex medical scans [9] impose high throughput requirements. These demands can slow system responsiveness, leading to delayed interactions and, in some cases, motion sickness. For example, Zhang et al. [10] demonstrate that performing object detection on a mobile CPU (Snapdragon 800) results in high motion-to-photon latency. When a user moves within the AR scene, the bounding boxes identifying objects fail to remain aligned with the actual object positions. To ensure smooth interaction, XR systems must support

low-latency operation. Studies suggest that latency should remain below 15 milliseconds to avoid degradation of user experience [8, 11].

Computation offloading, fills the gap between the demanding computational needs of XR applications and the limited resources of mobile devices by transferring intensive tasks to a more powerful remote server [10]. This approach aims to reduce compute latency and conserve the mobile device's battery life, thereby preventing the poor frame rates, high power consumption, and overheating that can result from running these tasks locally [8, 29]. For example, research shows that offloading AR object detection can reduce task latency by a factor of four and decrease energy consumption by as much as 77% [8]. Furthermore, offloading enables the execution of further complex computation that would be prohibitive to run on a constrained device, facilitating higher-quality and more immersive user experiences [6, 8, 25].

Different offloading architectures have been explored to achieve these benefits. Some systems use a "split rendering" approach, dynamically balancing the rendering load between the local device and a remote server to adapt to network conditions and device workloads [9]. Others decompose the object detection pipeline into distributable microservices, which can be scaled independently across multiple servers to handle concurrent user loads [6]. While early systems offloaded to the distant cloud, this often introduced significant network delays unsuitable for real-time interaction [8, 10, 11]. Consequently, modern solutions are increasingly leveraging edge computing for placing the processing closer to clients as a path to minimize latency [8, 11, 13, 20, 21, 28]. By reducing the round-trip time, edge-assisted offloading provides a more responsive experience for interactive XR applications.

The following sections provide concrete examples of how computation offloading is applied to some workloads in XR systems. These include real-time pose estimation for avatar control, complex 3D rendering, object detection for environmental awareness, and inference from MLLM to enable new forms of interaction.

Pose Estimation

As previously mentioned, *ViGather* [24] is a VR telepresence system that, independently of the platform, normalizes the sense of presence between cross-platform users by relaying teleconferencing participants' non-verbal cues to avatars. For this purpose, the system creates upper-body avatar representations based on users' 3D body poses and voice input captured by their device sensors. These captured non-verbal cues are then translated into avatar animations that augment the spatial presence of participants in the VR meeting room.

Whether participants join the VR meeting via a VR headset or a screen device, each client renders a local VR environment of the meeting room, synchronized in real-time with other participants. Client synchronization is performed through a relay server that broadcasts information among all meeting participants. Each client shares the information required to synthesize its own avatar, consisting of 3D body motion and recorded audio. In turn, it receives the information required to render the other participants' avatars, which are then rendered within its local VR environment using Meta's Avatar Software Development Kit (SDK).

While VR headsets can natively estimate head and hand poses using their multiple camera sensors, screen-devices face a more significant challenge in this regard. These devices typically capture the user with a limited field of view, as they rely solely on a monocular

video stream from a front-facing camera (i.e., webcam). This makes 3D body pose estimation for all screen-device users particularly difficult and, consequently, more computationally demanding.

As a consequence, *ViGather* offloads the computationally intensive 3D body pose estimation for all screen-device users to a motion server, which reduces the computational requirements on participants' device hardware. Essentially, the remote 3D-body-pose estimation unfolds as follows: The client resizes each image frame received from the monocular video stream and compresses it using the Joint Photographic Experts Group (JPEG) binary format. It then forwards the processed frame as a network packet through a TCP/IP connection to the motion server and receives the corresponding pose data in return. The motion server uses MediaPipe Pose Application Programming Interface (API) [50], a machine learning pose estimation library, to estimate the body pose of users in 3D world coordinates. Thus, it takes over a major part of the computational load from the client devices and, due to a more capable back-end GPU, reduces the overall latency of the system.

In a different context but within pose estimation, Cai et al. [28] highlighted the inability of co-located AR experiences to overlay virtual identity tags on human bodies. The authors argue that distinguishing user identities in AR enhances the ability to selectively interact with someone. For instance, similar to VR first-person shooting games where users inflict damage based on whom they aim at, AR systems should also provide this kind of targeted interaction, enabling users to detect humans in their view and distinguish their identities. To this effect, the authors propose *DiTing*, a system for rendering identity tags above individuals' heads through real-time human keypoint detection and device-shared Six Degrees of Freedom (6DoF) poses.

At its core, DiTing enables ID-aware interaction at 30 Frames Per Second (FPS) by combining on-device optimization with edge-assisted computation. Running both human keypoint detection and pose estimation directly on mobile devices typically exceeds their processing limits, making it difficult to maintain smooth AR experiences. To address this, DiTing offloads human keypoint detection to an edge server, where more intensive processing can be performed. Avoiding the high bandwidth consumption and the consequent increase in transmission latency, when scaling to multiple users, frames are selectively offloaded. Only key frames are delegated for remote processing, such as when a new body enters the camera's perspective, while client-side tracking is performed locally for intermediate frames.

The system optimizes local tracking by merging device pose estimation and human keypoint tracking into a unified pipeline. Performing these tracking tasks independently can lead to redundant computations, as both may operate on the same set of feature points. To address this, the system employs a hybrid optical flow method that tracks all relevant feature points in a single computation. This includes corner points from the static environment (e.g., walls, furniture) for estimating the device's 6DoF pose, as well as dynamic keypoints on human bodies (e.g., wrists, elbows, head) for human tracking. After tracking, the system aligns user virtual identities with the detected human bodies by leveraging shared device pose information. This enables it to match each detected human body to the corresponding user based on their known device pose.

Rendering

Lu et al. [9] introduces RenderFusion, a split rendering mechanism that optimizes 3D graphics rendering for XR devices. Recognizing that rendering high-resolution 3D content on

resource-constrained devices often leads to delayed user interactions, the authors propose leveraging high-end remote compute resources at a controlled network overhead. RenderFusion addresses the challenge of rendering high-resolution 3D graphics on lightweight XR devices (e.g., phones) that lack sufficient computing power, without compromising motion-to-photon latency, a critical factor for XR experiences.

The system distributes rendering across the user's device and a remote server, balancing local rendering complexity against the network overhead of remote rendering to maximize QoE. RenderFusion is demonstrated as an integration with the *ArenaXR* platform by overlaying remotely rendered objects on its web client. Objects selected for remote rendering are processed by Unity's [51] rendering engine and streamed to the client via WebRTC [14]. Both local and remote rendering environments are synchronized via ArenaXR's message broker, ensuring object properties are updated in a distributed manner. Furthermore, this message broker interface allows RenderFusion to signal which objects each environment should render.

At its core, the system employs an optimization algorithm that dynamically determines where and how different parts (objects) of a scene should be rendered as a user traverses an XR scene. Each object can be rendered using one of three variants: locally with high visual quality, locally with low visual quality, or remotely with high visual quality. Thus, the algorithm's task is to select the appropriate variant for each object, considering factors such as object occlusion, local compute constraints, network conditions, and other aspects that may impact user QoE.

Addressing a similar challenge, Wu et al. [20] propose offloading the rendering of dynamic 3D point cloud representations in VR scenes to the edge. The authors argue that previous methods for remotely rendering these collections of XYZ points incur redundancy overhead due to the repeated loading of the same VR scene across multiple edge servers. Fundamentally, this means that whenever rendering instances need to scale (e.g., when new users join a VR session), the full 3D point cloud data must be loaded onto each new rendering instance. Realizing these redundant operations, the authors propose a method they call object-level splitting. Instead of distributing users across rendering servers, it distributes the VR scene's objects. This edge-assisted rendering technique requires clients to establish multiple connections with their respective rendering servers and then blend the rendering results from the multiple instances.

Since the loading overhead may not always be significant enough to perform object-level splitting, the authors propose an edge-rendering system named PoCIVR that selectively determines the optimal method to utilize: either object-level or user-level splitting. This decision is handled by a dynamic task scheduler that evaluates rendering requirements and edge server resources in real time. The scheduler models the trade-off between user QoE (e.g., image quality), GPU resource consumption, and rendering performance as measured by frame rate. It benchmarks object complexity and available GPU capacity to choose the optimal splitting strategy for each user group. If object-level splitting offers significant gains by reducing redundant scene loading, it is selected; otherwise, user-level splitting is used. Once a mode is chosen, the scheduler allocates rendering tasks across servers to balance workload and avoid bottlenecks. It periodically re-evaluates assignments based on updated system conditions, ensuring efficient resource use and consistent frame rates.

Object Detection

Cozzolino et al. [8] proposes a task offloading scheme for computationally intensive computer vision algorithms, a common workload for AR applications. The author identifies that executing object detection and recognition algorithms on XR mobile devices often exceeds acceptable latencies for end-users, and prolonged usage leads to battery depletion. Consequently, the author proposes exploiting nearby edge infrastructure to support these deep learning-based real-time applications.

Despite edge computing offering a path to optimize performance by offloading tasks to nearby powerful machines, this paradigm is associated with heterogeneous compute resources. Unlike the cloud and its vast data centers, edge nodes have limited computational resources, thus restricting the number of clients that can be served simultaneously. For this reason, selecting the appropriate offloading edge node must account for already overloaded nodes. Addressing this, the author presents *Nimbus*, a real-time task offloading system designed to determine an optimal task placement strategy and prevent situations where too many clients are allocated to the same node, leading to competition for limited resources.

Nimbus operates within a multi-tier edge infrastructure, where resources are structured in three layers, increasing in compute capacity and latency as tasks are placed closer to the core. To initiate offloading, mobile devices first perform a one-time benchmark procedure to assess their local capabilities, after which a handshake phase is conducted to communicate performance requirements, such as frame rate and energy constraints. Based on this input, Nimbus selects suitable edge nodes using a Pareto-optimal strategy that jointly minimizes task latency and energy consumption. Tasks are assumed to be atomic and stateless, allowing for robust execution even under fluctuating network conditions. The system accounts for both current load and queuing times at each candidate node, avoiding overloading any single edge node and ensuring balanced resource utilization. When no edge candidate satisfies the application's constraints, a failover mechanism redirects the task to the cloud as a last resort. This design allows Nimbus to dynamically adapt to infrastructure heterogeneity and concurrent usage, significantly reducing latency and mobile energy consumption.

LLM Processing

Srinidhi, Lu, and Rowe [21] propose integrating MLLMs to enable more intuitive spatial interactions in AR applications. A prominent example is a "cognitive assistant" application that uses AR annotations, such as arrows and text boxes anchored at specific 3D locations, to guide users through complex physical tasks. This approach can be applied to scenarios like providing step-by-step instructions for assembling industrial machinery or operating a new coffee machine.

Integrating MLLMs into AR systems presents notable challenges. Their high computational and memory requirements make efficient operation on mobile AR platforms infeasible. While advancements have improved memory efficiency, reduced latency, and lowered network demands, these benefits often come at the cost of reduced descriptiveness, accuracy, and general applicability. Conversely, robust inference from models like GPT-4[8], Llama[9], and LLaVA[10], with their billions of parameters and vast training datasets, deliver highly detailed

[8] OpenAI GPT-4.5: `https://openai.com/index/introducing-gpt-4-5/` (Accessed 10-06-2025)
[9] Meta Llama: `https://www.llama.com/` (Accessed 28-06-2025)
[10] LLaVA: Large Language-and-Vision Assistant `https://llava-vl.github.io/` (Accessed 28-06-2025)

and imaginative responses across a wide range of topics. However, their size and resource demands necessitate deployment on powerful servers rather than client devices.

To address these limitations, the authors propose offloading MLLM inference to the cloud, where the XR device transmits image frames and audio to a MLLM back-end and subsequently renders the generated output. Using a simple language for AR annotations, the system can draw graphical primitives, such as arrows and text boxes, anchored at specific 3D locations.

Building on these insights, the state of the art review now transitions to an exploration of how Kubernetes can support these cloud-edge infrastructure requirements and embed session-awareness into workload lifecycle management.

2.3 Deploying XR telepresence on Kubernetes

In the previous section, it was observed that deploying XR application services across edge infrastructure is a common architectural practice. This finding drives this research to explore how Kubernetes can be leveraged to manage this class of applications across the cloud/edge fabric. In the realm of service and container orchestration, Kubernetes is regarded as the primary framework due to its widespread adoption in software production environments [40]. However, Kubernetes was originally designed for cloud-centric scenarios, and prior studies have highlighted a certain degree of unsuitability in edge environments due to a specific design decision: it stores the entire cluster state in etcd [52], a strongly consistent key-value store [40].

Transitioning from a typical Kubernetes deployment within a single data centre, to a distributed group of geographically dispersed edge machines introduces the necessity of a single Kubernetes cluster to span across all infrastructure nodes. Etcd, a control-plane component that serves as the Kubernetes cluster's single source of truth, stores information about the desired cluster state, the current status of applications, nodes, and other resources [40]. Horizontally scaling this component sacrifices availability in favour of consistency, illustrating the CAP theorem's trade-off between these two properties. Given the fact that 30% of etcd requests are writes and the synchronization effort between etcd nodes may increase proportionally as cluster sizes increase, horizontally scaling etcd can become a bottleneck [40]. This issue becomes even more pronounced in an edge setting, where the Kubernetes cluster spans across network slices with varying bandwidth and latency characteristics [40].

State-of-the-art solutions propose multi-cluster Kubernetes deployments instead of single-cluster setups to achieve better scalability [12, 34, 37, 38]. While this approach overcomes the limitations of cluster size, it introduces two new challenges: multi-cluster management and inter-cluster networking.

2.3.1 Multi-Cluster Deployments

ClusterAPI (CAPI) [53] is an open-source, vendor-agnostic Kubernetes sub-project that simplifies the coordination and deployment of multiple Kubernetes clusters. It enables cluster definitions through YAML files [54], facilitating the automation of each cluster lifecycle (create, upgrade and delete). Simply put, CAPI provides a set of components that enable the creation of a Kubernetes management cluster designed specifically to manage the lifecycle of other clusters handling actual application workload. Furthermore, CAPI has various provider implementations for cluster bootstrapping and infrastructure providers. By default, CAPI

uses Kubeadm[11] for cluster bootstrapping, a tool conceived to reduce the installation complexity of a Kubernetes cluster [53]. Other provider implementations do exist, for instance, MicroK8s[12] and K3s[13], enabling the bootstrapping of lightweight Kubernetes distributions instead of full Kubernetes clusters [53]. This characteristic is particularly advantageous, as physical infrastructure at the edge tends to be resource-constrained, and running full Kubernetes clusters can consume excessive resources [12]. CAPI's infrastructure provider component also has multiple implementations, including support for Amazon Web Services (AWS), Microsoft Azure, Google Cloud, and others [53]. This flexibility prevents vendor lock-in and enables automated deployment of multiple Kubernetes clusters across heterogeneous infrastructure.

Previous work has proposed the utilization of CAPI in the context of multi-cluster management. Theodoropoulos et al. [12] introduced a multi-cluster orchestration framework designed to support XR services, leveraging CAPI as a key architectural enabler. The framework comprises a central management cluster, which hosts CAPI components and intelligent algorithms, along with multiple workload Kubernetes clusters responsible for executing XR services. The management cluster operates using Observe-Orient-Decide-Act (OODA) loops, aggregating monitoring data collected by agents deployed on the workload clusters. This data is then fed into intelligent algorithms, which determine optimal actions such as scheduling service components to the most suitable clusters, deciding when and how to perform service migrations, and scaling resources as needed. Finally, the management cluster enforces the actions recommended by the intelligent algorithms.

Similarly, Nguyen et al. [34] utilized CAPI to abstract a set of Kubernetes clusters deployed across heterogeneous infrastructures and vendors into a single management cluster, offering an approach to managing multiple Kubernetes clusters as if they were a single entity. This was achieved through the development of a concept called a logical cluster, a higher-level abstraction that unifies the behaviour of various clusters to conceptually mimic a single Kubernetes cluster. Just as a Kubernetes Deployment [1] provides a declarative mechanism for defining a set of Pods [16] to run an application workload, with the Kubernetes Deployment controller [55] continuously monitoring the cluster state to maintain the desired state, a logical cluster offers a declarative approach for managing multiple Kubernetes clusters. These clusters are defined through logical cluster resource definitions, which are translated into CAPI resources by a framework component called the Logical Cluster Provider [56]. This component ensures that the state of the logical cluster is reconciled to match the desired specification.

In addition to CAPI, other multi-cluster management frameworks also exist, including Rancher[14] and Terraform[15]. Syrigos, Makris, and Korakis [38] proposed an orchestration framework for managing Container Network Functions (CNFs) over Kubernetes-based clusters to support the research and development of advanced network technologies. The aim of their study is to enable the creation of large-scale, realistic experimentation environments by combining the computational resources of multiple research facilities worldwide. The authors propose Rancher as the tool to manage and unify the various Kubernetes clusters deployed at each research facility, citing its graphical interface and its support for Role-based Access Control

[11] Kubeadm: `https://kubernetes.io/docs/reference/setup-tools/kubeadm/` (Accessed 28-06-2025)
[12] MicroK8s: `https://microk8s.io/` (Accessed 28-06-2025)
[13] K3s: `https://k3s.io/` (Accessed 28-06-2025)
[14] Rancher: `https://www.rancher.com/` (Accessed 28-06-2025)
[15] Terraform: `https://www.terraform.io/` (Accessed 28-06-2025)

(RBAC) for downstream clusters. The latter plays a vital role in this context, as it enables each research facility to independently manage permissions for its own clusters. Lastly, Terraform offers a method for declaring infrastructure-as-code and supports multiple infrastructure providers. However, it is not free to use, which may exclude some development parties.

2.3.2 Inter-Cluster Networking

Deploying Kubernetes clusters across various compute sites often results in a fragmented network infrastructure, limiting communication between services in separate clusters. This presents a significant challenge for XR telepresence systems. As previously noted, these architectures often require session-specific application services to be deployed across multiple, geographically dispersed clusters to serve users from different locations within the same session. Consequently, inter-cluster communication between these distributed services becomes critically important. This necessitates the creation of a unified network fabric that spans all infrastructure nodes, allowing communication regardless of the cluster in which a service is deployed.

Previous work has addressed the challenge of establishing cross-cluster networking by adopting networking technologies that create virtual networks across multiple clusters. Table 2.4 outlines the technologies used in each relevant source referenced in this state of the art. The identified solutions include Liqo (four sources) [57], Cilium Cluster (two sources) [58] and Submariner (two sources) [59].

Cilium Cluster Mesh Cilium is a network plugin for Kubernetes that implements the Container Networking Interface (CNI) specification and leverages eBPF, a Linux kernel feature, to reduce resource overhead [58]. What this means is that it uses Linux primitives to employ a sidecar-less architecture, allowing native performance via tunneling or direct-routing without requiring any gateways or proxies [60]. Furthermore, Cilium provides two routing modes one by establishing an overlay network and other by native-routing [58]. By default, Cilium creates an overlay network as routing strategy with UDP-based packet encapsulation [58]. This enables the abstraction of Pod Classless Inter-Domain Routing (CIDR) blocks—the range of IP addresses allocated to Pods within a specific node in the cluster—from the underlying network, provided that cluster nodes can communicate with each other using IP/UDP [58]. Traffic encryption is also supported through either IPsec or WireGuard [58]. Alternatively, native routing provides a packet forwarding mode that leverages the Linux kernel's routing subsystem, allowing packets to be forwarded as if they were emitted by a local process [58]. While the overlay network approach simplifies operational complexity by avoiding the need to route Pod IPs, it imposes additional computational overhead and reduces network throughput due to packet encapsulation. To address these trade-offs, a hybrid-routing mode is suggested to enable the use of direct routing when available, typically within a local cluster. For communication across multiple clusters, the mode falls back to tunneling.

Submariner While Cilium focuses on sidecar-less, kernel-based solutions, Submariner takes a different approach for enabling direct networking between workloads distributed across different clusters [59]. Although both tools offer similar functionality, their inner workings differ, with Submariner shifting away from a native setting by utilizing gateways to establish the overlay network [59]. Each participating cluster deploys a gateway engine responsible for

Ref.	Title	Cilium Cluster	Submariner	Liqo	Bespoke Solution
[12]	"Cross-Cluster Networking to Support Extended Reality Services"			✗	
[36]	"Latency-aware Scheduling in the Cloud-Edge Continuum"			✗	
[33]	"Live Migration of Multi-Container Kubernetes Pods in Multi-Cluster Serverless Edge Systems"			✗	
[37]	"Intelligent Multi-Domain Edge Orchestration for Highly Distributed Immersive Services: An Immersive Virtual Touring Use Case"			✗	
[34]	"A Design and Development of Operator for Logical Kubernetes Cluster over Distributed Clouds"	✗			
[39]	"Security automation for multi-cluster orchestration in Kubernetes"	✗			
[38]	"Multi-Cluster Orchestration of 5G Experimental Deployments in Kubernetes over High-Speed Fabric"		✗		
[35]	"FORK: A Kubernetes-Compatible Federated Orchestrator of Fog-Native Applications Over Multi-Domain Edge-to-Cloud Ecosystems"		✗		

Table 2.4: Cross-cluster networking technologies identified in the literature review.

setting up tunnels between clusters [59]. Tunneling can be either encrypted or unencrypted, with Submariner offering a pluggable component for managing tunnels [59]. By default, it uses encrypted tunneling via Libreswan but also supports WireGuard [59]. Unencrypted tunneling may be suitable for intra-cluster communication or scenarios such as on-premises clusters where the underlying network fabric is already controlled and encrypted by other means. This behavior can also be implemented with Cilium.

Liqo Liqo, similar to Submariner, uses gateways to establish an overlay network, with optional traffic encryption via WireGuard [57]. The key distinction of Liqo lies in its complementary notion of peering, which enables workload offloading between clusters. In Liqo, peering is defined as a relationship between two Kubernetes clusters, where one acts as the consumer and the other as the provider [57]. Once the peering relationship is established, an overlay network is formed between the clusters, and a virtual node is created on the consumer cluster [57]. This virtual node abstracts resources from the remote provider cluster, enabling the consumer cluster to delegate Pod execution [57]. Acting as a transparent extension of the consumer cluster, the virtual node integrates fully with the standard Kubernetes API, allowing Pods to be offloaded as if they were executed locally [57].

While all the previously discussed tools provide cross-cluster networking by sharing many similarities, Liqo stands out for its workload offloading capabilities, as highlighted in Table 2.5. Theodoropoulos et al. [12] describes deploying the Liqo control plane on a management cluster alongside CAPI, where it coordinates the network fabric across clusters.

	Cilium Cluster	**Submariner**	**Liqo**
In-cluster Network	Overlay Network, Direct-routing	Overlay Network	Overlay Network
Secure Tunneling Technology	IPSec, Wireguard	Libreswan, Wireguard	Wireguard
Workload Offloading	No	No	Yes

Table 2.5: Comparing cross-cluster networking technologies.

2.3.3 Game Server Workloads on Kubernetes

The video game industry's transition to a cloud-native operational model has introduced profound challenges for managing the lifecycle of dedicated game servers [61, 62]. Unlike traditional stateless web applications, dedicated game servers are inherently stateful, network-sensitive, and demand precise operational control [61]. Standard Kubernetes workload management capabilities, such as Deployment [1] and StatefulSet [3], while powerful for general-purpose applications (e.g., web applications, databases), fall short of addressing the unique requirements of gaming workloads [62].

Game servers often represent persistent worlds or specific match instances, requiring a stable network identity (i.e., IP address and port). Standard Deployment Pods are ephemeral and lack stable identity, while StatefulSet imposes rigid ordering constraints on updates and scaling (e.g., only deleting from the highest ordinal downwards), which is incompatible with the need to manage specific, targeted game servers [62].

Many of these game server characteristics are analogous to the requirements of the XR telepresence workloads studied in this work. The first similarity is the need for a stable network identity. Traditional Kubernetes Services [15], which are designed for load balancing across fungible, stateless replicas, are unsuitable for both use cases. Just as a game client requires a persistent, low-latency connection to a specific game server instance, an XR telepresence client needs a direct, stable connection to its session-specific application servers (e.g., a remote rendering instance). Second, both domains require specialized, domain-aware scheduling. For game servers, this is often handled by a high-level "matchmaker" API that requests server instances based on player demand. A similar requirement exists for telepresence: the infrastructure must scale workloads dynamically in response to application-level events, such as a client joining or leaving a session. For instance, a session might need to provision a new Pod performing object detection for serving each connecting client. Furthermore, these Pods should be placed in locations that minimize latency. This necessitates a mechanism for the application layer to communicate these domain-specific events to the infrastructure, enabling Kubernetes to orchestrate workloads accordingly. Finally, both cases require careful update management. A standard Pod recreation during a software update, which is typical in Kubernetes, can disconnect active players from a game or XR telepresence clients from their session, disrupting the user experience.

To further explore the concept of adapting Kubernetes for custom workloads, this section presents an analysis of two relevant projects, Agones [31] and OKG [32], two established solutions from the game server orchestration domain.

Agones Agones is an open-source platform designed to host, scale, and orchestrate dedicated game servers on Kubernetes. Its core design principle is to function as a native extension of the Kubernetes API [63], augmenting a standard cluster with a semantic understanding of game server lifecycles. This is primarily achieved by introducing new custom API resources and a corresponding custom controller, which together implement the Operator Pattern [64]. This approach provides a self-contained solution that can be deployed on any Kubernetes cluster, whether on-premises or in the cloud, without external dependencies beyond Kubernetes itself [61]. Agones does not replace Kubernetes but rather extends its capabilities. Consequently, interaction with its features is done through standard tooling, such as the kubectl command-line interface [65], just as with native Kubernetes resources.

The central abstraction Agones introduces is the Fleet [66], a custom resource that can be understood as a specialized version of a Kubernetes Deployment, but tailored specifically for game servers. A Fleet resource allows to declaratively define a collection of game servers, specifying a desired number of replicas and an update strategy. The intelligence of the system is encapsulated in the agones-controller [67], the central brain of the platform. This controller embodies the operational knowledge required to manage Fleet resources. For instance, the controller understands which Pods belong to a Fleet, when to scale the number of game servers up or down, and how to perform rolling updates by targeting only game servers that are not currently allocated to a player session.

To bridge the application layer with the infrastructure and integrate with external systems like matchmaking services, Agones provides the agones-allocator component [68]. This component offers a high-level gRPC[16] and REST service that decouples the matchmaker from the underlying complexity of the Kubernetes API. A matchmaker can simply request a ready game server via an API call, and the allocator handles the process of finding and allocating a suitable server from a Fleet. This process is complemented by the Agones SDK, which is integrated into the game server binary, allowing the server to communicate its state (e.g., Ready, Allocated, Shutdown) back to the agones-controller.

OpenKruiseGame OKG is a specialized Kubernetes workload for managing game servers, functioning as a sub-project of the broader OpenKruise [69] application management suite. Its architecture is explicitly layered, leveraging the foundational Pod management capabilities of its parent project. This creates a dependent architecture where OKG provides the game-specific semantics, while the core OpenKruise controller provides the underlying workload management [32]. OKG's design principles are similar to those of Agones. Both projects extend the Kubernetes API to manage game servers. Similar to how Agones uses a Fleet, OKG employs a GameServerSet [70] to declaratively manage a game server collection.

The central philosophy of OKG is its commitment to a declarative, externally managed state model. Unlike Agones relying on an SDK for the game server process to manage its own state, OKG treats the game server as a passive entity whose lifecycle is controlled by external systems. The primary mechanism for this control is the operational state field on the game server Pod [71]. An external entity, such as an operations platform, can patch this field to signal intent. At a higher level, this mechanism transforms server lifecycle management

[16] gRPC: `https://grpc.io/` (Accessed 28-06-2025)

into a declarative, operations-driven process [72]. Control over a server's state, whether it is active, undergoing maintenance, or being retired, is handled externally, rather than being hard-coded into the application itself.

This design choice creates a fundamental difference in the responsibility of control when compared to Agones. In Agones, control is internal; the game server process itself drives state transitions by calling SDK functions. The decision to terminate is made from within the Pod. In OKG, control is external; lifecycle events like scaling and updates are initiated by changing the GameServerSet specification or patching an individual game server's operational state. The decision to terminate a server is made from outside the Pod.

This distinction is crucial and leads to different operational paradigms and ideal use cases. Operating Agones feels like managing a dynamic pool of requestable resources, making its internal control model highly effective for session-based games where the server itself is the best authority on when a match is over. In contrast, operating OKG feels like managing a stateful application workload. Its external control model is superior for managing persistent, long-running servers, such as those hosting Massively Multiplayer Online (MMO) worlds, where the decision to take a server down for maintenance or scaling is an operational one, not one driven by the game logic itself [73].

2.4 Conclusions

This state of the art review conducted an analysis of 28 studies, investigating the challenges of deploying XR telepresence applications to Kubernetes. Guided by two research questions, the review focused on two primary areas: the architectural design of XR telepresence systems, and the application of Kubernetes for managing their workloads.

The findings reveal that standard Kubernetes lacks support for hosting such applications, a limitation stemming from several key characteristics: their stateful, session-based nature; the need for a stable network identity; and the complexity of scaling workloads for geographically distributed clients. This results in the need for a highly intricate, coordinated infrastructure that conventional Kubernetes primitives do not natively provide.

RQ1 - What are the architectural characteristics of XR telepresence applications?

As established in the reviewed literature, XR telepresence aims to enhance remote collaboration by overcoming the limitations of traditional 2D systems; such limitations present themselves as the lack of spatial presence, gaze awareness, and body orientation. Achieving greater levels of immersion increases architectural complexity, resulting in a hard-to-accomplish set of requirements.

A primary characteristic is high computational demand. Among the reviewed systems, workloads spanning pose estimation, 3D rendering, object detection, and MLLM were identified. These all share a common characteristic; they require intensive GPU processing. Since client devices have constrained compute resources, executing these workloads locally leads to unacceptable latency, poor frame rates, and high power consumption. Consequently, a common architectural pattern is computational offloading, where intensive tasks are transferred to more powerful remote servers. The consensus in the reviewed literature is to place these servers at the network edge, as close to the end-users as possible to minimize network delay.

Second, these offloaded functions are session-bound. Unlike generic web services, these workloads are dedicated to the clients of a specific session, and distinct workload topologies may exist, according to the variety of sessions that the system provides. For example, one session might require a dedicated object detection service for each client, while another might use a single remote rendering service for every five clients. This implies that the infrastructure must be aware of session-level requirements, scheduling compute resources not just accounting for individual users, but also considering the sessions they belong to.

Finally, this session-bound model imposes critical networking and scaling requirements. Communication is often a continuous data stream, not a series of discrete Hypertext Transfer Protocol (HTTP) requests. This necessitates a stable network identity for each workload instance, as clients require a persistent, direct connection. Furthermore, the infrastructure must be elastic. As the number of users in a session fluctuates, the corresponding session-bound workloads must scale dynamically to meet demand, ideally on the most performant and geographically closest infrastructure nodes to minimize latency.

RQ2 - How can XR telepresence applications be deployed on Kubernetes?

Deploying XR telepresence applications on Kubernetes requires addressing the challenges of geographically distributed infrastructure. A single Kubernetes cluster spanning multiple edge locations is often impractical, as the etcd data store can become a performance bottleneck. As a result, the state-of-the-art approach, identified in the literature, is to use a multi-cluster architecture, where each location operates as an independent cluster.

This multi-cluster approach, however, introduces its own complexities, such as cluster lifecycle management and inter-cluster networking. The literature review identified several tools that address these generic multi-cluster problems. For example, CAPI provides components to automate the lifecycle of workload clusters from a central management cluster. Liqo, on the other hand, creates an overlay network to enable inter-cluster communication and workload offloading. While promising, these tools solve the general problem of managing multiple clusters; they do not, by themselves, address the specific domain logic of telepresence.

The most critical requirement for XR telepresence is bridging the application's domain logic with the infrastructure. Kubernetes must be able to react to application-level events, such as a client joining or leaving a session, and translate them into infrastructure actions like scaling Pods. This review identified the domain of game server orchestration as the source of an analogous solution to the challenge of deploying XR telepresence to Kubernetes. Frameworks operating in this field, such as Agones and OKG extend Kubernetes with custom API resources and controllers (operator pattern) to embed domain-specific operational knowledge directly into the Kubernetes control plane, making the cluster itself session-aware.

In summary, the literature reveals a significant gap: while tools exist for generic multi-cluster management and for custom workload orchestration in analogous domains, there is no integrated solution tailored for the specific lifecycle of XR telepresence. This finding directly motivates the solution proposed in this dissertation. Building upon these insights, the next chapter details the design of a framework created specifically to bridge this gap and make Kubernetes better suited to host and manage this type of workload.

Chapter 3

Design

As mentioned before, Kubernetes' built-in workload management resources (i.e., Deployment [1], ReplicaSet [2] and StatefulSet [3]) do not satisfy the requirements for managing the workload of XR telepresence applications. Nevertheless, Kubernetes was inherently designed with extensibility in mind and offers a variety of well-documented extension points to adapt the cluster's behaviour according to the specific demands of its operating environment [74].

This chapter is heavily influenced by the conclusions drawn in the state of the art review. As established previously, the primary challenge is enabling Kubernetes to react to application-level events, such as clients joining or leaving a session. Drawing upon insights from the literature review, this work deems that the Agones project, from the domain of game server orchestration, provides the most relevant existing model for solving this problem. Agones successfully bridges this application-infrastructure gap with a dedicated SDK, which allows a game server to communicate its state directly to the Kubernetes control plane. Therefore, this work adopts a similar philosophy, aiming to create a mechanism that endows Kubernetes with the necessary session-awareness for XR telepresence workloads.

While the aim is to support multi-cluster environments, this work deliberately focuses on a specific subset of multi-cluster challenges. The literature review identified promising tools including CAPI for cluster lifecycle automation and Liqo for inter-Pod networking across clusters. However, based on the analysed XR telepresence architectures, dynamically scaling the number of clusters (the problem CAPI solves) and enabling direct service-to-service communication across clusters (the problem Liqo solves) were not considered primary requirements for this foundational version. The core objective is instead to enable the orchestration of session-aware XR workloads across a pre-existing multi-cluster topology. Therefore, the integration of tools like CAPI and Liqo is identified as a valuable direction for future work.

3.1 Requirements

Figure 3.1 presents a generalized deployment architecture of XR telepresence, derived from the literature review. The purpose of this illustration is not to give a rigid idea of a specific set of components but rather to categorize the common types of workloads found in XR telepresence systems and identify their characteristics. The diagram shows these workloads deployed in a distributed environment with two clusters (A and B), representing a network-sliced infrastructure, analogous to a cloud/edge multi-site setting.

The key principle highlighted here is that computationally intensive tasks are offloaded to session-bound remote compute servers. The figure demonstrates this with two distinct sessions. *Session-1*, for which infrastructure is coloured in yellow, utilizes dedicated *pose*

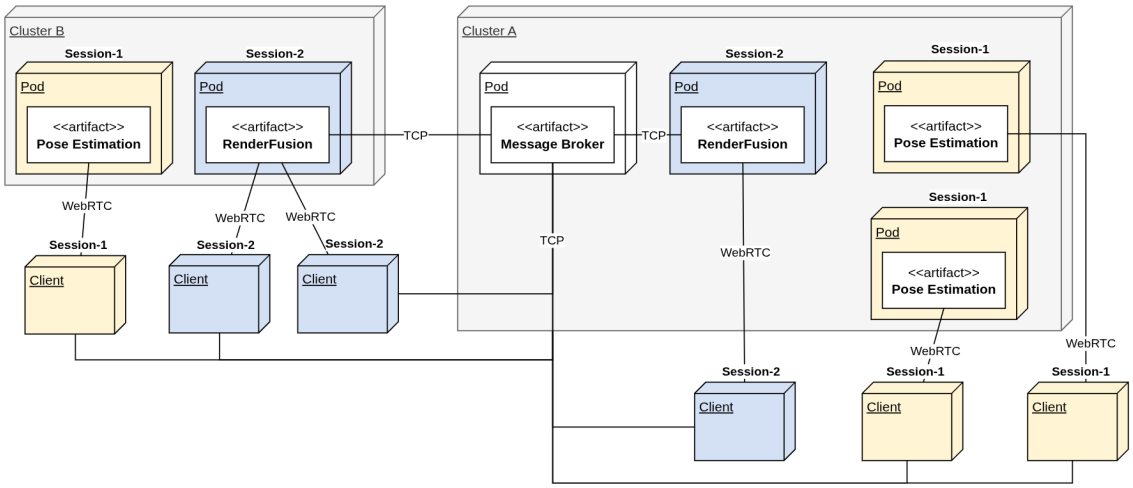


Figure 3.1: Reference deployment architecture for XR telepresence.

estimation Pods for its clients, the workload identified in ViGather [24]. In a different format, *session-2*, for which infrastructure is coloured in blue, requires a different type of offloaded function; in this case, *RenderFusion* [9] performing remote rendering. In both examples, clients connect only to the servers associated with their specific session.

In addition to session-bound workloads, the architecture can also include central, shared application services. The message broker on cluster A, for instance, is depicted as a shared component responsible for synchronizing state and interactions within all sessions, similar to its role in the ArenaXR [13] platform.

This work focuses solely on the session-bound offloaded model on which networking and scaling behaviour are critical. First, the communication between clients and their dedicated services is often a continuous data stream (illustrated here as WebRTC, following the approach of [9, 21]), which necessitates a stable network identity for each offloaded function; clients must be able to discover and connect to their specific workload instance, not a random one. Furthermore, these services must be elastic. As the number of users in a session fluctuates, the corresponding session-bound workloads must be able to scale dynamically to meet demand. This scaling logic must also account for the fact that different services have different capacities; for example, a *pose estimation* server might be designed to serve a single client, whereas a *RenderFusion* instance could potentially serve several clients concurrently. This entire process should ideally occur on the most performant and geographically closest infrastructure nodes to minimize latency.

Also, the immediate termination of Pods after a client disconnects must be prevented. This is necessary to account for temporary network issues, as it ensures a client's dedicated service remains available if they are able to rejoin the session shortly after being disconnected.

These insights lay the foundation for the proposed solution. The resulting framework is guided by the following set of requirements:

1. *Session Lifecycle Management*: The framework must provide an interface to manage the lifecycle of multiple, concurrent telepresence sessions. This includes creating sessions and tracking client activity (e.g., joining or leaving a session), and it must translate these events into corresponding workload scheduling actions within Kubernetes.

2. *Multi-Cluster Support and Location Affinity*: The framework must support workload deployment across distinct, geographically distributed clusters. It must also provide a mechanism for clients to measure network latency to these locations and select the most performant one for hosting their workloads.
3. *Configurable Session Topologies*: The framework must support different types of sessions, each with a unique, configurable workload topology. This allows for diverse deployment models; for example, one session type could require a dedicated Pod performing object detection for each client, while another type could require a single, shared Pod performing rendering for every five clients who join the session.
4. *Garbage Collection and Reuse*: The framework must provide mechanisms for efficient resource utilization. This includes automatically terminating idle Pods after a configurable grace period. Furthermore, it should support an optional Pod reuse policy by preserving idle Pods for a second configurable timeout, making them available to new clients joining the same session.

3.2 Design Alternatives: API Aggregation Layer vs. Operator

In essence, extending Kubernetes involves deploying an additional software component that integrates deeply with the platform. Some of the available extension points include plug-ins to customize Kubernetes API clients such as kubectl [65], extending the Kubernetes API itself, replacing or augmenting the default Kubernetes scheduler, and more [74]. The following subsections will explore the design alternatives considered to extend Kubernetes and support the workload management of XR telepresence applications. These alternatives both target the Kubernetes API as an extension point but take two different approaches: the API aggregation layer [75] and Kubernetes operators [64]. The rationale behind choosing the Kubernetes API as the target extension point is rooted in the need to adapt the cluster to understand the concept of a telepresence session. Consider the typical workflow in Kubernetes, for example, executing `kubectl apply -f deployment.yaml`. In this case, the kubectl command-line tool interacts with the Kubernetes API, which natively understands the concept of a Deployment, and the system responds by managing the desired workload accordingly. Similarly, the objective is to enable Kubernetes to grasp the semantics of a telepresence session. This means that when a session is created or modified (such as when users join or leave) the Kubernetes API should be capable of interpreting these changes and adjusting the workload as needed.

3.2.1 API Aggregation Layer

The Kubernetes API is exposed as a HTTP server that enables manipulation of cluster objects (e.g., Pod, Deployment, Service), persisting their serialized state by writing them into etcd [76]. To facilitate extensibility, the Kubernetes API organizes its resources (endpoints) into structured units called API groups [76], following a consistent Uniform Resource Locator (URL) pattern: `/apis/{group_name}/{version}/namespaces/{namespace}/{resource_type}/{resource_name}`[1] [63].

The API aggregation layer is a feature in Kubernetes that enables the extension of the core Kubernetes API with additional, custom APIs [75]. It provides modularity, as it allows the

[1] For instance, to PATCH a Deployment named `test` within the `test` namespace, the corresponding endpoint would be: `/apis/apps/v1/namespaces/test/deployments/test`

definition of custom resources organized under new API groups [77]. This method involves the Kubernetes API server proxying requests for resources under a certain API group to a stand-alone API server [78]. Through this mechanism, developers can implement dedicated API server extensions that provide specialized handling for custom resources within a specific group [78].

This capability opens the door to domain-specific adaptations of Kubernetes, such as integrating support for orchestrating XR telepresence workloads. Such an extension would expose a dedicated interface capable of managing sessions and related workloads in response to clients joining or leaving a session. As requests are proxied through the Kubernetes API server, the extension would process them according to the embedded orchestration logic, while simultaneously interacting with the Kubernetes API to provision or adjust the underlying workload accordingly. Figure 3.2, illustrates this idea by extending the Kubernetes API with the Session API. A brief explanation of each component includes:

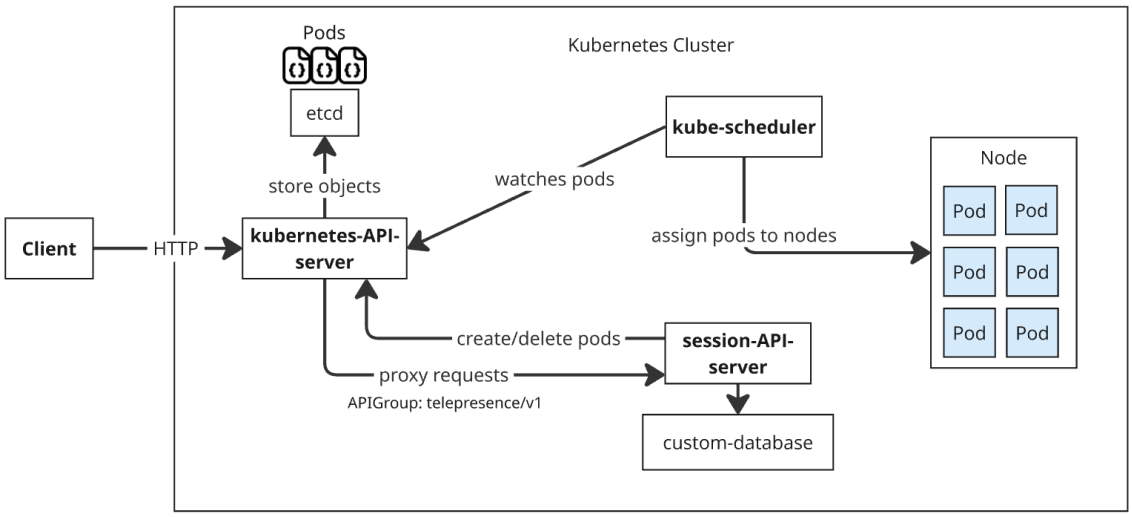


Figure 3.2: Design alternative: Using the API aggregation layer to manage the workload of XR telepresence applications.

- *kubernetes-API-server*: Core component of the Kubernetes system that exposes an HTTP API for object manipulation and uses etcd as its backing store to persist cluster state [76].
- *session-API-server*: Extension API server that understands the concept of a telepresence session and provides an interface for XR telepresence applications to interact with the Kubernetes cluster. It exposes operations related to session lifecycle management and enables the dynamic deployment of workloads as required.
- *kube-scheduler*: A Kubernetes component responsible for assigning newly created Pods to nodes within the cluster [79].
- *client*: This can be either kubectl, a command-line tool used to communicate with the Kubernetes API [76], or a framework-specific client managing telepresence sessions (e.g., creating sessions, registering users within sessions).

To configure the aggregation layer, an APIService [63] resource must be created in Kubernetes. This object determines which client requests should be proxied to the extension server—in this case, the *session-API-server*. Specifically, the APIService definition includes

a group and version, ensuring that all requests directed to resources within this group and version are forwarded to the corresponding extension server. As illustrated in Figure 3.2, the *session-API-server* handles requests for resources under the `telepresence/v1` API group.

In this context, the *session-API-server* could expose a custom resource named `session`, whose schema uniquely identifies a session and maintains a list of clients connected to it. For example, to create a session, a client would issue a POST request to `/apis/telepresence/v1/session`, which the *session-API-server* would process and persist to its database. When users join or leave a session, the corresponding `session` resource would be updated via a PATCH request. Based on these updates, the *session-API-server* would interact with the *kubernetes-API-server* to adjust the underlying workload—for instance, by creating or deleting Pods. The *kube-scheduler*, a core component of Kubernetes, would then assign the Pod to a suitable node for execution [79].

3.2.2 Operator

By design, the Kubernetes API enforces a declarative approach [78] for managing its built-in workload management resources [1–3, 80–82]. In practice, this means that API resources are limited to storing and retrieving structured data, without directly impacting the cluster workload environment [78]. For example, when deploying a Pod, the process unfolds as follows: a client issues a request to the Kubernetes API server to create a Pod object, which is then persisted in etcd. Subsequently, the Kubernetes scheduler monitors for unscheduled Pods and assigns them to appropriate nodes for execution. This sequence illustrates a clear separation of responsibilities: while the Kubernetes API is responsible for storing the cluster's desired state, the actual progression toward that state is handled by the scheduler, a special kind of controller [74].

A Kubernetes controller is a client of the Kubernetes API [74] that runs a non-terminating loop, monitoring the cluster's state and ensuring that it progresses towards the desired state [83]. This behaviour of continuously regulating the state of a system is known as a control loop and describes the core mechanism of a Kubernetes controller [83]. In more granular terms, the cluster's state is expressed as two separate fields inside a Kubernetes persistent object: `.spec`, which defines the desired state, and `.status`, which reflects the current state [84]. Access to these objects is provided through the Kubernetes API server under their respective API resource [78]. Each API resource provides the manipulation of a persistent object respecting a given kind [78]. In Kubernetes, a kind acts as a blueprint or schema for objects instantiated within the system (e.g., the Deployment API resource defines the Deployment kind) [84].

Essentially, a controller monitors at least one Kubernetes resource type [83]. When an object of that type is created or updated, particularly when its `.spec` field changes, the controller reads the new desired state, performs any required operations (e.g., creating or deleting Pods), and subsequently updates the object's `.status` field to reflect the current observed state [74]. As an example, the ReplicaSet resource is designed for workload management [2]. Its kind includes a `.spec` that allows the definition of the number of Pod replicas to be deployed within the cluster [2]. When a request is made to the Kubernetes API server, the desired state of the resource is declared (e.g., if the number of Pod replicas is set to 3, the desired state corresponds to 3 Pod replicas). The API server itself does not perform any actions to progress the cluster state. Instead, it adheres to a declarative paradigm, in which the request simply creates a record of the intended state.

This is where the controller comes in. In addition to these built-in resource types, Kubernetes also provides their respective controllers. In this case, the ReplicaSet controller will monitor resources (objects) of the ReplicaSet kind and manage the state by interacting with the cluster API server [85]. When the controller detects a new ReplicaSet resource, it will progress the cluster state by calling the API server to create the required number of Pod replicas. If the `.spec` field is subsequently updated to reduce the desired number of replicas, the ReplicaSet controller identifies that the current state exceeds the desired configuration and proceeds to delete the unnecessary Pod replicas to re-establish alignment between the actual and desired states [85].

Operator Pattern In addition to the API aggregation layer, Kubernetes provides another mechanism for API extension through a dedicated API resource: `/apis/apiextensions.k8s.io/v1/customresourcedefinitions` [63]. This resource manages objects of a special kind known as CustomResourceDefinitions (CRDs) [86], which enable the declaration of custom API resources, with user-specified kinds (i.e., schemas), under new API groups [86]. In practice, once a new CRD object is registered in the cluster, the Kubernetes API server automatically exposes a corresponding RESTful resource path for handling objects of the custom kind described by the CRD.

Unlike the API aggregation layer, extending the Kubernetes API through a CRD does not allow control over the resource's API implementation [78]. In this model, the behaviour strictly adheres to the same declarative paradigm as built-in resources (e.g., Pod, Deployment, ReplicaSet). Consequently, defining a custom resource via a CRD merely enables the storage and retrieval of objects of its kind as the Kubernetes API server itself does not autonomously act to progress the cluster [78]. This responsibility, therefore, falls to a custom Kubernetes controller, which must encode the domain-specific knowledge required to manage the application workload that the custom resource is intended to support. This illustrates the essence of the operator pattern. Its core purpose is to enable the implementation of domain-specific logic for managing application workloads as a Kubernetes-native software extension [64]. At its core, an Operator is a custom Kubernetes controller linked with a custom resource type [64].

Figure 3.3 presents an alternative architectural approach, where the *session-API-server* described earlier is replaced by a *session-controller*. In this configuration, the *kubernetes-API-server* is extended via a CRD that introduces a new resource kind named Session. Consequently, Session objects are handled like any other native Kubernetes resource and are persisted in etcd. The procedure for session creation remains largely unchanged: a client issues a POST request to the endpoint defined by the CRD, such as `/apis/telepresence/v1/session`. The fundamental difference lies in how this resource behaves. Unlike the previous architecture, where the *session-API-server* actively processed requests and initiated workload changes, a Session resource defined via a CRD remains purely declarative. Its creation or modification does not directly trigger Pod operations. Instead, responsibility for enacting changes falls to the *session-controller*.

Broadly speaking, orchestrating an XR telepresence application involves deploying specific workloads for users participating in a session. If this orchestration were performed manually by a human operator, continuously watching Session resources, the logic might resemble the following: upon the creation of a Session object via the *kubernetes-API-server*, the operator identifies the application services required for the users in that session and proceeds to deploy the corresponding Pods. If a session object is updated to reflect the departure or arrival of

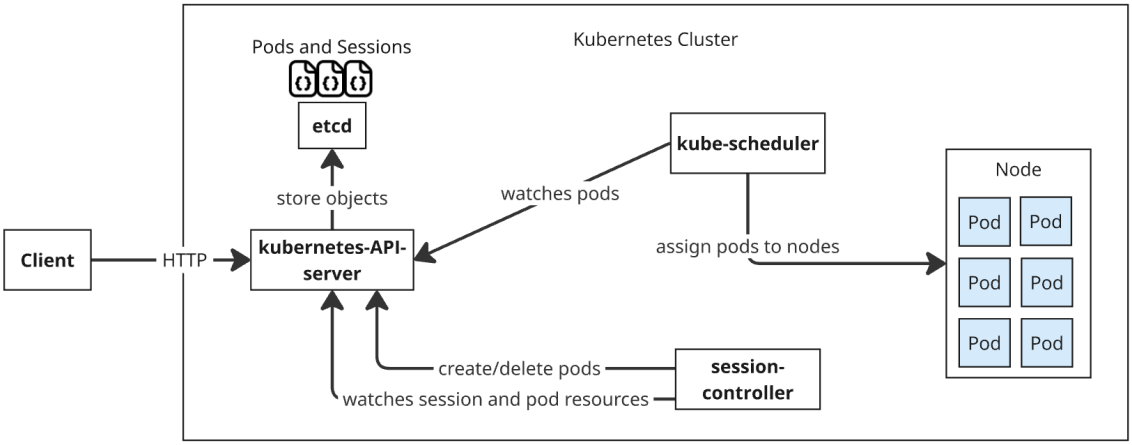


Figure 3.3: Design alternative: Using a Kubernetes operator to manage the workload of XR telepresence applications.

a client, the operator would then adjust the number of Pods to reflect the desired state. Essentially, the operator pattern encapsulates this domain-specific logic within a custom controller, automating behaviour that extends beyond the default capabilities provided by Kubernetes.

3.2.3 Extension Strategy: Rationale and Selection

While the operator pattern extends the Kubernetes declarative control plane philosophy, the API aggregation layer provides a mechanism for procedural extension of the API server. The selection, therefore, is not a matter of inherent superiority but of architectural suitability for the problem at hand.

The foundation of the operator pattern lies in the CRD. CRDs provide the definition of Kubernetes API resources allowing to define new kinds of objects that the *kubernetes-API-server* can understand and manage. In essence, this method teaches the Kubernetes API a new vocabulary. When a CRD is created, the *kubernetes-API-server* dynamically creates a new RESTful resource path for the specified API group and version, handles the storage of these new objects in etcd, and serves them just like native resources. The operator pattern represents the embodiment of Kubernetes' declarative philosophy. It combines a CRD with a custom controller that continuously works to reconcile the cluster's actual state with the desired state declared in a custom resource instance. Its primary purpose is to capture the domain-specific operational knowledge of a human expert and encode it into software.

This is the usual pattern for automating the complete lifecycle of complex applications, particularly stateful ones [87]. For example, database operators (e.g., PostgreSQL Operator [88], MongoDB Operator [89]) manage not only the initial deployment but also automate "Day 2" operations such as failover, backups, and upgrades. A common task performed in high-availability database clusters is managing distinct Kubernetes Services [15] to route traffic [90]. One Service for read-write operations pointing to the primary instance, and another for read-only traffic load-balanced across replicas. Upon primary failure, the operator can automatically promote a replica, update the Service endpoint, and provision a new replica to restore the desired level of availability. This level of automation is central to the operator's value proposition.

In contrast, the API aggregation layer provides a mechanism for a more low-level, procedural extension of the Kubernetes API. This approach involves deploying a stand-alone extension API server that registers itself with the main *kubernetes-API-server*. The API server then acts as a proxy, forwarding requests for a specific API group to this extension server. This model grants complete control over the API mechanics, enabling functionalities impossible with the CRD-based operator pattern. These capabilities include custom verbs, that is, implementing non-Create, Read, Update and Delete (CRUD) operations and using custom storage back-ends other than the cluster's etcd instance.

A great example utilizing the aggregation layer is KubeVirt [91], a project for running Virtual Machines (VMs) in Kubernetes. While KubeVirt uses a CRD, the VirtualMachineInstance (VMI) resource [92], to represent the declarative state of a VM, it also deploys an extension API server to handle imperative actions. This server implements custom sub-resources on the VMI object, such as `/console` (for command-line access), and `/restart` (for restarting VMs) [77]. Requests to these resources are proxied to the KubeVirt extension server, which contains the procedural logic to execute the intended operations. Such imperative actions cannot be modelled with a standard CRD.

Ultimately, the decision between these patterns hinges on whether the goal is to automate application management or to fundamentally alter the API's behaviour. In the context of this work, the requirements for managing XR telepresence sessions align directly with the strengths of the operator pattern. The requirements described earlier necessitate the automation of workload lifecycles for managing Pods based on session state, which is the core competency of an operator. There is no requirement for custom API verbs, alternative storage back-ends, or other advanced features provided by the aggregation layer.

Furthermore, adopting the API aggregation layer would introduce unnecessary architectural complexity and operational risk. An extension API server is a component that must be separately developed, deployed, and maintained. Its failure can cascade and impact the availability of the Kubernetes control plane [93]. Therefore, for the reasons of suitability and robustness, this work follows the operator pattern to implement a custom controller for managing the lifecycle of XR telepresence applications.

3.3 Proposed System Architecture

Building upon the preceding analysis, this section details the final proposed architecture, which is illustrated in Figure 3.4. The solution extends Kubernetes with a new API for managing XR telepresence sessions, implemented via the *operator pattern*. The figure shows this architecture in a multi-cluster context, with two instances of the proposed *Session Custom Resource*: session-1 and session-2.

Conceptually, this resource represents session clients as the desired state; the presence of a client in the Session resource signifies that a corresponding workload must be deployed. Following Kubernetes API conventions, this desired state is defined in the resource's `.spec` field, which specifies not only the clients present in the session but also the workload characteristics they require. The operator reconciling this resource is the *Session Controller*. It continuously observes the desired state and adjusts the application workload (Pods) to serve each registered client. As part of this reconciliation, it also updates the resource's `.status` field with workload readiness information and the endpoints for clients to connect to. All client connections are then proxied through an *NGINX reverse proxy* managed by the NGINX Ingress Controller [94].

As illustrated by the session-1 workload spanning two locations in Figure 3.4, the architecture supports a distributed, multi-cluster infrastructure. This is achieved by replicating the Session resource across all configured clusters. This multi-cluster approach allows clients to first benchmark latency to each location using the *Ping Server* and then connect to the cluster that offers the lowest network latency. When a client is registered in the Session resource of the chosen cluster, that change serves as a trigger for the local *Session Controller*. The controller then reconciles the resource, leading to the deployment of the necessary Pods in that specific location.

The *Session Manager* serves as an abstraction layer to manage the complexity of this multi-cluster process. It unifies the distributed Kubernetes API servers into a single interface that translates high-level domain operations, such as registering a client in an optimal location, into the requisite, cluster-specific API calls. This design aims to simplify integration, enabling the application runtime to directly request compute infrastructure as clients join or leave a session.

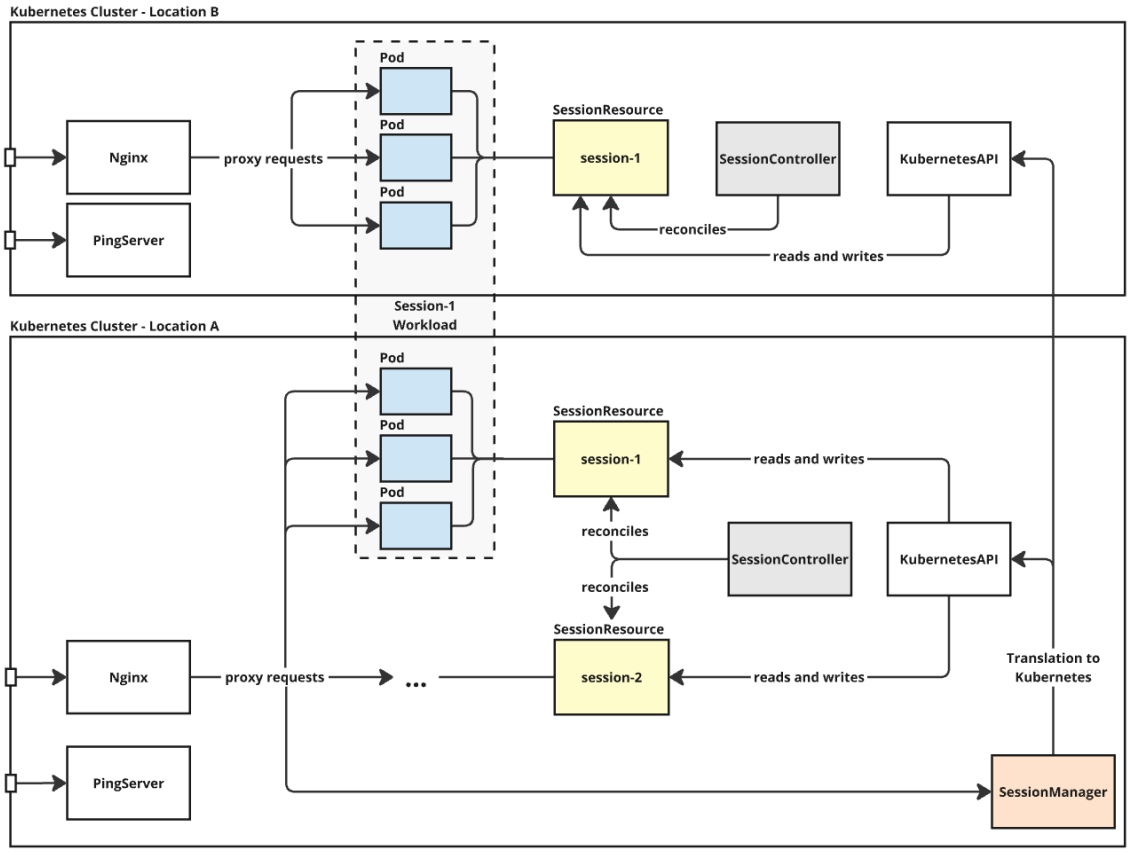


Figure 3.4: High-level architecture of the proposed solution, featuring a Session Manager that provides a unified interface to a multi-cluster environment. In each location, a Session Controller reconciles the local Session Custom Resource to deploy workloads based on client needs, with an NGINX proxy managing external access.

Chapter 4

Implementation

This chapter transitions from the high-level architecture presented previously to the concrete implementation details of the proposed framework. It provides an in-depth exploration of the system's core components, beginning with the *Session Controller* and the schema of its associated *Session Custom Resource*. Subsequently, it examines the implementation of two other controllers utilized in the system, *Network Controller* and *Garbage Collection (GC) Controller*, detailing their respective roles in providing stable network reachability and resource GC. This is followed by a description of the *Session Manager*, the HTTP API that serves as the unified entry point for the multi-cluster infrastructure. Finally, the chapter examines the integration of STUNner [95], a third-party WebRTC gateway required to handle Network Address Translation (NAT) traversal in WebRTC Kubernetes workloads.

4.1 Session Controller

The session controller, a custom Kubernetes operator, forms the core of this framework. It continuously reconciles the *Session Custom Resource*, coordinating the orchestration logic needed for XR telepresence applications to be deployed within a Kubernetes cluster. Although Figure 3.4 depicts the Session Controller as a single logical unit, its implementation is split across three distinct controllers. These include the Session Controller itself, responsible for reconciling the Session custom resource; a *Network Controller* designed to ensure targeted reach for Pods assigned to specific users; and a *GC Controller* handling Pod re-utilization and deletion throughout a session lifecycle.

4.1.1 Session Custom Resource

The session controller is responsible for reconciling the Session Custom Resource. This resource extends the Kubernetes API, providing a native way to orchestrate telepresence workloads similarly to other built-in resources. As a custom resource, it requires a dedicated controller to manage its state. The architecture of this resource is illustrated in Figure 4.1, followed by a description of each of its attributes.

Session

- *metav1.TypeMeta*: Standard Kubernetes field which indicates the type of object and its API schema version.
- *Meta*: Standard Kubernetes field which all persisted resources must have. It includes the name and namespace of the resource.
- *Spec*: Standard Kubernetes field which indicates the desired state of the resource.

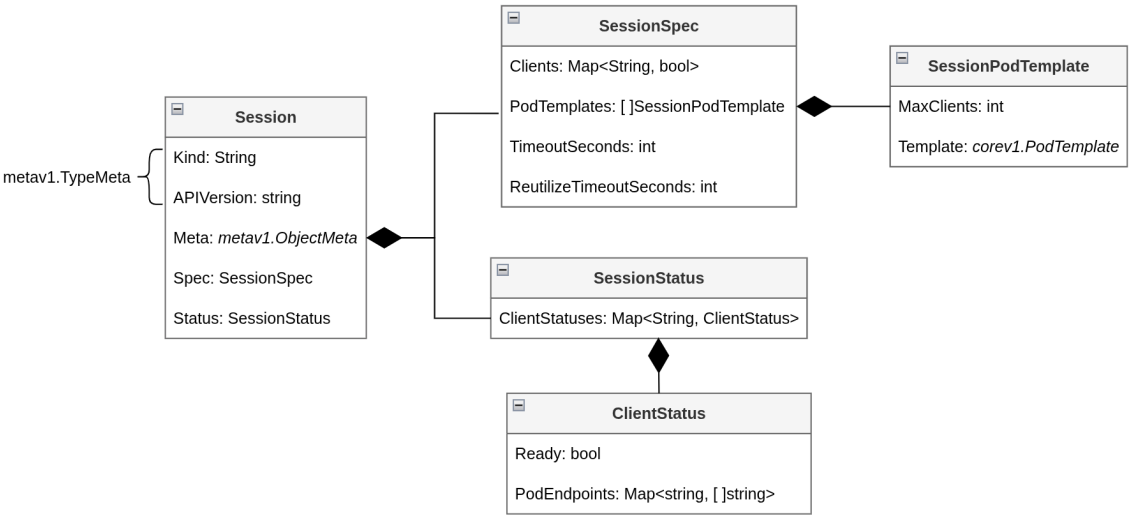


Figure 4.1: A detailed view of the Session Custom Resource schema. The Spec entity outlines the desired state, including the list of active Clients, the required PodTemplates, and idle Pod timeouts. The Status entity reflects the actual state of the cluster, providing clients with the necessary connection PodEndpoints and their Ready state.

- *Status*: Standard Kubernetes field which indicates the current state of the resource.

SessionSpec

- *Clients*: Mapping between client identifiers and a boolean value representing their connection state. When a client joins a session, their identifier is added to this mapping with the value set to true, indicating that client is active and requires compute resources (Pods). Conversely, when a client explicitly leaves the session, their identifier is removed from the mapping. If the application runtime detects that a client has unintentionally lost connection, its corresponding value in the mapping is changed to false. This state signifies that the client still holds allocated Pods but is not currently active. Each of these modifications (i.e. clients joining, leaving, or disconnecting), triggers a controller reconciliation that adjusts the session's workload to reflect the current number of clients and their connection state.
- *PodTemplates*: Collection of Pod templates (blueprints) the session requires to execute its workload (e.g., rendering, object detection) and serve its clients.
- *TimeoutSeconds*: Duration, in seconds, before a Pod is either deleted or scheduled for re-utilization if a client unintentionally loses connection. This prevents clients from losing their allocated compute resources due to temporary disconnections. If set to 0, Pods are immediately deleted or scheduled for re-utilization as soon as a client's flag in the Clients mapping is set to false.
- *ReutilizeTimeoutSeconds*: Duration, in seconds, a Pod has to be re-utilized by another client joining the session before it's deleted (further explained in section 4.1.4). If set to 0, Pods are immediately deleted as clients disconnect from the session.

SessionPodTemplate

- *MaxClients*: Maximum number of clients a Pod can serve.

- *Template*: Pod template used by the Session Controller to instantiate Pods for clients joining the session.

SessionStatus

- *ClientStatuses*: Mapping between client identifiers and their status. Essentially, the client status specifies the endpoints through which a client can access its allocated compute resources (Pods), and whether these are ready for use.

ClientStatus

- *Ready*: Indicates whether the client's allocated Pods are ready for utilization. This status is true if all Pods allocated to the client are ready. This top-level ready condition is inspired by Kubernetes API conventions, which suggest a common top-level condition that summarizes more detailed conditions [84].
- *PodEndpoints*: Mapping between the Session Pod templates' identifiers and the endpoints of their respective Pod instances, through which clients can connect to.

4.1.2 Implementing Kubernetes Controllers

Kubebuilder [96], a framework for building Kubernetes APIs (custom resources) and their corresponding operators, was used to develop the three project controllers: the Session Controller, the Network Controller, and the GC Controller. Kubebuilder scaffolds a Go [97] codebase with boilerplate for defining custom resource types, their reconciliation logic, and associated tests. It also provides a Makefile[1] with predefined automations for tasks like generating YAML[2] configurations and deploying all necessary components to a cluster.

While Kubebuilder is a high-level framework, the underlying tooling for controller development relies on two foundational libraries: controller-runtime [98] and client-go [99]. The controller-runtime library builds upon client-go (the official Kubernetes Go client) to offer higher-level abstractions, such as shared dependencies and optimized reconciliation mechanisms. Typically, controllers interact with API objects using a cached client for reads (to minimize API server load) and a direct client for writes. In scenarios with multiple controllers, creating separate clients and caches for each one introduces significant resource overhead. Controller-runtime addresses this by providing a central Manager component. The Manager instantiates a single, shared client cache utilized by all controllers registered with it, thereby optimizing resource usage. Alongside the Manager, controller-runtime provides other two core components, the Controller and Reconciler. Controllers launch worker Goroutines to invoke the reconciliation logic defined in its associated Reconciler. Furthermore, Controllers can be configured with predicates, which act as filters to control which events trigger a reconciliation.

While it is technically feasible to build these controllers using only controller-runtime or client-go, Kubebuilder was chosen for its automated development process. It automates critical tasks such as generating resource manifests from Go structs and deploying all components into a cluster using Make commands.

Figure 4.2 illustrates how the client-go library architecture supports the development of Kubernetes controllers. At the core is the *Informer*, a caching and event dispatch mechanism

[1] Learn Makefiles: `https://makefiletutorial.com/` (Accessed 28-06-2025)
[2] YAML Web Site: `https://yaml.org/` (Accessed 28-06-2025)

that underpins both the controller's cached client and its reconciliation triggers. Each Informer is backed by a *Reflector*, which is responsible for interacting with the Kubernetes API server and monitoring changes to specific API resources stored in etcd. This monitoring follows a list-then-watch strategy: the Reflector first issues a list request to retrieve the current state of resources and then establishes a watch to subscribe to future changes [100]. As updates occur, the Reflector delivers them to the Informer. The Informer performs two key actions: it enqueues the affected resource's key (namespace and name) into a *work-queue*, marking it for reconciliation, and updates the *shared client cache* to reflect the latest state of the resource. The library's documentation refers to this *work-queue* as a queue of events that initiate reconciliations. However, these events differ from traditional notions such as create, update, or delete; instead, each queue item simply identifies a resource that has undergone a change, without including details of the change itself. This abstraction arises from Kubernetes' level-based triggering model, which is further discussed in the following paragraph. The controller (reconciler) continuously pulls items from this *work-queue* and invokes its *reconcile function* to align the cluster's actual state with the desired state (reconcile the dequeued resource). To reduce load on the Kubernetes API server, the controller retrieves cluster resources from the Informer's cache instead of issuing direct API requests. If reconciliation fails, the controller can re-queue the resource key to attempt reconciliation again, ensuring robustness in the presence of transient errors (e.g., network issues, Pods crashing).

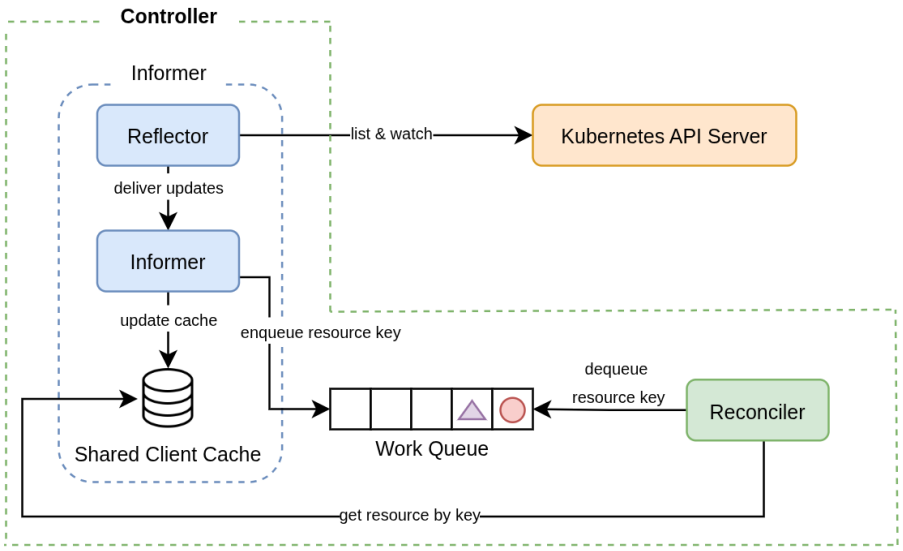


Figure 4.2: Core architecture of the client-go library, which facilitates controller development through its caching (Informer) and queuing (work-queue) mechanisms that drive the reconciliation process.

An important aspect of standard controller behaviour is that the context of an event does not affect the reconciliation process. By design, Kubernetes is a level-based rather than an edge-based system [84]. This means it does not operate based on the specific event that triggered a reconciliation; instead, it reconciles the current state by re-reading all the necessary information. These concepts derive from electronic circuit design: level-based triggering involves reacting to a state after receiving an event, whereas edge-based triggering involves reacting to a state variation upon receiving an event (e.g., create, update, delete) [101]. During the reconcile cycle, the controller is responsible for checking that the current state matches the desired state described by the watched resource. Notably, by design,

the triggering event itself is not passed to the reconcile cycle. This forces the controller to re-read and consider the entire state of the instance referenced by the event. While arguably less efficient because it requires a re-evaluation of the entire state, this level-based approach provides robust behaviour, especially in the presence of missed events or intermediate state changes [84]. For instance, consider the Session Controller watching Session resources. Suppose two new clients are registered sequentially in a Session, but the Pod hosting the controller crashes before it can reconcile these changes. Upon restart, the controller's informers repopulate its local cache with the latest state of the Session resource from the API server, which now includes both new clients. The controller then reconciles based on this final state, never having processed the intermediate state where only the first client was registered. This demonstrates the resilience of the level-based approach, as the system correctly achieves the desired state despite the crash and missed events.

Reading from a cache introduces the risk of operating on stale data, making deterministic behaviour a critical concern when developing Kubernetes controllers. As noted earlier, the client provided by controller-runtime reads objects from a local cache and writes directly to the Kubernetes API server. This design introduces a window in which the cache may be out of sync with the API server, particularly after updates. Furthermore, the cache does not guarantee immediate consistency or sequential create/get coherence [102]. Consider a scenario in which a PUT request to the Kubernetes API server registers a client in a Session resource. The Session Controller reconciles the resource by launching the required Pods and then updates the Session `.status` field to include the endpoints through which the client can access the workload. If a second PUT request registers another client in the same Session resource before the controller's cache reflects the status update from the first registration, the controller may reconcile based on stale data. Given that Kubernetes controllers do not receive the triggering event context, and operate in a level-based manner, this can lead to the controller redundantly launching Pods for the first client, even though they already exist in the cluster. To address such issues, a common practice among controllers (e.g., StatefulSet, Deployment) is to adopt optimistic concurrency control by leveraging deterministic naming conventions for created resources [103]. Since Kubernetes enforces uniqueness of object names within a namespace, attempting to create an already-existing object will result in an error. This behaviour can be used to detect and handle duplicated operations gracefully. To achieve deterministic Pod deployment, this work implements a two-step strategy. The controller first generates a unique name for each required Pod using a Universally Unique Identifier (UUID) and commits these names to the Session `.status` field. Only after this API update is confirmed does it proceed to create the Pods. This sequence guarantees that the Pod allocation is durably stored in the Session resource before any Pods are created. Consequently, the Pod creation step becomes idempotent. If a reconciliation loop is triggered by stale cache data, the controller will simply re-attempt to create Pods using the names registered in the `.status`. The Kubernetes API server will reject any requests to create Pods that already exist, allowing the controller to gracefully handle these errors and avoid redundant operations.

As previously mentioned, controller-runtime launches worker Goroutines for reconciliation, enabling concurrent processing of resources. The level of concurrency can be configured using the `MaxConcurrentReconciles` field in the Controller struct. This feature is particularly beneficial in scenarios where the states of watched objects change frequently, leading to a high volume of reconciliation requests being queued [102]. Enabling multiple concurrent reconcile loops can significantly increase throughput by draining the reconcile queue more quickly than the default single-threaded approach. However, a major concern arises

regarding concurrency handling: it's necessary to ensure that the same resource isn't being reconciled simultaneously by two or more Goroutines. Such overlapping executions can result in race conditions or unpredictable behaviour, as the state of a resource can be modified by an external side effect during reconciliation. Handling this difficulty, controller-runtime leverages the client-go library's work-queue, which by default already handles concurrency issues and guarantees that unique resources are reconciled in a sequential manner. For a more in-depth understanding of the internal implementation of the work-queue mechanism, readers are encouraged to consult the corresponding source code in the client-go repository[3] as well as the detailed explanation provided by OpenKruise [102]. Furthermore, expanding the discussion beyond controller-level concurrency, it is important to consider that other clients, such as external tools, or users, may also interact with the same resources concurrently. This can result in conflicting updates to a single resource. Even within a single Goroutine, concurrency-related inconsistencies can arise if a reconcile loop attempts to update a stale resource due to an outdated cache. To mitigate these risks, Kubernetes employs an optimistic concurrency control mechanism known as *resource versions* [84]. Every resource persisted in etcd includes a `resourceVersion` field, which is incremented automatically upon each modification. When a client attempts to update a resource, the `resourceVersion` it provides is compared against the current value. If the versions do not match, indicating that the resource has been modified in the meantime, the update is rejected by the API server with a 409 Conflict status code.

4.1.3 Network Controller

While the Session Controller schedules workload to serve clients within a session, a mechanism is required to provide stable network reachability for those Pods. This is the role of a Kubernetes Service [15], a resource designed to provide a stable network endpoint for a dynamic set of Pods.

Services were primarily designed for stateless applications where back-end Pods are fungible, for example, exposing a replicated REST API. In such cases, the client connects to the stable Service endpoint and does not need to know which specific back-end Pod handles the request. In these scenarios, Pods are often short-lived, with the number of replicas fluctuating according to user load. Without a Service, clients would need to implement a discovery mechanism to track the changing IP addresses of the available back-end replicas. The Service, therefore, decouples the client from the lifecycle of the back-end Pods, allowing the replica set to scale up or down without requiring any changes on the client's side.

However, telepresence architectures require a different approach that challenges this conventional use of Kubernetes Services. While Pods in stateless applications are fungible, telepresence workloads are stateful and therefore non-fungible. This means each client must have stable network reachability to its own dedicated Pod, not just any available instance. As reviewed in the state of the art, communication for offloaded computation (e.g., object detection, rendering) is often a continuous stream of data rather than a series of discrete requests. This type of interaction requires that clients maintain a persistent connection to their specifically allocated Pod. Therefore, unlike the stateless model where any replica suffices, the identity of the target Pod is crucial in these stateful scenarios.

[3] Client-go work-queue source code: `https://github.com/kubernetes/client-go/blob/master/util/workqueue/queue.go` (Accessed 28-06-2025)

To address the need for stable, targeted network reachability, this work adopts a *service-per-pod* pattern. This approach is inspired by the DecoratorController from the Metacontroller project [104], which creates a dedicated Service for each Pod managed by a StatefulSet. Similarly, the Network Controller implemented in this project creates a unique Service for each Pod associated with a Session resource. This pattern provides each Pod with a stable, addressable endpoint. As a result, clients can establish continuous data streams with a specific Pod by targeting its persistent Service address, rather than relying on the Pod's ephemeral IP address, which can change upon restarts.

By default, Kubernetes Services are not accessible from outside the cluster. The default Service type, ClusterIP, assigns an internal IP address that is only reachable within the cluster's network. However, since clients in this architecture must access workloads externally, a different approach is needed to expose these Services. Kubernetes provides several methods for this. One of the most basic is the NodePort type, which opens a specific port on every node in the cluster; any traffic sent to this port is then forwarded to the Service. A more robust method is the LoadBalancer type. When a Service is configured as a LoadBalancer, it provisions an external load balancer from the underlying cloud provider, which is then assigned a public, routable IP address. For example, when deploying on Microsoft Azure, this action automatically configures an Azure Load Balancer[4] to route external traffic into the cluster and to the Service.

A primary concern with the NodePort and LoadBalancer approaches is their creation of multiple, disparate entry points for external traffic, which complicates security and operational management. For example, if a cluster serves 50 clients, each requiring a dedicated Pod, using a NodePort or LoadBalancer Service for each Pod would create at least 50 distinct entry points into the cluster. To centralize traffic management, Kubernetes provides the Ingress resource [105], which acts as a reverse proxy. Although the Ingress resource is native to Kubernetes, its reconciliation is delegated to third-party components, developed by the community, known as Ingress Controllers.

This work utilizes the NGINX-Ingress Controller [94], chosen specifically for its ability to manage both Layer 7 (HTTP/HTTPS) and Layer 4 (UDP) traffic. This is a critical feature, as standard Ingress resources do not natively support UDP. The NGINX-Ingress Controller addresses this by using the standard Ingress resource for HTTP/HTTPS routing while employing a separate mechanism for UDP services via a dedicated ConfigMap. Updating this ConfigMap dynamically reconfigures the underlying NGINX proxy to listen on specific UDP ports and forward traffic to the corresponding Services. This dual approach allows all external traffic to be consolidated through a single entry point, an NGINX Ingress Controller Service of type LoadBalancer. Consequently, all inbound traffic, whether HTTP on port 443 or a UDP service on port 5060, is directed to a stable public IP address.

The interaction between these components is illustrated in Figure 4.3. The process begins when a client's registration modifies a Session resource. Using the standard client-go work-queue pattern, the Session Controller, which watches Session resources, dequeues the resource's key to begin reconciliation. Its primary task is to create the necessary Pod to serve the client. The creation of this new Pod is, in turn, detected by the Network Controller, as it is configured to watch for Pods associated with Sessions. This controller then dequeues the key for the new Pod and enters its own reconciliation loop. Here, it performs several key actions: first, it creates a dedicated Service for the new Pod; next, it updates

[4] Azure load balancer: `https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview` (Accessed 28-06-2025)

the NGINX Ingress resources (i.e., Ingress for HTTP, ConfigMap for UDP) according to the protocol specified in the Service ports. Finally, it registers the public-facing endpoints in the `.status` field of the parent Session resource. This last step makes the connection information available to the client, allowing it to connect to its allocated workload.

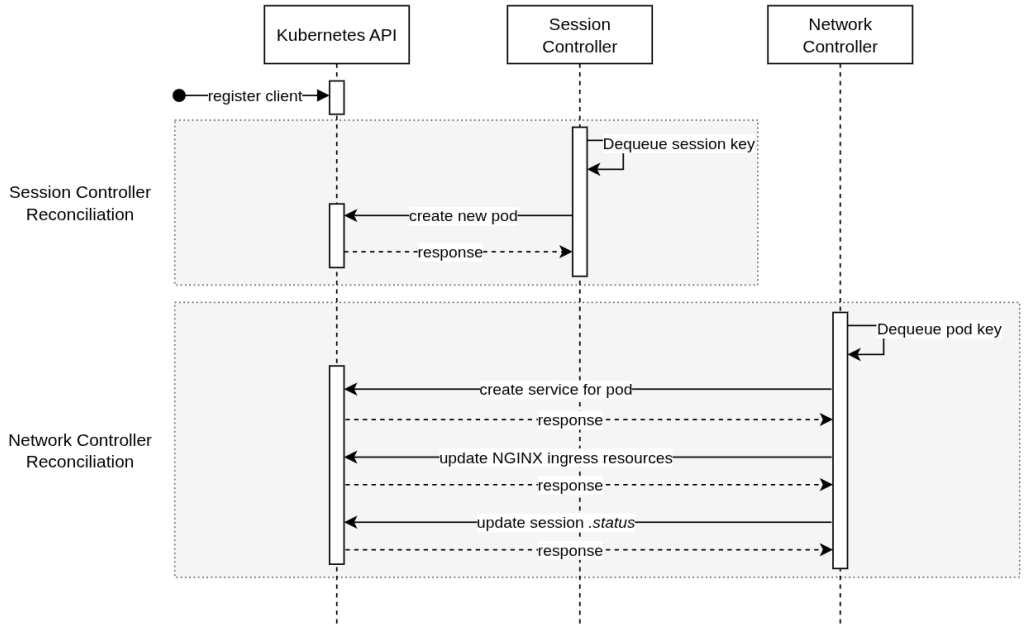


Figure 4.3: Sequence diagram of the two-stage reconciliation process. The Session Controller first creates the client's Pod, which in turn triggers the Network Controller to establish external network access by creating a dedicated Service and updating the Ingress resources.

4.1.4 Garbage Collection Controller

The GC Controller is a component responsible for garbage-collecting workloads in a telepresence-friendly manner. It provides the flexibility to keep workloads allocated for clients who temporarily lose their connection, rather than immediately deleting them. Additionally, it allows Pods to be reused; instead of terminating an idle Pod immediately, the controller keeps it in a waiting state for a configured period, allowing it to serve a new client entering the same Session. This reuse mechanism minimizes the overhead associated with starting new Pods. Currently, Pod reuse is supported only within the same Session, but making this a configurable option to allow reuse between different Sessions is planned for future work.

Unlike conventional Kubernetes controllers that are triggered solely by resource changes, the GC Controller primarily operates on a time-based reconciliation loop. The process begins when a Pod becomes idle, at which point the Session Controller creates a corresponding *GCRegistration resource*. This emission occurs whenever a client is deleted from a Session or its connection state is set to false. The GCRegistration resource is a simple data structure containing metadata about the idle Pod, including a reference to it and its current timeout phase, all defined in its `.spec` field. While the initial creation of this resource triggers the first reconciliation, the controller subsequently uses a timed, repeating re-queue mechanism. This approach is well-suited for time-based operations, as it provides a continuous loop to check if the timeout periods for idle Pods have expired. To implement this, the controller's reconciliation loop returns a `RequeueAfter` result, scheduling the next reconciliation after

a specified duration. In each cycle, the GC Controller iterates through all GCRegistration resources and deletes any Pod whose idle time has exceeded its grace period.

The timeout phase within the GCRegistration resource indicates whether the idle Pod is in a grace period for client reconnection (WaitForReconnection) or is available for reuse by a new client (WaitForReuse). As detailed in Section 4.1.1, the Session `.spec` contains two fields that control this behaviour: TimeoutSeconds and ReutilizeTimeoutSeconds.

The logic governing these timeout phases is illustrated in Figure 4.4. The TimeoutSeconds field defines the grace period for a client that has unintentionally lost its connection, which is signified by its connection state being set to false. As shown in the diagram, if the client does not reconnect within this time, the GC Controller proceeds to check the ReutilizeTimeoutSeconds field. If this field's value is greater than zero, the controller transitions the GCRegistration's phase to WaitForReuse, making the Pod available for a new client. If ReutilizeTimeoutSeconds is zero, or if a Pod in the WaitForReuse phase is not claimed before this second timeout expires, the controller deletes the Pod. Alternatively, if the GCRegistration was created because a client explicitly left the session (i.e., was deleted from the Session `.spec`), the diagram shows that the initial WaitForReconnection phase is skipped entirely, and the Pod moves directly into the reuse or deletion logic. Finally, if both timeout fields are set to zero, this waiting behaviour is disabled and the Pod is deleted immediately.

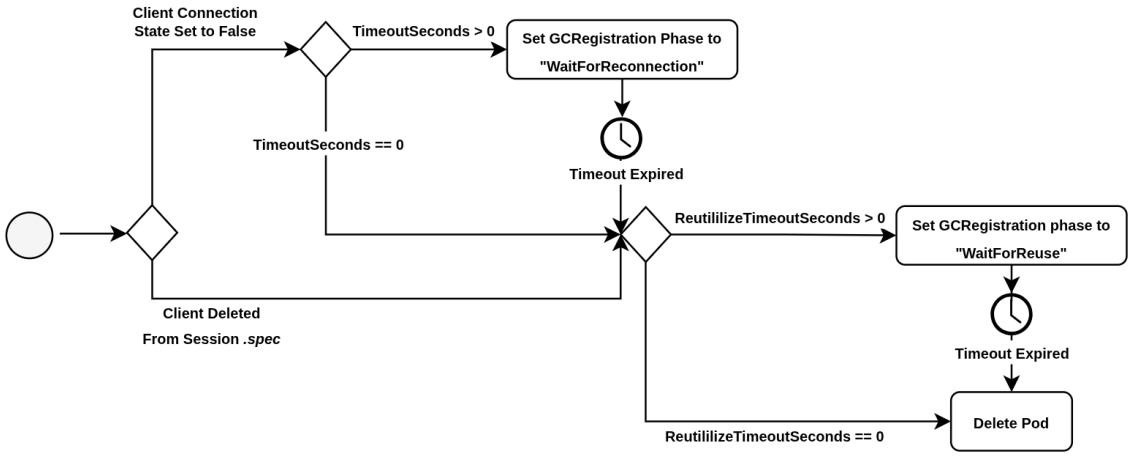


Figure 4.4: Logical process for deleting idle Pods. The controller evaluates two distinct scenarios (a lost client connection versus an explicit client departure) to determine whether to enter the WaitForReconnection and WaitForReuse grace periods before the Pod is ultimately deleted.

4.2 Session Manager

The Session Manager is an HTTP API, developed in Go with the Gin Web Framework [106], that serves as a bridge between multiple infrastructure nodes. Its primary role is to abstract several Kubernetes clusters into a single entry point, through which the application runtime can interact and request compute resources for users in specific locations. This interface translates telepresence domain operations (e.g., creating a session, a user joining a session) into Kubernetes API requests, enabling application developers to embed infrastructure logic directly into their applications. Session domain controllers, deployed on each Kubernetes cluster, provide support for the Session resource across multiple infrastructure locations.

This setup allows the Session Manager to relay the reconciliation workload from its interface to the corresponding cluster's Kubernetes API, thereby establishing support for a multi-cluster environment that can span cloud and edge compute resources.

This component manages the framework's core configuration, accepting configurable cluster credentials for accessing other clusters and session templates (blueprints for creating Session resources). Cluster credentials are stored using the standard kubeconfig format [107], as also employed by the kubectl command-line tool [65]. During deployment of the framework's components, these credentials are written into a Kubernetes Secret [108], an object designed to securely store sensitive information. Similarly, session templates configuration is stored in a Kubernetes ConfigMap [109], a type intented to hold non-sensitive key-value data. Both these configuration objects are mounted as volumes when the Session Manager Pod starts, making the configuration readily available at runtime.

Session templates are based on the same schema as the Session resource, with the exception of the `.objectMeta` and `.status` fields, which are generated during runtime. This design allows the session template data structure to be easily mapped into a Session resource, which the Session Manager then delivers to the Kubernetes API. Each configured session template must have a unique name for unambiguous identification. When the application runtime requests a new session, the Session Manager scaffolds a new Session resource from the corresponding configured template. It then generates the `.objectMeta`, as all resources within a Kubernetes cluster must include a `.objectMeta.name` and a `.objectMeta.namespace`. A generated UUID is assigned as the name, and 'default' is configured as the namespace. Once constructed, the Session resource is applied to the configured Kubernetes clusters. Fig. 4.5 visually demonstrates this behaviour in a scenario with two configured clusters. The application runtime initiates this process by requesting the Session Manager to create a new session using the template named 'template1'. The Session Manager then scaffolds the Session resource, first verifying that the requested template exists in the configuration and then generating a UUID to serve as the Session name. Once prepared, this resource is replicated across all configured clusters.

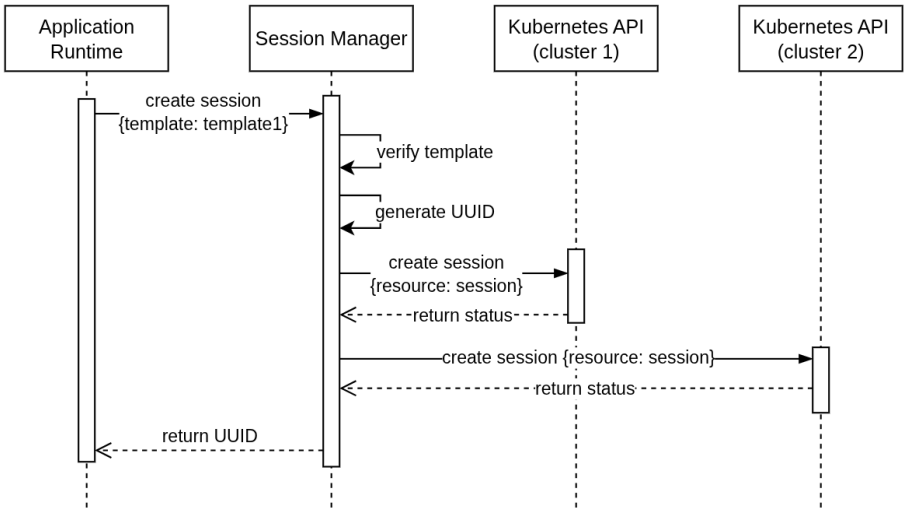


Figure 4.5: Sequence diagram illustrating the creation of a telepresence session through the Session Manager.

This component also provides an endpoint allowing the application runtime to query available infrastructure locations for resource allocation. This query returns a list of locations along

with their respective Ping Servers, enabling clients to benchmark latencies and select the most suitable cluster. Fig. 4.6 demonstrates this in practice as a client pretends to enter a session. For the presented scenario it's assumed that 'cluster2' is the most suitable cluster to run the client's workloads. Once a location is chosen, the application runtime requests the Session Manager to create a new client within a specified Session on the selected cluster. The Session Manager then queries all configured clusters to build an aggregated Session result and verify whether any compute resources (Pods) are already allocated to the client within that Session. It then communicates with the Kubernetes API of the selected cluster and updates the corresponding Session resource. When the Session resource is updated with the new client, the cluster's controller detects this change and reconciles the resource, adjusting the workload as needed.

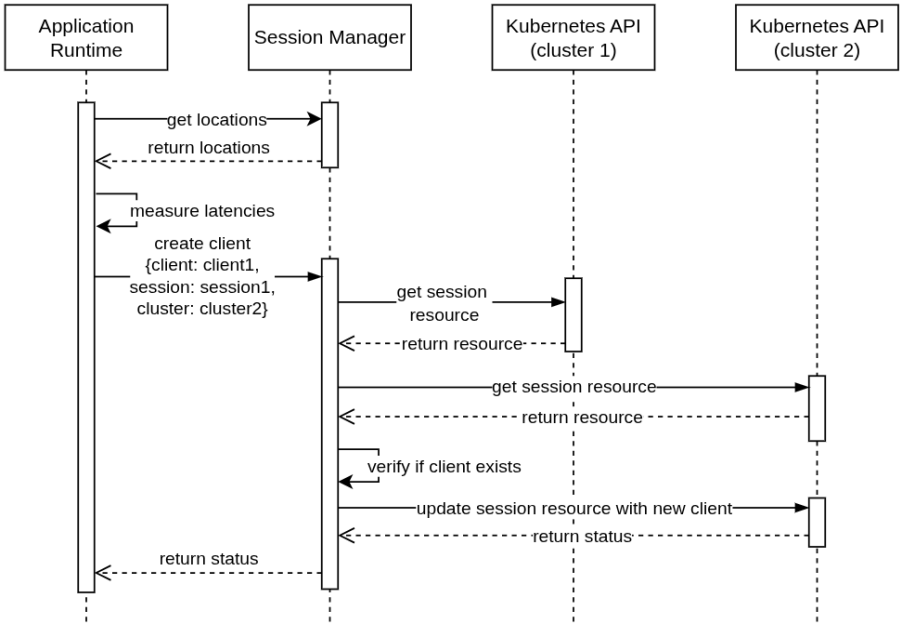


Figure 4.6: Sequence diagram illustrating the interaction between the Application Runtime and the Session Manager for selecting an optimal cluster and registering a new client into a specific session.

As the system scales with an increasing number of clusters and clients, the operational behaviour of the Session Manager can introduce significant challenges, including resource synchronization issues and the potential for overloading edge clusters. For example, because the Session Manager must replicate a new Session's resource across all configured clusters, there is a possibility that some clusters may fail during this operation. Therefore, it is essential to ensure that the system can detect and recover from partial failures to maintain a consistent view of the resource across all clusters. Furthermore, as reviewed in the state of the art, edge infrastructure offers considerably more limited compute resources compared to cloud data centres. Consequently, allocating clients based solely on minimizing latency can lead to overloaded clusters, as many clients might be directed to the same cluster without any upper capacity limit.

Analysing Fig. 4.6, one can also argue that the system may suffer from concurrency issues. The process of aggregating results from multiple clusters may lead to updates being applied to an outdated resource. In the case of two concurrent threads attempting to create the same client within the same cluster, Kubernetes' optimistic locking ensures consistency, and

no concurrency issues would arise. However, this guarantee does not extend to the case where two concurrent threads attempt to create the same client in two distinct clusters. Addressing these challenges lies outside the scope of the current work but is identified as an area for future development.

4.3 STUNner: WebRTC Gateway for Kubernetes

The state of the art review identified WebRTC as a protocol used for real-time communication between clients and their offloaded functions [9, 21]. Following this precedent, WebRTC was also adopted during the evaluation of the framework, presented in the next chapter. However, during the implementation stage, it was noted that deploying WebRTC servers directly within a Kubernetes cluster is not straightforward due to NAT issues. To address this, this section explores the integration of a third-party technology: STUNner, a WebRTC gateway for Kubernetes. It is important to note that STUNner is not considered an integral component of the solution. Rather, it is presented here as a complementary and compatible tool that enables the deployment of WebRTC-based workloads within the framework's ecosystem.

WebRTC is a protocol that enables real-time, peer-to-peer media exchange between devices [110]. Establishing this direct connection requires a signalling server to orchestrate how the peers connect. This signalling process involves the two peers exchanging a message in the standardized Session Description Protocol (SDP) format [111], which serves as a discovery mechanism for establishing the final peer-to-peer link. An SDP message describes the connection's media properties, such as supported media types (e.g., audio, video, or data channels) and codecs. Crucially, it also contains a list of network candidates gathered by the Interactive Connectivity Establishment (ICE) framework [112]. These candidates represent potential IP addresses and ports where a peer can receive media. Determining these network addresses is particularly challenging within a Kubernetes environment.

STUNner Gathering ICE candidates without a NAT-traversal mechanism is not possible for servers deployed inside Kubernetes clusters. Network traffic targeting a Pod is subjected to several rounds of NAT, as packets must traverse cloud load-balancers and the node's network to finally arrive at the Pod's private network address [95]. For this reason, the state-of-the-art approach for deploying WebRTC servers in Kubernetes is often to configure the Pods with `hostNetwork=true` [95]. This setting makes the Pod share the network namespace of its host node, thereby inheriting the node's public IP address. However, not only is this approach recognized as an anti-pattern due to security and scalability issues, but it also fails to solve the problem completely, as external clients still require NAT-traversal facilities (i.e., Session Traversal Utilities for NAT (STUN) [112] and Traversal Using Relays around NAT (TURN) [112] servers) to establish a connection [95]. STUNner is a gateway designed to ingest WebRTC media traffic into a Kubernetes cluster by providing an integrated, public-facing STUN/TURN server [113]. The primary advantage of this approach is that all NAT-traversal facilities are located directly within the cluster [113]. This allows both the server-side Pods and external clients to share the same computes resources, and avoids the security risks of hostNetwork and the cost of third-party STUN/TURN services [113]. This co-location also addresses a major concern when TURN is used as fallback. When a peer-to-peer connection fails and communication falls back to a TURN server, latency can become a bottleneck if that server is geographically distant. With STUNner, the relay server is co-located with the application Pods, ensuring round-trip time is minimized. Furthermore, because STUNner

acts as a gateway for all WebRTC packets, it consolidates all traffic into a single exposed UDP port [113].

Chapter 5

Evaluation

To evaluate the proposed system design, a series of experiments were created to collect relevant performance metrics under various conditions. The evaluation is primarily guided by two questions:

a) How does the framework perform in terms of compute resource utilization (CPU and memory) and key telepresence metrics (connection and recovery latency) when compared to conventional baseline implementations?

b) How do the framework's connection and recovery latencies behave when the workload is distributed across multi-cluster environments?

To answer these evaluation questions, a strategy was designed that addressed three main challenges: (1) generating realistic, session-based user load; (2) designing appropriate baseline scenarios for a comparative analysis; and (3) implementing a representative XR telepresence workload.

First, creating a realistic load profile was non-trivial due to the session-based nature of the domain. Load can be modelled as users joining a single session or as users distributed across many concurrent sessions. This work primarily focused on the latter, as simulating a multi-session, multi-user environment provides a more realistic test of the framework's capabilities. This required the stochastic generation of user and session events to simulate a dynamic, multi-session load. Second, to provide a basis for comparison, two baseline cases were modelled using mostly conventional Kubernetes primitives. These cases served as benchmarks against which the performance of the proposed framework could be measured. Third, to ensure the evaluation was representative of a real-world scenario, an application service was implemented. This service simulates an object-detection pipeline, ensuring that deployed Pods perform meaningful work rather than remaining idle during trace execution. The following sections detail how each of these challenges was addressed and present the results of the subsequent comparative analysis and benchmarking.

5.1 Trace Generation and Execution

To experimentally evaluate and quantify the system, this work utilized two methods for generating user load: synthetic traces and a real-world trace derived from a multi-university poster session [27]. In this context, a trace represents a series of timed events corresponding to interactions with the system. Applying these traces to a set of configurations enabled the collection of performance metrics under load, facilitating a comparative analysis and benchmarking of the proposed framework. The event model comprised five distinct interactions: the creation and deletion of a session, the creation and deletion of a client within a session,

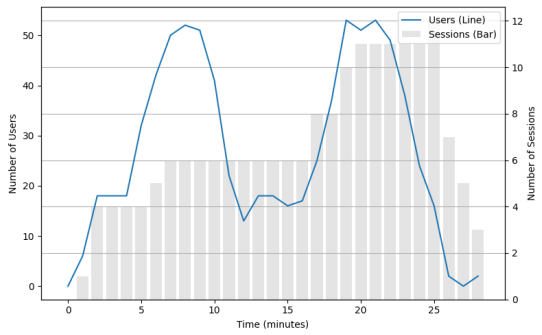
and the deletion of a Pod. While the first four events map directly to operations provided by the *Session Manager API*. The final event, deleting a Pod, was included to simulate Pod failures and measure the time required for a client's workload to recover.

The synthetic traces were generated using Poisson Processes [114], a stochastic model suitable for describing random events occurring independently over time. The real-world trace was constructed by parsing and extracting event data from the logs of a multi-university poster session [27]. Figure 5.1 illustrates these traces, with sub-figures 5.1a, 5.1b, 5.1c and 5.1d showing the synthetically generated traces and sub-figure 5.1e showing the real-world trace from the poster session. A notable difference between them is their scope. The synthetic traces model a dynamic, multi-session environment, showing fluctuations in both the number of concurrent users and the number of active sessions. In contrast, the real-world trace (Figure 5.1e) represents the user load within a single, longer-running session. As such, it consists of a single "create session" event, followed only by events corresponding to clients joining, leaving, or Pods being deleted within that one session. While this trace does not evaluate the framework's performance across multiple concurrent sessions, it provides a valuable complement to the synthetic data by representing a non-simulated user load pattern.

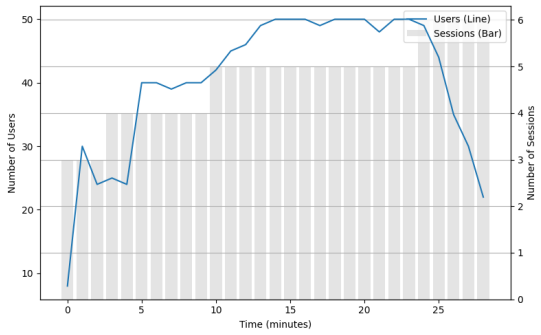
The synthetic traces were generated using a stochastic model based on a Non-Homogeneous Poisson Process (NHPP). This approach is frequently used to model event arrivals in various queuing systems, from customer arrivals at service stations like banks or ATMs to network packet arrivals in telecommunications [114]. In this work, the process is analogous to simulating sessions being created and users subsequently entering or leaving them. Unlike a standard (or homogeneous) Poisson process where the rate λ of events is constant, an NHPP features a time-dependent rate, denoted as $\lambda(t)$. A key aspect of this model is that each event type is governed by its own distinct $\lambda(t)$ function. This allows the model to generate traces with varying intensity over time, more closely mimicking real-world usage patterns. For example, the generation script was configured to create 30-minute traces with event rates to produce distinct spikes in client load.

From the synthetically generated set, four traces were chosen with the aim of creating a balanced evaluation. The selection was made to include two traces expected to highlight the framework's strengths and two hypothesized to be less favourable. This hypothesis is based on a key limitation observed in the comparative scenarios (detailed in the next section): their inability to de-provision resources when clients leave a session. Therefore, it is anticipated that traces featuring significant periods of client departure (Figures 5.1a and 5.1c) will showcase the framework's superior efficiency. Conversely, traces 5.1b and 5.1d are expected to show a less pronounced performance difference when compared to the alternative models.

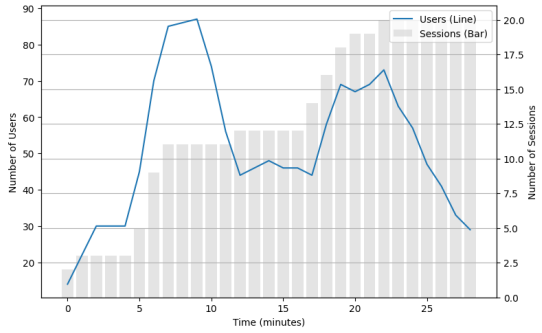
A key architectural feature of this model is the decoupling of the trace generation and load execution phases. First, a Python [115] script generates the timed events and writes them to a trace file. A separate script then reads this file and executes the events against the compute infrastructure, which ensures a consistent and reproducible load is applied across all experimental cases. During trace generation, the model uses a dynamic data structure to track active sessions. This allows it to randomly select a valid session target for events like creating or deleting a client. A similar random selection is used for events that delete entire sessions. In contrast, to simulate unpredictable infrastructure failures, Pod deletion is handled dynamically at runtime. Instead of being predetermined in the trace file, the specific



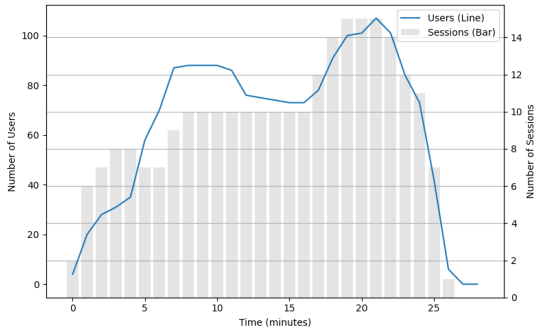
(a) Load variance trace



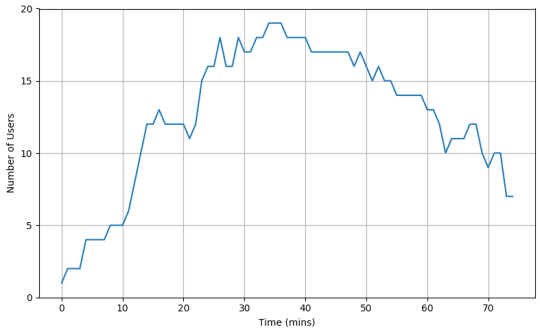
(b) Sustained load trace



(c) Load variance trace



(d) Sustained load trace



(e) Real trace from a multi-university poster session

Figure 5.1: A collection of traces used to simulate client load for the evaluation. Sub-figures (a) through (d) illustrate the synthetic traces for multi-session scenarios, while sub-figure (e) corresponds to the single-session trace from a real-world event.

Pod to be terminated is chosen by the execution script, which queries the Kubernetes API server for a list of available Pods and then randomly selects one.

5.2 Experimental Setup

The previous traces were executed using a one-pod-per-client configuration, where each client connecting to a session is allocated a dedicated Pod. To simulate a realistic workload, a Python-based offloading server was developed. This server mimics an object detection pipeline by receiving a continuous stream of image frames and returning a simulated data stream of bounding boxes and object identifiers.

Following the approach in [9, 21], WebRTC was implemented as the communication protocol between clients and their respective remote Pod, with STUNner configured as media gateway. The WebRTC peer-to-peer connection workflow [116] begins with the offering peer (the client) using the ICE framework to gather its network address candidates. These candidates are then bundled into an SDP offer and sent to the other peer (the remote Pod) via its signalling interface. The Python offloading server provides the signalling interface as an HTTP endpoint. The remote Pod, in turn, receives the offer, generates its own ICE candidates, and returns them within an SDP answer in the HTTP response body. With both peers now possessing the other's SDP and a list of potential network paths, the ICE framework begins testing connectivity on the various candidate pairs. Once a viable path is found, the direct peer-to-peer media connection is established.

Following are the distinct architectural cases evaluated in the comparative analysis presented in Section 5.3. Each case represents a different Kubernetes configuration for deploying the WebRTC object-detection workload within the context of an XR telepresence scenario.

Case 1: Baseline Deployment This case models a conventional approach to hosting an XR telepresence application on Kubernetes, representing a vanilla deployment. For this scenario, the cluster was configured with a single Kubernetes Deployment. The number of replica Pods was predefined to a static value sufficient to cover the maximum concurrent user load required by the evaluation trace. This setup intentionally lacks any session-awareness or infrastructure elasticity.

All Pods were exposed behind a single Service to handle the initial WebRTC signalling requests. As established in Section 4.1.3, using a single Service for multiple stateful Pods typically creates a reachability problem for continuous data streams. However, this issue is partially mitigated here because the signalling itself is a discrete HTTP request. Once the peer-to-peer connection is brokered via signalling, the subsequent media stream is managed directly by the STUNner gateway, bypassing the Service.

Despite its viability, this native approach has two significant drawbacks. First, because the Service distributes signalling requests randomly across all Pods, a client may attempt to connect to a Pod that is already occupied by another user. Since each application Pod is configured to serve only one client (one-pod-per-client), this can lead to failed connection attempts. Second, the static number of replicas provides no compute elasticity. Although the number of Pods in this case was provisioned to handle the peak load of the evaluation traces, this static approach is inherently inefficient. Resources are not scaled down during low-traffic periods, leading to significant resource waste. Moreover, in a real-world production scenario where user load could exceed the tested peak, this model would be unable to scale up, resulting in insufficient capacity.

Case 2: Session-Scoped Deployment Progressing towards a more elaborate approach, this case introduces a more advanced and dynamic architecture by creating a dedicated Kubernetes Deployment for each session. When a new session is required, a corresponding Deployment is created with zero replicas; as clients subsequently join the session, the number of replicas is scaled up accordingly. To handle network reachability, this case adopts the *service-per-pod* pattern, similar to the proposed framework's architecture. This pattern is implemented using a custom Python script that acts as a "pseudo-controller". The script watches for new Pod creation and, in response, automatically provisions a dedicated Service and updates the Ingress resource with the new signalling endpoint. This pseudo-controller is

intentionally simplistic and does not adhere to the standard design patterns of a Kubernetes controller. While this component could be exploited for it's lack of robustness behaviour in the presence of missed intermediate state changes, we purely utilize it to enable the necessary networking for this specific evaluation case.

Despite these improvements, this model has several significant drawbacks. The first is an inefficient client connection process. Although a new Pod is created when a client requests to join, the client has no direct way of knowing which Pod was created for it. The client must therefore iterate through all Pods managed by the Deployment, attempting to signal each one until it finds an available instance. This process is inefficient and introduces unnecessary load on already-occupied Pods. Second, the design is prone to concurrency issues. When multiple clients join a session simultaneously, they all attempt to update the same Deployment resource to scale its replicas. This can lead to optimistic locking conflicts and failed API requests, creating unnecessary load on the Kubernetes API server. Finally, this model does not solve the scale-down problem. Since a Deployment treats its Pods as interchangeable, fungible replicas, it is not possible to selectively delete the specific Pod that has become idle after a client leaves. Deleting a replica is a non-deterministic process that could terminate a Pod serving an active client. For this reason, Pods are never scaled down in this case; they are only removed when the entire session and its corresponding Deployment are deleted.

Case 3: Proposed Framework This final case evaluates the complete framework proposed in this work, using the same Python-based offloading server to represent the WebRTC object-detection workload. The deployment follows the architecture detailed in Chapter 3 and demonstrates the synergistic cooperation between the framework's custom controllers and the STUNner gateway. In this model, responsibilities are clearly divided: the STUNner gateway coordinates the WebRTC media streams, while the signalling process is managed directly by the framework's components. Unlike the previous cases where signalling could be inefficient or unreliable, this approach ensures that each client communicates directly with its own dedicated Pod.

The client discovers its dedicated endpoint by watching the `.status` field of the Session resource. Once the *Session Controller* and *Network Controller* have provisioned the Pod and its associated Service, they update the status mapping with the client's entry and its corresponding signalling endpoint. The client then sends its HTTP signalling requests directly to this address, guaranteeing a conflict-free connection to its specifically allocated Pod.

5.3 Comparative Analysis

The comparative analysis was performed by subjecting each of the three previously illustrated architectural cases to the load traces presented in Section 5.1. For each test run, key performance metrics were collected and compared across the configurations. Two categories of metrics were measured. The first included standard compute resources, such as CPU and RAM usage. The second category focused on metrics specific to the telepresence use case: Connection Latency, defined as the time from a client's resource request to the successful establishment of the WebRTC connection, and Recovery Latency, the time required for a client to reconnect after its associated Pod fails and is restarted.

The traces were injected into the Kubernetes test-bed from a separate client machine running Fedora Linux 40 (Workstation Edition), equipped with an Intel Core i5-10300H CPU (4 cores) and 16 GB of RAM. For this comparative analysis, a single Kubernetes cluster was utilized for each test case. This approach was chosen because both Case 1 and Case 2 do not support multi-cluster environments, making a single-cluster setup the only fair basis for comparison. The cluster was hosted on Microsoft Azure[1] using its Azure Kubernetes Service (AKS)[2] offering. It ran Kubernetes version 1.31.8 and was deployed in the Spain Central region, configured with two nodes: one with 4 vCPUs and 8 GB of memory, and the other with 2 vCPUs and 7 GB of memory.

5.3.1 Memory and CPU

Memory consumption data was collected using Azure's built-in metrics feature, which exported memory usage for the duration of each trace execution. The results for each individual trace are illustrated in Figures 5.2a, 5.2b, 5.2c, 5.2d and 5.2e. Each of these figures provides a direct comparison of the memory consumption across the three architectural cases over the trace period. To provide a consolidated overview, Figure 5.2f presents a Complementary Cumulative Distribution Function (CCDF). This plot summarizes the memory usage difference between the *Proposed Framework* (Case 3) and the other two cases (Case 1 and 2).

As previously mentioned, the trace selection included two scenarios expected to highlight the framework's strengths and two hypothesized to be less favourable. This is reflected in Figures 5.2a and 5.2c, which correspond to traces with significant periods of client departure (Figures 5.1a and 5.1c). During these periods of declining user load, the *Proposed Framework* consumes significantly less memory than both the *Baseline Deployment* and the *Session-Scoped Deployment*. This clear advantage stems from its garbage collection mechanism, which successfully de-provisions compute resources as users leave. In contrast, the other two cases lack this capability and therefore maintain their peak memory allocation regardless of the actual number of active clients.

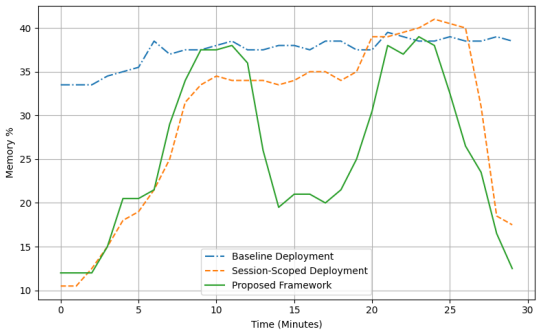
Conversely, this performance difference is less pronounced in traces where the user load remains high without significant periods of decrease (Figures 5.1b and 5.1d). The corresponding memory consumption for these scenarios is shown in Figures 5.2b and 5.2d. In these cases, the memory usage of the *Proposed Framework* and the *Session-Scoped Deployment* are nearly identical. This is because both models scale up resources reactively as users join, and since there is no significant drop in user load, the garbage collection mechanism of the *Proposed Framework* has fewer opportunities to de-provision resources.

As expected, memory usage for the *Baseline Deployment* across all traces remains relatively constant. This behaviour reflects its static, pre-provisioned design, where the number of Pods is fixed to handle the peak load. The slight fluctuations observed are not due to infrastructure scaling but are caused by the operational load from clients into these Pods. This memory impact is marginal, as the Pods only return a simulated object detection result rather than performing a computationally expensive analysis.

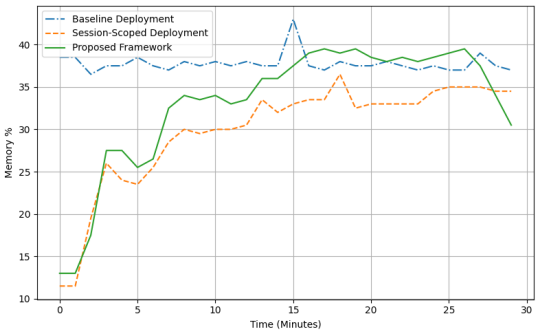
Additionally, an analysis of the initial periods of trace execution reveals a slight memory overhead for the *Proposed Framework* when compared to the *Session-Scoped Deployment*.

[1] Microsoft Azure: `https://azure.microsoft.com/en-us/` (Accessed 28-06-2025)

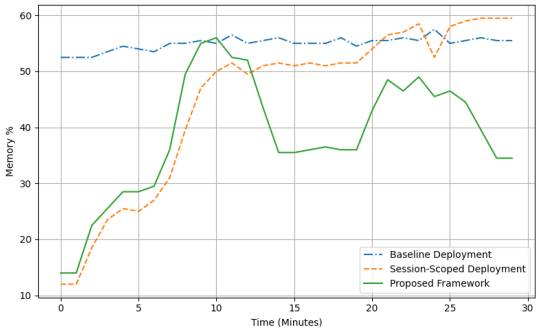
[2] Azure Kubernetes Service: `https://azure.microsoft.com/en-us/products/kubernetes-service` (Accessed 28-06-2025)



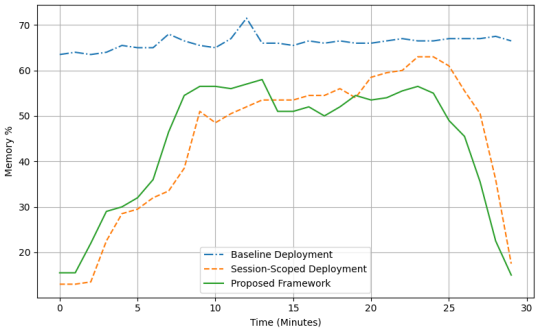
(a) Memory utilization for trace in Figure 5.1a



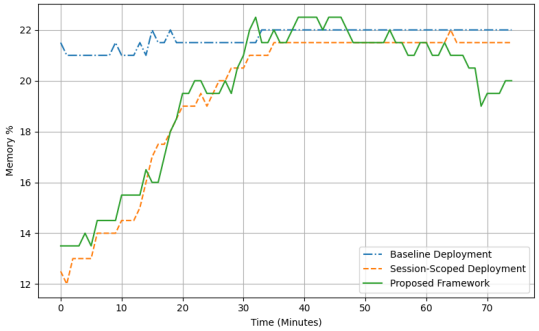
(b) Memory utilization for trace in Figure 5.1b



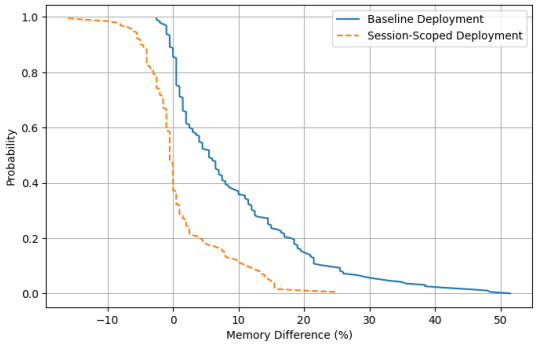
(c) Memory utilization for trace in Figure 5.1c



(d) Memory utilization for trace in Figure 5.1d



(e) Memory utilization for trace in Figure 5.1e



(f) Memory difference Complementary CDF

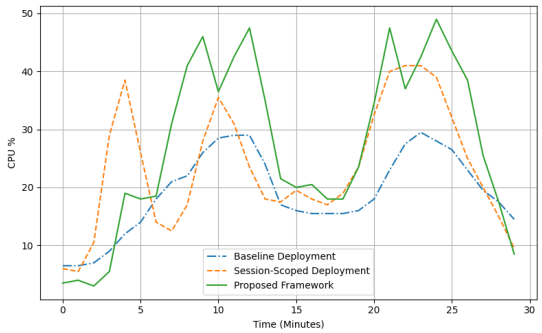
Figure 5.2: A comparison of memory utilization for the three comparative cases across the five evaluation traces. Sub-figures (a) through (e) show the time-series data for each trace, while (f) presents a consolidated CCDF of the memory usage difference between the Proposed Framework and the remaining cases.

This overhead can be attributed to two main factors. First, the framework's custom controllers introduce additional workloads that the cluster must execute. Second, the *Proposed Framework* uses a greater number of API resources (such as Session, GCRegistration and Pod resources), which are stored in etcd and contribute to higher memory usage.

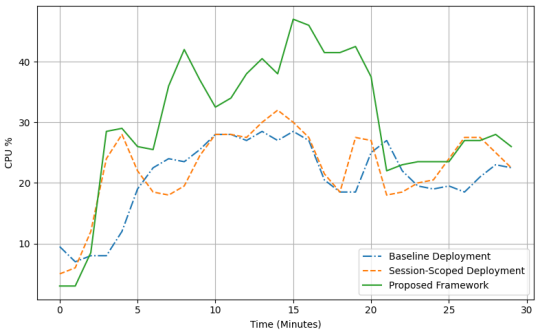
The CCDF in Figure 5.2f consolidates these findings by plotting the memory usage difference between the *Proposed Framework* and the other two cases. The plot shows that the memory difference relative to the *Baseline Deployment* is positive approximately 80% of the time, quantitatively confirming that the *Proposed Framework* is significantly more memory-efficient. In contrast, the comparison with the *Session-Scoped Deployment* highlights a key trade-off. The plot indicates that the *Proposed Framework* consumes less memory for

about 40% of the time, achieving a memory reduction of up to 25%. For the remaining 60% of the time, the memory difference is negative, signifying a memory overhead of up to approximately 15%.

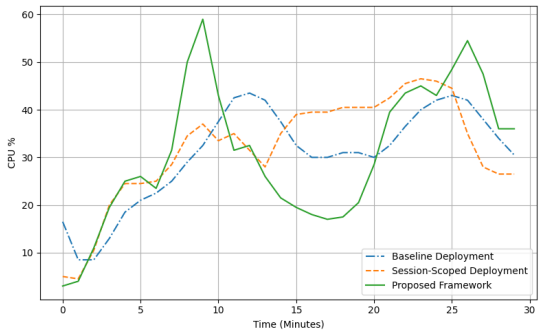
Following the same approach as with memory, CPU consumption data was collected using Azure's built-in metrics feature. The results for each individual trace are illustrated in Figures 5.3a, 5.3b, 5.3c, 5.3d and 5.3e. To provide a consolidated overview, Figure 5.3f plots the CCDF of the CPU usage difference between the *Proposed Framework* and the other two cases.



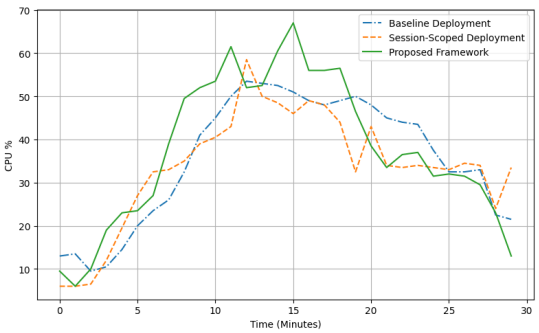
(a) CPU utilization for trace in Figure 5.1a



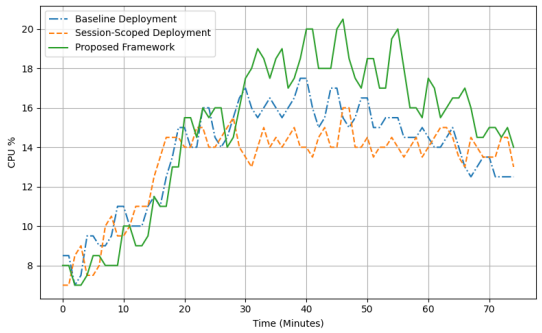
(b) CPU utilization for trace in Figure 5.1b



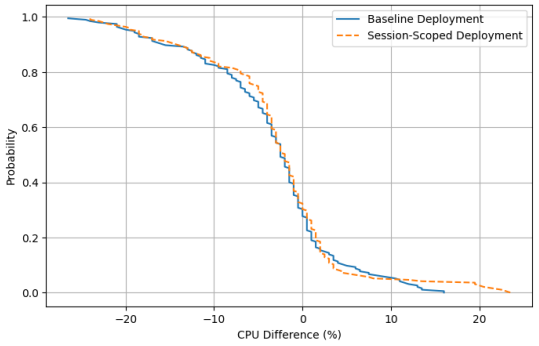
(c) CPU utilization for trace in Figure 5.1c



(d) CPU utilization for trace in Figure 5.1d



(e) CPU utilization for trace in Figure 5.1e



(f) CPU difference Complementary CDF

Figure 5.3: A comparison of CPU utilization for the three comparative cases across the five evaluation traces. Sub-figures (a) through (e) show the time-series data for each trace, while (f) presents a consolidated CCDF of the CPU usage difference between the Proposed Framework and the remaining cases.

As anticipated, an analysis of the collected data reveals that the *Proposed Framework* introduces a CPU consumption overhead compared to the other models. This is particularly

noticeable during periods of high client activity; as more clients join sessions, the CPU usage of the *Proposed Framework* spikes, surpassing the other cases. In both Figures 5.3a and 5.3c, for example, the framework's CPU consumption rises sharply during time periods that correspond to a high rate of client arrival in the respective traces (Figures 5.1a and 5.1c). This overhead arises from the active reconciliation performed by the framework's custom controllers, which must process Session resource updates and launch new Pod workloads to serve incoming clients. A second factor contributing to this overhead, which is explored further in the next section, is the relationship between connection latency and effective system load. Because clients using the *Proposed Framework* connect more rapidly, they spend a greater portion of the trace's duration actively generating a workload. This higher cumulative load, in addition to the controller's reconciliation work, helps to explain the increased CPU consumption when compared to the other two cases, where clients spend more time idle while waiting to connect.

5.3.2 Connection and Recovery Latency

Connection latency was measured as the time elapsed from the start of processing for a client creation event until the successful establishment of the WebRTC connection with its designated Pod. Figure 5.4 presents the Cumulative Distribution Function (CDF) of this metric, comparing the performance of the three experimental cases.

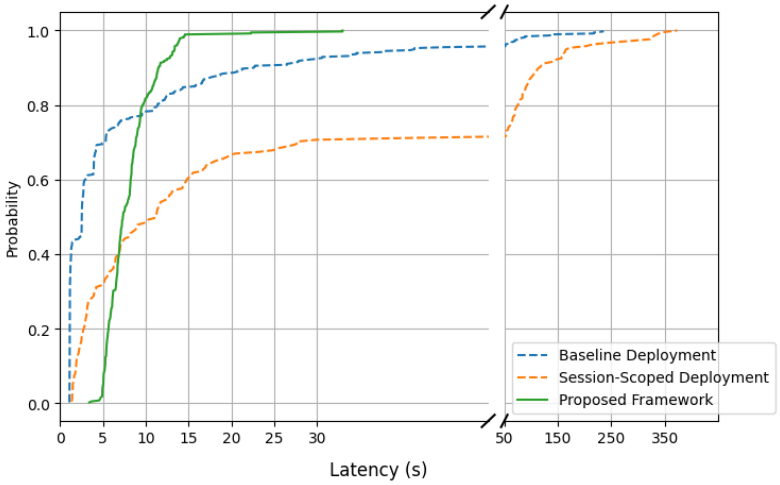


Figure 5.4: A CDF of the connection latency for the three comparative cases. The plot shows that the Proposed Framework exhibits a much narrower distribution of latencies, while the other two cases exhibit a wider, less predictable distribution of latencies.

The *Proposed Framework* demonstrates the most consistent and performant behaviour. The data shows that 80% of its connection latencies are below 10 seconds, with the maximum peaking at 32 seconds. In contrast, both the *Baseline Deployment* and *Session-Scoped Deployment* exhibit a much more dispersed range of latencies, with some connections taking several minutes to establish. This variance is a direct result of the non-deterministic connection behaviour inherent in both these cases where a client's signalling request can be routed to any Pod in the available set. This can lead to one of two extremes: either a very low latency connection, if the client is immediately routed to an available, already-running Pod, or a very high latency connection, if the client must repeatedly attempt to signal different Pods before an available one is found.

It is important to note that both the *Baseline Deployment* and *Session-Scoped Deployment* can achieve the lowest-latency connections observed in the CDF plot. This occurs in "best-case" scenarios that are unavailable to the *Proposed Framework* in this particular test configuration. In the *Baseline Deployment*, all Pods are pre-provisioned, meaning clients can sometimes connect instantly without waiting for a Pod to spin up. Similarly, in the *Session-Scoped Deployment*, a new client may be immediately routed to an idle, already-running Pod from a previous client. In contrast, the *Proposed Framework* in this test always provisions a new Pod on demand, which introduces a predictable startup latency. However, as will be demonstrated in Section 5.4, this initial connection latency can be significantly optimized by configuring a Pod reuse policy (ReutilizeTimeoutSeconds > 0), allowing this configuration to also benefit from connecting clients to already-running Pods.

The prolonged connection times observed in the *Baseline Deployment* and *Session-Scoped Deployment* also have a direct impact on the effective load exerted on the cluster. While a client is attempting to connect, it is not yet generating its workload (i.e., the object-detection stream). Because clients using the *Proposed Framework* connect more rapidly, they begin generating this workload sooner. In contrast, for the other two cases, the long connection periods mean that at any given moment, fewer clients are actively running their workloads compared to the *Proposed Framework*. In addition to the controller overhead, this also helps justify the higher CPU usage observed earlier, as it is serving more active clients for a greater portion of the time.

Figure 5.5 presents the CDF of recovery latency, defined as the time a user takes to reconnect to their workload after its Pod restarts due to a transient failure. For the same reasons discussed in the connection latency analysis, the recovery process for both the *Baseline Deployment* and *Session-Scoped Deployment* is inefficient; a client must re-initiate the entire non-deterministic discovery and signalling process to find its new Pod, which is identical to making a connection from scratch. This performance gap is quantified in the plot. The *Proposed Framework* consistently achieves recovery in under 40 seconds, whereas for the other two cases, approximately 30% of recovery attempts experience latencies greater than 40 seconds, with some extending up to 400 seconds.

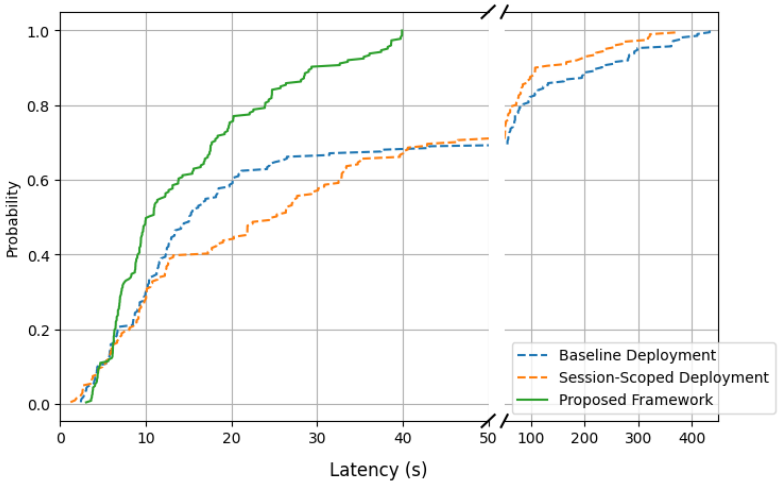
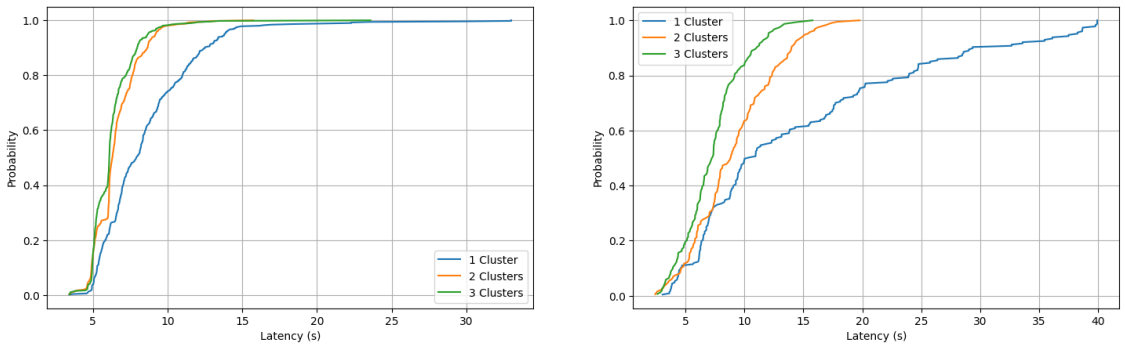


Figure 5.5: A CDF of the recovery latency simulated with Pod failures. The plot highlights the efficiency of the Proposed Framework's deterministic recovery process compared to the higher, more dispersed latencies of the other two cases.

5.4 Benchmarking

While the previous section analysed the framework's performance in a single Kubernetes cluster, this section benchmarks key telepresence metrics in a multi-cluster environment. The methodology is similar, executing the same load traces across a distributed infrastructure. For these tests, events such as client creation and Pod deletion were distributed across the available clusters in a round-robin manner. To ensure a fair comparison, all clusters were deployed in the same physical location (the Azure Spain Central region) to normalize network latency. The test-bed configuration for each cluster was the same to the one previously described, with the only difference being the number of clusters (one, two, or three). This setup splits the total load, which was previously directed at a single cluster, across the configured number of clusters.

Figures 5.6a and 5.6b illustrate the connection and recovery latencies, respectively, for the one, two, and three cluster configurations. As expected, distributing the client load across multiple clusters improves both connection and recovery latencies. This is because the reconciliation workload is shared among multiple sets of controllers, reducing the number of concurrent requests each control plane must process. An analysis of the connection latency CDF (Figure 5.6a) reveals that the performance gain from two to three clusters is minimal. This suggests that the primary bottleneck is not the control plane but rather the inherent "cold start" time required to spin up a new Pod, a factor that cannot be reduced further simply by adding more clusters. In contrast, the recovery latency CDF (Figure 5.6b) shows a more pronounced improvement as clusters are added. For the three-cluster setup, 80% of recovery latencies are below 10 seconds, while for the two-cluster setup, only 60% of recoveries fall under this same 10-second mark. We theorize this is because when a Pod is deleted and then immediately replaced, the scheduler can utilize cached state information, turning recovery into a task that scales with the addition of more clusters.



(a) Connection latency CDF

(b) Recovery latency CDF

Figure 5.6: CDFs of the connection and recovery latency, benchmarking the performance of the proposed framework with one, two, and three-cluster configurations.

To minimize connection latency, the framework's Pod reuse policy can be configured via the ReutilizeTimeoutSeconds parameter. Figure 5.7 illustrate the trade-off between this timeout value and the resulting system compute resource usage. An analysis of the box plots reveals that while increasing the reuse timeout does not significantly lower the median connection latency, it effectively reduces the latency for the slowest connections. As the timeout increases from 0 to 40 seconds, the upper quartiles show a clear decreasing trend,

with the 75th percentile dropping below the 10-second mark. The minimum observed latencies also decrease in the initial range, with the lower whisker dropping from approximately 4 seconds to 2 seconds as the timeout is increased from 0 to 20 seconds. However, further increases beyond a 40-second timeout yield diminishing returns, with the latency distribution remaining relatively stable. This comes at the cost of increased memory and CPU utilization, as illustrated by the line plots. For example, as the ReutilizeTimeoutSeconds increases from 0 to 60 seconds, the memory consumption steadily rises from approximately 35% to 38%. CPU utilization shows a similar, though less uniform, upward trend from roughly 32% to over 36%, confirming the direct relationship between longer Pod reuse times and higher sustained resource usage.

It is important to note that, in the current implementation, this Pod reuse policy is only available for clients joining the same session. Expanding this feature to allow Pods to be shared between different Session resources that use the same template could be a beneficial area for future work.

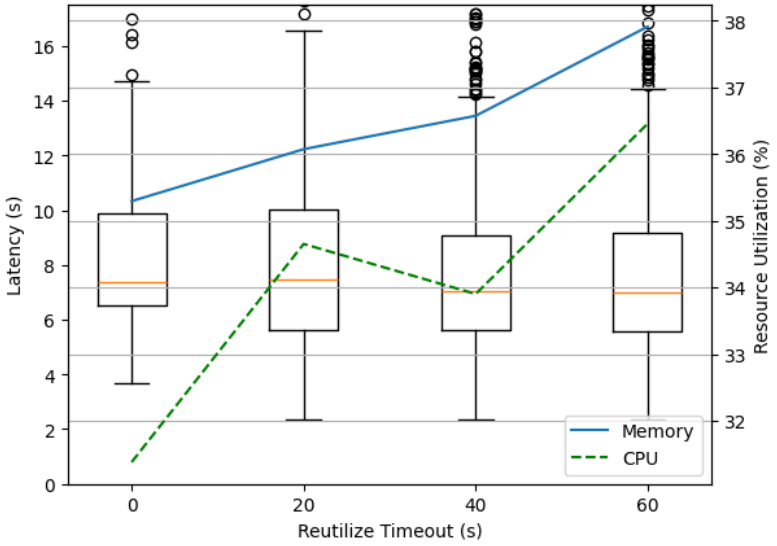


Figure 5.7: The trade-off between Pod reuse timeout, connection latency, and resource utilization. Increasing the timeout reduces the latency values at the cost of higher memory and CPU usage.

5.5 Discussion of Results

In the beginning of this chapter we raised two questions with the purpose of guiding this evaluation. This section answers these questions by comparing the expectations with the actual results. It also expands upon the findings to propose potential architectural improvements and future research directions where the framework's trade-offs could be further optimized.

a) *How does the framework perform in terms of compute resource utilization (CPU and memory) and key telepresence metrics (connection and recovery latency) when compared to conventional baseline implementations?*

The memory utilization results demonstrate an advantage for the *Proposed Framework* (Case 3), particularly in scenarios with dynamic client loads. As anticipated, the *Baseline Deployment* (Case 1) was the least efficient due to its static nature, consistently consuming

the most memory. The *Session-Scoped Deployment* (Case 2) also proved inefficient, as its inability to scale down resources meant it could not adapt to declining user loads. In contrast, the *Proposed Framework* exhibited a slight memory overhead compared to Case 2 during periods of stable or increasing load, a reasonable trade-off for its additional reconciliation processing and custom resources it requires.

However, this memory overhead could be minimized by re-designing the Session custom resource. Currently, when multiple Session resources are created from the same template, static configuration data (e.g., PodTemplates, TimeoutSeconds) is duplicated in each resource, leading to inefficient memory usage in the etcd data store. A more efficient design would decouple this static data into a separate data structure. The Session resource would then only need to store dynamic data (i.e., the client map) and a reference to its template, significantly reducing data duplication.

In terms of CPU utilization, the *Proposed Framework* consistently consumed more resources than the baseline cases, as expected. This overhead is attributable to two factors. First is the extra processing required for the framework's three custom controllers to perform their continuous reconciliation work. Second, and more subtly, the framework's inferior connection and recovery latencies mean that clients spend more time actively connected and generating a workload, which naturally leads to higher CPU usage on the Pods.

b) *How do the framework's connection and recovery latencies behave when the workload is distributed across multi-cluster environments?*

As expected, the results from the multi-cluster benchmark demonstrate that both connection and recovery latencies decrease as the workload is distributed across more clusters. This validates the framework's horizontal scalability, as splitting the load reduces contention on each individual cluster's control plane, leading to faster reconciliation times. However, we theorize that these latencies could be optimized even further. Allowing Pod reuse between different Sessions that share the same template, for instance, could provide significant performance gains.

Furthermore, in multi-cluster deployments, a key area for future optimization is reducing the aggregate memory and CPU consumption. The current design replicates the Session resource and deploys the framework's controllers across all clusters. This approach results in duplicated Session data in each cluster's etcd store and requires that each cluster's control plane independently performs reconciliation logic. An alternative approach could leverage Liqo, therefore extending Kubernetes with cross-cluster resource synchronization via its offloading mechanism. Liqo establishes a bridge between the nodes of remote clusters, enabling the entire distributed infrastructure to behave as a single logical cluster.

This would allow the Session resource to exist and be reconciled in only one primary cluster. The resulting Pod resources, however, could still be selectively synchronized and deployed to the remaining edge clusters as needed. Such a design would eliminate redundant Session resources and controller instances across the infrastructure, thereby significantly reducing overall memory and CPU consumption. The potential of integrating Liqo into the architecture is discussed further as an area for future work in the following chapter.

Chapter 6

Conclusion and Future Work

This work addresses the challenge of deploying XR telepresence workloads on Kubernetes, highlighting the platform's limitations when handling stateful applications requiring stable network identity. XR telepresence is characterized by complex deployment architectures where application services are bound to a session's context and must scale across different geographical locations to meet low-latency requirements. Consequently, the standard workload management capabilities of Kubernetes fail to satisfy these specific demands.

To overcome these limitations, this work aims to simplify the deployment of XR telepresence systems. The proposed solution introduces mechanisms that abstract infrastructure complexity and integrate resource management directly into the application's runtime. This approach enables compute resources to be scheduled dynamically as clients join and leave sessions from the application layer. This description broadly addresses the central question of this dissertation, introduced in Section 1.2: *How can XR telepresence applications be deployed in Kubernetes?* To answer this, three specific objectives were derived, which form the core contributions of this work.

Contributing towards the first objective—*establish a state of the art on XR telepresence architectures and their deployment on Kubernetes*—a state of the art analysis was performed. The literature review led to a generic architecture characterized by session-bound application services distributed across multiple locations. However, it was identified that literature addressing XR workload deployment on Kubernetes did not cover the session-awareness required in telepresence systems. Critically, this research identified the field of game server orchestration as a powerful analogue for the problem at hand. This work drew inspiration from this domain, adopting the techniques used by these platforms to extend Kubernetes for supporting custom workloads. These insights formed the technical foundation for the solution proposed herein.

Regarding the second objective—*design and implement a framework for deploying XR telepresence on Kubernetes*—a two-phase design process was conducted: first, a high-level architecture was established to bring telepresence concepts into the Kubernetes ecosystem, followed by a more fine-grained design and implementation details of the solution's components. The proposed framework comprises two core pieces which form the basis for supporting XR telepresence workloads on Kubernetes, the *Session Controller* and the *Session Manager*. Following the Operator Pattern, the design proposes an extension to the Kubernetes API server with a CRD and a custom Kubernetes controller reconciling the state of the new resource. This teaches Kubernetes how to orchestrate the workload of telepresence applications by embedding the domain expertise into software, that is, the *Session Controller*. Regarding the *Session Manager*, this acts as an abstraction layer over the distributed infrastructure (i.e., the multiple Kubernetes clusters). It exposes an HTTP API

that translates high-level telepresence operations, such as creating a session or scheduling compute resources for a client in a latency-optimized location, into the necessary cluster-specific API calls. This allows infrastructure management to be integrated directly into the application layer, controlled via these remote functions.

The final objective—*evaluate the performance of the proposed solution and extract conclusions*—encompassed a comparative analysis and benchmarking of the framework. The comparative analysis was performed against two baseline cases, modelled using conventional Kubernetes primitives with minor workarounds to create viable comparative scenarios. The results demonstrated a clear trade-off. The proposed framework enabled clients to connect to their allocated workloads more rapidly and reliably, a direct benefit of the allocation and stable network identity mechanisms it provides. In contrast, the baseline cases consumed fewer compute resources (i.e., CPU and memory), as expected. This lower resource usage is a consequence of their simpler design, which lacks the framework's advanced features such as session-awareness, workload elasticity, garbage collection mechanisms, multi-cluster support, and direct integration with the application layer. Nevertheless, the trade-offs identified in this analysis are not static. The following section on future work explores a few potential optimizations designed to minimize these overheads and further enhance the framework's overall efficiency.

The contributions of this dissertation have been accepted for publication and are scheduled for presentation at the fifth Symposium of Applied Science for Young Researchers[1], as part of the "Digital Architectures and Intelligent Infrastructure" area.

6.1 Future Work

Despite the accomplishment of all the proposed objectives, the process of designing and evaluating the resulting framework has revealed several areas for future work. While the current solution provides a robust foundation, its performance trade-offs can be further optimized, and its capabilities can be extended to support even more complex scenarios. The following contents detail potential architectural enhancements, including a redesign of the Session resource and integration with edge computing frameworks like Liqo, as well as new avenues for experimental validation.

Enhancements for the Session Resource As raised in the discussion of results (Section 5.5), a redesign of the Session resource could significantly reduce the framework's memory consumption. The current issue arises from data duplication: when multiple Session resources are created from the same template, each instance stores a complete copy of the same static configuration data (e.g., PodTemplate, TimeoutSeconds) within its `.spec` field. Because the PodTemplate in particular can be a considerably large data structure, this duplication leads to inefficient memory usage in the etcd data store. The proposed solution is to decouple this static data into a new, reusable SessionTemplate resource. In this new model, the Session resource's `.spec` field would no longer contain the full workload definition. Instead, it would only store the truly dynamic data, the client map, along with a reference to its parent SessionTemplate. This design would eliminate redundant data storage, leading to a significant reduction in the memory footprint of each Session resource.

[1]Symposium of Applied Science for Young Researchers: `https://sasyr.ipb.pt/EN_index.html` (Accessed 28-06-2025)

The framework's connection latency could be further optimized by enhancing the Pod reuse policy. Currently, the ReutilizeTimeoutSeconds parameter only allows an idle Pod to be reused by a new client joining the same Session. A significant improvement would be to make this behaviour configurable, allowing idle Pods to be reused by clients joining any Session that derives from the same template. This would create a shared pool of warm Pods for a given session type, increasing the probability that a new client can connect to an already-running Pod and thereby minimizing overall connection latency.

A potential limitation of the current framework is that the Session resource can only manage individual Pods. A valuable area for future work would be to extend this capability to allow other native Kubernetes resources, such as Deployments or StatefulSets, to be bound to a Session. A primary use case for this would be deploying stateless application services, for instance, an HTTP server that provides static content (e.g., 3D models, images, sounds) exclusively to the clients within a specific session. This concept could be further extended to bind resources not just to a single Session, but to an entire group of Sessions created from the same template. For example, a single Deployment for an HTTP server could be shared among all Sessions of a particular type. The framework's controller would create this shared resource only when the first client joined any Session in the group, and would garbage-collect it only after the last client across all related Sessions had departed. It is important to clarify the separation of concerns in such a design. The *Session Controller* would only be responsible for managing the lifecycle (i.e., creation and deletion) of the bounded resource. The internal reconciliation of that resource, for example managing the Pod replicas of a Deployment, would remain the responsibility of its own native Kubernetes controller.

Liqo As identified in Section 4.2, the current multi-cluster design introduces challenges related to resource synchronization and concurrency. These issues stem from the fact that the *Session Manager* is tasked with two distinct responsibilities: providing a telepresence API for the application layer, and managing the synchronization of resources across multiple Kubernetes clusters. The proposed improvement is to decouple these responsibilities by delegating the synchronization tasks to a specialized framework, Liqo.

As identified in the literature review, Liqo is an inter-cluster networking tool capable of establishing an overlay network across multiple Kubernetes clusters. However, beyond networking, its relevant capability for this matter is its workload offloading mechanism. Liqo achieves this through resource reflection [117], a process that synchronizes control plane information—Pod resources—from one cluster to another. This is implemented via a concept called namespace extension, where a local namespace is bound to a remote cluster. Any Pod created in this offloading-enabled namespace is then automatically reflected on the designated remote cluster and scheduled for execution by its local scheduler.

Leveraging this concept, a single primary cluster would host the Session resources and the framework's controllers. The *Session Controller*, running only in this primary cluster, would reconcile the Session resources and schedule Pods across the entire infrastructure. By delegating resource synchronization to Liqo, the previously identified problems would be resolved, as the *Session Manager* would no longer need to make direct API calls to each cluster. A single controller can therefore manage the entire infrastructure by creating Pod resources in the primary cluster. Liqo then handles the two-way synchronization: it reflects the Pod specification to the appropriate remote cluster for its local scheduler to perform the spin-up, and continuously mirrors the Pod's status back to the primary cluster. This redesign would not only reduce development complexity but would also significantly lower the aggregate

memory and CPU consumption across the infrastructure. The current approach requires replicating the entire set of custom controllers (*Session Controller*, *Network Controller*, and *Garbage Collection Controller*) and Session resource data to every cluster. The integration of Liqo would eliminate these redundant controller instances and duplicated resources, leading to a more efficient system.

Another limitation of the current implementation is its simplistic approach to cluster selection, which relies solely on network latency. This can lead to unbalanced load distribution, where multiple clients are directed to the same low-latency cluster, potentially overloading its nodes and degrading performance for all users. A more robust scheduling strategy is required as part of future work. Such a strategy would need to consider multiple factors beyond just latency, including real-time cluster resource utilization (e.g., CPU and memory) and overall QoE. This would enable the system to make more intelligent trade-offs, for instance, by occasionally placing a client in a location with slightly higher latency to avoid overwhelming an already busy cluster.

In addition to the architectural improvements suggested, further work could involve experimenting the framework under different conditions and with different types of workloads. For example, a valuable experiment would be to deploy RenderFusion, the remote rendering system discussed in the state of the art review, using the proposed framework. This would require modifying the ArenaXR application frontend to use the *Session Manager* for resource scheduling.

Beyond specific experiments, improving the developer experience is another area for future work. To facilitate easier integration, for example, a valuable enhancement would be to wrap the *Session Manager*'s HTTP API into an SDK.

Bibliography

[1] *Kubernetes Documentation: Deployments*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Accessed 28-06-2025].

[2] *Kubernetes Documentation: ReplicaSet*. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/. [Accessed 28-06-2025].

[3] *Kubernetes Documentation: StatefulSets*. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/. [Accessed 28-06-2025].

[4] *Kubernetes: Production-Grade Container Orchestration*. https://kubernetes.io/. [Accessed 28-06-2025].

[5] *US Virtual Reality Market Size (2020–2028)*. https://www.oberlo.com/statistics/us-virtual-reality-market-size. [Accessed 28-06-2025].

[6] Giovanni Bartolomeo et al. "Characterizing Distributed Mobile Augmented Reality Applications at the Edge". In: *CoNEXT Companion 2023 - Companion of the 19th International Conference on emerging Networking EXperiments and Technologies*. Association for Computing Machinery, Inc, Dec. 2023, pp. 9–18. isbn: 9798400704079. doi: 10.1145/3624354.3630584.

[7] Lei Zhang et al. "Edge-Based Video Stream Generation for Multi-Party Mobile Augmented Reality". In: *IEEE Transactions on Mobile Computing* 23.1 (2024), pp. 409–422. doi: 10.1109/TMC.2022.3232543.

[8] Vittorio Cozzolino et al. "Nimbus: Towards Latency-Energy Efficient Task Offloading for AR Services". In: *IEEE Transactions on Cloud Computing* 11.2 (2023), pp. 1530–1545. doi: 10.1109/TCC.2022.3146615.

[9] Edward Lu et al. "RenderFusion: Balancing Local and Remote Rendering for Interactive 3D Scenes". In: *2023 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. 2023, pp. 312–321. doi: 10.1109/ISMAR59233.2023.00046.

[10] Wenxiao Zhang et al. "EdgeXAR: A 6-DoF Camera Multi-target Interaction Framework for MAR with User-friendly Latency Compensation". In: *Proc. ACM Hum.-Comput. Interact.* 6.EICS (June 2022). doi: 10.1145/3532202. url: https://doi.org/10.1145/3532202.

[11] Theodoros Theodoropoulos et al. "Cloud-based XR Services: A Survey on Relevant Challenges and Enabling Technologies". In: *Journal of Networking and Network Applications* 2 (1 2022). doi: 10.33969/j-nana.2022.020101.

[12] Theodoros Theodoropoulos et al. "Cross-Cluster Networking to Support Extended Reality Services". In: *IEEE Network* (2024). doi: 10.1109/MNET.2024.3453301.

[13] Nuno Pereira et al. "Arena: The augmented reality edge networking architecture". In: *Proceedings - 2021 IEEE International Symposium on Mixed and Augmented Reality, ISMAR 2021*. Institute of Electrical and Electronics Engineers Inc., 2021, pp. 479–488. isbn: 9781665401586. doi: 10.1109/ISMAR52148.2021.00065.

[14] *WebRTC: Real-time communication for the web*. https://webrtc.org/?hl=pt-br. [Accessed 28-06-2025].

[15] *Kubernetes Documentation: Service*. https://kubernetes.io/docs/concepts/services-networking/service/. [Accessed 28-06-2025].

[16] *Kubernetes Documentation: Pods*. https://kubernetes.io/docs/concepts/workloads/pods/. [Accessed 28-06-2025].

[17] *Code of Good Practices and Conduct of P.Porto*. https://www.iscap.ipp.pt/regulamentos/CodigoboaspraticasedecondutaIPP.pdf. [Accessed 28-06-2025]. 2020.

[18] Kai Petersen et al. "Systematic mapping studies in software engineering". In: *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*. EASE'08. Italy: BCS Learning & Development Ltd., 2008, pp. 68–77.

[19] Akihiro Kiuchi et al. "MiniMates: Miniature Avatars for AR Remote Meetings within Limited Physical Spaces". In: *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. CHI '25. New York, NY, USA: Association for Computing Machinery, 2025. isbn: 9798400713941. doi: 10.1145/3706598.3714328. url: https://doi.org/10.1145/3706598.3714328.

[20] Ximing Wu et al. "Edge-assisted Real-time Dynamic 3D Point Cloud Rendering for Multi-party Mobile Virtual Reality". In: *Proceedings of the 32nd ACM International Conference on Multimedia*. MM '24. Melbourne VIC, Australia: Association for Computing Machinery, 2024, pp. 2824–2832. isbn: 9798400706868. doi: 10.1145/3664647.3681650. url: https://doi.org/10.1145/3664647.3681650.

[21] Sruti Srinidhi, Edward Lu, and Anthony Rowe. " XaiR: An XR Platform that Integrates Large Language Models with the Physical World ". In: *2024 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. Los Alamitos, CA, USA: IEEE Computer Society, 2024, pp. 759–767. doi: 10.1109/ISMAR62088.2024.00091. url: https://doi.ieeecomputersociety.org/10.1109/ISMAR62088.2024.00091.

[22] Andrew Kang, Hanspeter Pfister, and Tica Lin. "PanoCoach: Enhancing Tactical Coaching and Communication in Soccer with Mixed-Reality Telepresence". In: *Companion Proceedings of the 2024 Conference on Interactive Surfaces and Spaces*. ISS Companion '24. Vancouver, BC, Canada: Association for Computing Machinery, 2024, pp. 13–17. isbn: 9798400712784. doi: 10.1145/3696762.3698043. url: https://doi.org/10.1145/3696762.3698043.

[23] Xincheng Huang et al. "VirtualNexus: Enhancing 360-Degree Video AR/VR Collaboration with Environment Cutouts and Virtual Replicas". In: *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. UIST '24. Pittsburgh, PA, USA: Association for Computing Machinery, 2024. isbn: 9798400706288. doi: 10.1145/3654777.3676377. url: https://doi.org/10.1145/3654777.3676377.

[24] Huajian Qiu et al. "ViGather: Inclusive Virtual Conferencing with a Joint Experience Across Traditional Screen Devices and Mixed Reality Headsets". In: *Proc. ACM Hum.-Comput. Interact.* 7.MHCI (2023). doi: 10.1145/3604279. url: https://doi.org/10.1145/3604279.

[25] Andrew Irlitti et al. "Volumetric Mixed Reality Telepresence for Real-time Cross Modality Collaboration". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. Hamburg, Germany: Association for Computing Machinery, 2023. isbn: 9781450394215. doi: 10.1145/3544548.3581277. url: https://doi.org/10.1145/3544548.3581277.

[26] Mehrad Faridan, Bheesha Kumari, and Ryo Suzuki. "ChameleonControl: Teleoperating Real Human Surrogates through Mixed Reality Gestural Guidance for Remote

Hands-on Classrooms". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. Hamburg, Germany: Association for Computing Machinery, 2023. isbn: 9781450394215. doi: 10.1145/3544548.3581381. url: https://doi.org/10.1145/3544548.3581381.

[27] Mallesham Dasari et al. "Scaling VR Video Conferencing". In: *2023 IEEE Conference Virtual Reality and 3D User Interfaces (VR)*. 2023, pp. 648–657. doi: 10.1109/VR55154.2023.00080.

[28] Xinjun Cai et al. "DiTing: Edge Assisted Real-time ID-aware Visual Interaction for Multi-user Augmented Reality". In: *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*. 2023, pp. 737–744. doi: 10.1109/ICPADS56603.2022.00101.

[29] Anik Mallik and Jiang Xie. "H.264 Video Encoding-based Edge-assisted Mobile AR Systems: Network and Energy Issues". In: *ICC 2022 - IEEE International Conference on Communications*. 2022, pp. 1046–1051. doi: 10.1109/ICC45855.2022.9838862.

[30] Tero Kaarlela et al. "Digital Twins Utilizing XR-Technology as Robotic Training Tools". In: *Machines* 11 (1 2023). issn: 20751702. doi: 10.3390/machines11010013.

[31] *Agones: Host, Run and Scale dedicated game servers on Kubernetes*. https://agones.dev/site/. [Accessed 28-06-2025].

[32] *OpenKruiseGame: A Custom Kubernetes Workload Designed for Game Server Scenarios*. https://openkruise.io/kruisegame/introduction. [Accessed 28-06-2025].

[33] Leonardo Poggiani et al. "Live Migration of Multi-Container Kubernetes Pods in Multi-Cluster Serverless Edge Systems". In: *Proceedings of the 1st Workshop on Serverless at the Edge*. Association for Computing Machinery (ACM), June 2024, pp. 9–16. isbn: 9798400706479. doi: 10.1145/3660319.3660330.

[34] Thanh Nguyen Nguyen et al. "A Design and Development of Operator for Logical Kubernetes Cluster over Distributed Clouds". In: *Proceedings of IEEE/IFIP Network Operations and Management Symposium 2024, NOMS 2024*. Institute of Electrical and Electronics Engineers Inc., 2024. isbn: 9798350327939. doi: 10.1109/NOMS59830.2024.10575914.

[35] Shahmir Ejaz and Mays Al-Naday. "FORK: A Kubernetes-Compatible Federated Orchestrator of Fog-Native Applications Over Multi-Domain Edge-to-Cloud Ecosystems". In: *Proceedings of the 27th Conference on Innovation in Clouds, Internet and Networks, ICIN 2024*. Institute of Electrical and Electronics Engineers Inc., 2024, pp. 57–64. isbn: 9798350393767. doi: 10.1109/ICIN60470.2024.10494435.

[36] Cristopher Chiaro et al. "Latency-aware Scheduling in the Cloud-Edge Continuum". In: *Proceedings of IEEE/IFIP Network Operations and Management Symposium 2024, NOMS 2024*. Institute of Electrical and Electronics Engineers Inc., 2024. isbn: 9798350327939. doi: 10.1109/NOMS59830.2024.10575183.

[37] Tarik Zakaria Benmerar et al. "Intelligent Multi-Domain Edge Orchestration for Highly Distributed Immersive Services: An Immersive Virtual Touring Use Case". In: *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*. 2023, pp. 381–392. doi: 10.1109/EDGE60047.2023.00061.

[38] Ilias Syrigos, Nikos Makris, and Thanasis Korakis. "Multi-Cluster Orchestration of 5G Experimental Deployments in Kubernetes over High-Speed Fabric". In: *2023 IEEE Globecom Workshops, GC Wkshps 2023*. Institute of Electrical and Electronics Engineers Inc., 2023, pp. 1764–1769. isbn: 9798350370218. doi: 10.1109/GCWkshps58843.2023.10465054.

[39] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. "Security automation for multi-cluster orchestration in Kubernetes". In: *2023 IEEE 9th International Conference on Network Softwarization: Boosting Future Networks through Advanced Softwarization, NetSoft 2023 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., 2023, pp. 480–485. isbn: 9798350399806. doi: 10.1109/NetSoft57336.2023.10175419.

[40] Andrew Jeffery, Heidi Howard, and Richard Mortier. "Rearchitecting Kubernetes for the Edge". In: *EdgeSys 2021 - Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking, Part of EuroSys 2021*. 2021. doi: 10.1145/3434770.3459730.

[41] *Microsoft HoloLens*. https://learn.microsoft.com/en-us/hololens/. [Accessed 28-06-2025].

[42] *Meta Quest*. https://www.meta.com/quest/. [Accessed 28-06-2025].

[43] Fumio Kishino Paul Milgram. "A Taxonomy of Mixed Reality Visual Displays". In: *IEICE Transactions on Information and Systems* E77-D.12 (Dec. 1994), pp. 1321–1329.

[44] *WebXR standards*. https://immersiveweb.dev/. [Accessed 28-06-2025].

[45] *OAuth 2.0 standard*. https://oauth.net/2/. [Accessed 28-06-2025].

[46] *JSON Web Token*. https://jwt.io/. [Accessed 28-06-2025].

[47] *WebAssembly: Binary instruction format for a stack-based virtual machine*. https://webassembly.org/. [Accessed 28-06-2025].

[48] *ArenaXR: Python library*. https://docs.arenaxr.org/content/python/. [Accessed 28-06-2025].

[49] *Jitsi Meet*. https://jitsi.github.io/handbook/docs/intro. [Accessed 28-06-2025].

[50] *MediaPipe: Pose landmark detection guide*. https://ai.google.dev/edge/mediapipe/solutions/vision/pose_landmarker. [Accessed 28-06-2025].

[51] *Unity*. https://unity.com/. [Accessed 28-06-2025].

[52] *etcd: A distributed, reliable key-value store for the most critical data of a distributed system*. https://etcd.io/. [Accessed 28-06-2025].

[53] *Cluster API Documentation*. https://cluster-api.sigs.k8s.io/. [Accessed 28-06-2025].

[54] *YAML: YAML Ain't Markup Language*. https://yaml.org/. [Accessed 28-06-2025].

[55] *Kubernetes Github Repository: Deployment Controller*. https://github.com/kubernetes/kubernetes/blob/master/pkg/controller/deployment/deployment_controller.go. [Accessed 28-06-2025].

[56] *Logical Cluster Provider GitHub Repository*. https://github.com/kubernetes-sigs/logical-cluster. [Accessed 28-06-2025].

[57] *Liqo Documentation*. https://docs.liqo.io/en/v1.0.0-rc.2/. [Accessed 28-06-2025].

[58] *Cilium Documentation*. https://docs.cilium.io/en/stable/. [Accessed 28-06-2025].

[59] *Submariner Documentation*. https://submariner.io/. [Accessed 28-06-2025].

[60] *Cilium Blog: Deep Dive into Cilium Multi-cluster*. https://cilium.io/blog/2019/03/12/clustermesh/. [Accessed 28-06-2025].

[61] Mark Mandel. *Introducing Agones: Open-source, multiplayer, dedicated game-server hosting built on Kubernetes*. https://cloud.google.com/blog/products/containers-kubernetes/introducing-agones-open-source-multiplayer-dedicated-game-server-hosting-built-on-kubernetes. [Accessed 28-06-2025]. 2018.

[62] Gautam Manchandani. *Best Practices for Traditional PvE Games*. `https://openkruise.io/kruisegame/best-practices/pve-game/`. [Accessed 28-06-2025]. 2025.

[63] *Kubernetes Documentation: API Reference*. `https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.32/`. [Accessed 28-06-2025].

[64] Kubernetes Documentation: Operator pattern. `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/`. [Accessed 28-06-2025].

[65] *Kubernetes Documentation: Command line tool (kubectl)*. `https://kubernetes.io/docs/reference/kubectl/`. [Accessed 28-06-2025].

[66] *Agones Documentation: Fleet Specification*. `https://agones.dev/site/docs/reference/fleet/`. [Accessed 28-06-2025]. 2025.

[67] *Agones Documentation: Custom Controller for Agones Game Servers*. `https://0-8-0.agones.dev/site/docs/examples/custom-controller/`. [Accessed 28-06-2025]. 2025.

[68] *Agones Documentation: Allocator Service*. `https://agones.dev/site/docs/advanced/allocator-service/`. [Accessed 28-06-2025]. 2025.

[69] *OpenKruise: Automate application management on Kubernetes*. `https://openkruise.io/`. [Accessed 28-06-2025].

[70] Gautam Manchandani. *OpenKruiseGame Documentation: Deploy Gameservers*. `https://openkruise.io/kruisegame/user-manuals/deploy-gameservers/`. [Accessed 28-06-2025]. 2025.

[71] Gautam Manchandani. *OpenKruiseGame Documentation: Gameservers Scale*. `https://openkruise.io/kruisegame/user-manuals/gameservers-scale`. [Accessed 28-06-2025]. 2025.

[72] Gautam Manchandani. *OpenKruiseGame Documentation: Best Practices for Agile Delivery and Operations Management of GameServers*. `https://openkruise.io/kruisegame/best-practices/gameserver-delivery-management`. [Accessed 28-06-2025]. 2025.

[73] Qiuyang Liu. *Alibaba Cloud Blog: Agones Summary and Outlook*. `https://www.alibabacloud.com/blog/agones-series-part-5-agones-summary-and-outlook_599431`. [Accessed 28-06-2025]. 2022.

[74] *Kubernetes Documentation: Extending Kubernetes*. `https://kubernetes.io/docs/concepts/extend-kubernetes/`. [Accessed 28-06-2025].

[75] *Kubernetes Documentation: API Aggregation Layer*. `https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation/`. [Accessed 28-06-2025].

[76] *Kubernetes Documentation: The Kubernetes API*. `https://kubernetes.io/docs/concepts/overview/kubernetes-api/`. [Accessed 28-06-2025].

[77] Andrei Kvapil. *Kubernetes Blog: How we built a dynamic Kubernetes API Server for the API Aggregation Layer in Cozystack*. `https://kubernetes.io/blog/2024/11/21/dynamic-kubernetes-api-server-for-cozystack/`. [Accessed 28-06-2025]. 2024.

[78] *Kubernetes Documentation: Custom Resources*. `https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/`. [Accessed 28-06-2025].

[79] *Kubernetes Documentation: Kubernetes Scheduler*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/. [Accessed 28-06-2025].
[80] *Kubernetes Documentation: DaemonSet*. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/. [Accessed 28-06-2025].
[81] *Kubernetes Documentation: Jobs*. https://kubernetes.io/docs/concepts/workloads/controllers/job/. [Accessed 28-06-2025].
[82] *Kubernetes Documentation: CronJob*. https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/. [Accessed 28-06-2025].
[83] *Kubernetes Documentation: Controllers*. https://kubernetes.io/docs/concepts/architecture/controller/. [Accessed 28-06-2025].
[84] *Kubernetes API Conventions*. https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md. [Accessed 28-06-2025].
[85] *Kubernetes Github Repository: ReplicaSet Controller*. https://github.com/kubernetes/kubernetes/blob/master/pkg/controller/replicaset/replica_set.go. [Accessed 28-06-2025].
[86] *Kubernetes Documentation: Extend the Kubernetes API with CustomResourceDefinitions*. https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/. [Accessed 28-06-2025].
[87] *CNCF Blog - Kubernetes Operators: what are they? Some examples*. https://www.cncf.io/blog/2022/06/15/kubernetes-operators-what-are-they-some-examples/. [Accessed 28-06-2025]. 2022.
[88] *CloudNativePG Github Repository*. https://github.com/cloudnative-pg/cloudnative-pg. [Accessed 28-06-2025].
[89] *MongoDB Kubernetes Operator Github Repository*. https://github.com/mongodb/mongodb-kubernetes-operator. [Accessed 28-06-2025].
[90] *EDB Postgres for Kubernetes: Operator capability levels*. https://github.com/mongodb/mongodb-kubernetes-operator. [Accessed 28-06-2025].
[91] *KubeVirt: Building a virtualization API for Kubernetes*. https://kubevirt.io/. [Accessed 28-06-2025].
[92] *KubeVirt User Guide: Virtual Machines Instances*. https://kubevirt.io/user-guide/user_workloads/virtual_machine_instances/. [Accessed 28-06-2025].
[93] Sumanth Reddy. *Debugging namespace deletion issue in Kubernetes*. https://sumanthkumarc.medium.com/debugging-namespace-deletion-issue-in-kubernetes-f6f8b40a4368. [Accessed 28-06-2025]. 2022.
[94] *NGINX Ingress Controller Documentation*. https://docs.nginx.com/nginx-ingress-controller/. [Accessed 28-06-2025].
[95] *STUNner Documentation: Why STUNner*. https://docs.l7mp.io/en/latest/WHY/. [Accessed 28-06-2025].
[96] *The Kubebuilder Book*. https://book.kubebuilder.io/. [Accessed 28-06-2025].
[97] *The Go Programming Language*. https://go.dev/. [Accessed 28-06-2025].
[98] *Controller-runtime Github Repository*. https://github.com/kubernetes-sigs/controller-runtime. [Accessed 28-06-2025].
[99] *Client-go Github Repository: client-go under the hood*. https://github.com/kubernetes/sample-controller/blob/master/docs/controller-client-go.md. [Accessed 28-06-2025].
[100] *Kubernetes Documentation: API Concepts*. https://kubernetes.io/docs/reference/using-api/api-concepts. [Accessed 28-06-2025].

[101] *Red Hat Blog: Kubernetes Operators Best Practices*. https://www.redhat.com/en/blog/kubernetes-operators-best-practices. [Accessed 28-06-2025]. 2019.
[102] *OpenKruise Blog: Learning Concurrent Reconciling*. https://openkruise.io/blog/learning-concurrent-reconciling/. [Accessed 28-06-2025]. 2019.
[103] *controller-runtime Github Repository FAQ*. https://github.com/kubernetes-sigs/controller-runtime/blob/main/FAQ.md. [Accessed 28-06-2025].
[104] *MetaController GitHub Repository*. https://github.com/GoogleCloudPlatform/metacontroller/tree/master/examples/service-per-pod. [Accessed 28-06-2025].
[105] *Kubernetes Documentation: Ingress*. https://kubernetes.io/docs/concepts/services-networking/ingress/. [Accessed 28-06-2025].
[106] *Gin Web Framework*. https://gin-gonic.com/. [Accessed 28-06-2025].
[107] *Kubernetes Documentation: Organizing Cluster Access Using kubeconfig Files*. https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/. [Accessed 28-06-2025].
[108] *Kubernetes Documentation: Secrets*. https://kubernetes.io/docs/concepts/configuration/secret/. [Accessed 28-06-2025].
[109] *Kubernetes Documentation: ConfigMaps*. https://kubernetes.io/docs/concepts/configuration/configmap/. [Accessed 28-06-2025].
[110] *MDN Web Documentation: WebRTC Signaling and video calling*. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling. [Accessed 28-06-2025].
[111] *MDN Web Documentation: Session Description Protocol*. https://developer.mozilla.org/en-US/docs/Glossary/SDP. [Accessed 28-06-2025].
[112] *MDN Web Documentation: Introduction to WebRTC protocols*. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols. [Accessed 28-06-2025].
[113] *STUNner GitHub Repository*. https://github.com/l7mp/stunner. [Accessed 28-06-2025].
[114] *Geeks for Geeks: Poisson Processes*. https://www.geeksforgeeks.org/maths/poisson-processes/. [Accessed 28-06-2025]. 2025.
[115] *Python Programming Language*. https://www.python.org/. [Accessed 28-06-2025].
[116] *MDN Web Documentation: WebRTC connectivity*. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity. [Accessed 28-06-2025].
[117] *Liqo Documentation: Offloading*. https://docs.liqo.io/en/v1.0.0-rc.2/. [Accessed 28-06-2025].