Grant Agreement No.: 101016509
Research and Innovation action
Call Topic: ICT-40-2020: Cloud Computing

CHARITY
Cloud for Holography and Augmented RealITY

Cloud for Holography and Cross Reality

D3.2: Energy, data and computational-efficient mechanisms supporting dynamically adaptive and network-aware services (final version)

Version: v0.4

Deliverable type	R (Document, report)
Dissemination level	PU (Public)
Due date	31/12/2023
Submission date	21/12/2023
Lead editor	Massimiliano Corsini (CNR)
Authors	Antonios Makris (HUA), Konstantinos Tserpes (HUA), Theodoros Theodoropoulos (HUA), Michael McElligott (CAI), Tom Loven (PLEXUS), Laura Sande (PLEXUS), Yago Gonzalez Rozas (PLEXUS), Antonis Protopsaltis (ORAMA), Maria Pateraki (ORAMA), Enrico Zschau (SRT), Federico Ponchio (CNR), Pedro Sá (ONE), Joao Rodrigues (DOTES), Tarik Taleb (ICT-FI), Nora Taleb (ICT-FI), Massimiliano Corsini (CNR), Somnath Dutta (CNR)
Reviewers	Fermin Calvo, Ferran Diego Andilla
Work package, Task	WP3
Keywords	XR enablers, storage system, software dynamic adaptation, rendering algorithms, data compression

Abstract

WP3 is the work package devoted to the research and development of strategies, mechanisms, and algorithms, for the efficient exploitation of available network and computational resources to enable sophisticated XR applications. Several aspects are investigated; innovative management of advanced computational resources, intelligent solutions for data storage and data access, innovative strategies to adapt the Quality of Experience of the running application according to the available resources. Regarding the advancement of XR technologies, we investigated techniques to obtain more complex realistic VR simulation, technical solutions for rendering adaptation, novel algorithms for 3D point

cloud compression and for the next-gen multi-user AR gaming experience, and for the editing and streaming of immersive 360 video.

Document revision history

Version	Date	Description of change	List of contributor(s)
v0.1	17/10/23	Initial skeleton	Massimiliano Corsini (CNR)
V0.2	10/12/23	Contributions	Antonios Makris (HUA), Konstantinos Tserpes (HUA), Theodoros Theodoropoulos (HUA), Michael McElligott (CAI), Tom Loven (PLEXUS), Laura Sande (PLEXUS), Yago Gonzalez Rozas (PLEXUS), Antonis Protopsaltis (ORAMA), Maria Pateraki (ORAMA), Enrico Zschau (SRT), Federico Ponchio (CNR), Pedro Sá (ONE), Joao Rodrigues (DOTES), Tarik Taleb (ICT-FI), Nora Taleb (ICT-FI), Massimiliano Corsini (CNR)
V0.3	20/12/2023	Post internal review	Ferran Diego Andilla (TID), Fermin Calvo (PLEXUS), Massimiliano Corsini (CNR)
V0.4	21/12/2023	Version for the GA	Massimiliano Corsini (CNR), Alessandro Bassi (EURESCOM)

Disclaimer

[1] http://creativecommons.org/licenses/by-nc-nd/3.0/deed.en_US

Acknowledgement

The research conducted by CHARITY receives funding from the European Commission H2020 programme under Grant Agreement No 101016509. The European Commission has no responsibility for the content of this document.

Executive Summary

The research and development activities in the WP3 will drive the advancement of complex and highly demanding, in terms of computation and/or bandwidth resources, XR applications. The systems and the algorithms delivered by WP3 are under integration into the CHARITY platform and in some of the Use Cases (UCs) of the CHARITY project according to the integration plan described in the Deliverable 4.2. These ad-hoc technological solutions regard different aspects of the deployment, the development, and the lifecycle management of advanced XR applications.

Advanced computing mechanisms to enable the management of VMs, essential to manage the complex software stack of XR applications, and GPUs, necessary for XR application has been developed and tested.

A flexible monitoring framework which fills the needs of the different UCs of CHARITY has been defined and realized. This monitoring framework is based on the open-source Prometheus technology. According to the metrics identified, different types of exporters are under development to enable the different UCs to monitor their metrics of interest. The monitoring framework interacts with the orchestration system of the CHARITY platform.

A new intelligent data management system, for highly efficient data storage and data access has been developed in the ambit of the CHARITY project. This system is called CHES, which stands for CHARITY Edge Storage. The CHES takes into account the high degree of heterogeneity that characterises the computational resources considered in the CHARITY project and it is lightweight so that it can also be used on edge devices with limited capabilities, such as a Raspberry Pi. The CHES has reached its third version. Experimental results demonstrate the validity of the system, also in terms of the KPIs achieved. The CHES is released as an open-source software under GPL 3 license.

Often, XR applications are demanding in terms of computational and network resources, and the environmental circumstances may become sub-optimal during their running, for example, due to a reduction of bandwidth. In many of these cases, it is convenient to modify the behaviour of the running application so that the application itself adapts to the available resources instead of deploying it again in an environment with more resources. In CHARITY, a variant of the MAPE-K Loop [7] approach, based on micro-services, is proposed to perform an adaptation of XR applications at runtime. This novel solution has been carefully designed and some preliminary studies related to the flight simulator of the Collins Aerospace (UC3-2 Manned-Unmanned Operations Trainer Application) have been conducted.

Virtual Reality applications often require high realism in rendering and physical simulation. The UC2-1 VR Medical Training Application of CHARITY is one of these types of virtual reality applications. This VR UC is currently being optimized by exploiting multi-threading to make the rendering and the physics part more efficient. Experiments conducted demonstrate that the physics simulation can benefit of this type of optimization, while multi-threaded rendering has shown limits of applicability in Unity, that is the framework where the UC2-1 runs.

The immersive applications, to reach high-quality levels of experience, require ultra-low latency and large bandwidth resources. To improve the performance of immersive applications, we have investigated how adaptive rendering solution may be integrated into some selected UCs to reduce the motion-to-photon computational burden, and hence, the overall latency of the application.

Another important aspect of immersive applications is 360-degree video. The UC2-2 VR Tour Creator Application of CHARITY regards the advancement of a platform for the creation of virtual tours based on 360° video. To follow this goal new features and technological advancements have been implemented in the Cyango Cloud Studio, that is the name of the UC2-2 platform. Additionally, investigation and experiments about cross-video streaming has been done.

Specific data services to satisfy the needs of XR applications like the UC1-3 Holographic Assistant and the UC3-1 Collaborative AR Gaming have been developed and tested. Respectively, a novel ad-hoc point cloud encored/decoder, to allow the transmission from the cloud to the edge (the holographic

display) of a large amount of 3D points, and a geometry processing algorithm to enable the creation and the continuos updated real and the virtual gaming environment, called *Mesh Merger* service. At the moment of writing the Mesh Merger prototype is ready and the first integration phase under finalization; the PC encoder/decoder is under tests inside its target UC.

In this deliverable, the research and the technical work related of the activities mentioned above is described and the corresponding results and software products are reported.

Table of Contents

List of Figures

List of Tables

Abbreviations

AR	Augmented Reality
CCU	Concurrent Users
CHES	CHARITY Edge Storage
COS	Container Orchestration Information
CPU	Central Processing Unit
CRD	Custom Resource Definition
CSI	Container Storage Information
DLF	(Kubernetes) Dynamic Lifecycle Framework
DLSP	Dynamic Software Product Line
DoW	Description of Work
GPU	Graphics Processing Unit
HMD	Head Mounted Display
ISP	Internet Service Provider
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
PC	Point Cloud
PVC	Persistent Volume Claim
QoE	Quality of Experience
QoS	Quality of Service
SPLE	Software Production Line Engineering
TCP	Transmission Control Protocol
WP	Work Package
UC	Use Case
UDP	User Datagram Protocol
VM	Virtual Machine
VR	Virtual Reality
XR	Extended Reality

1 Introduction

WP3 is devoted to the development of strategies, mechanisms, and algorithms that support both parts of the CHARITY Framework (described in the Deliverable D1.3 and D4.1) and the Use Cases (UCs) (described in D1.2 and D1.3). The UCs support is provided by developing specific technologies and data services that are used by some of the XR applications selected as case studies in the CHARITY project. Most of the technological solutions developed are generic and can be applied by any advanced XR application. The XR data services have been developed to fulfil the specific requirements of specific UCs, but they can be used/adopted by other XR applications with similar needs, beyond the ones involved in the project itself. For example, the 3D point cloud encoder/decoder can be used by any XR application which needs to transfer a huge amount of 3D data points.

The R&D work conducted is reported in the following way: first, a brief introduction of the different activities is given, together with their mapping with the WP3 Tasks. The relationships between the WP3 activities and the rest of the project (others WPs/tasks) are also described. After this introduction, the R&D activities are described in details in the subsequent sections. The activities description is organized by topic and does not follow the task subdivision.

1.1 Activities in a nutshell

The activities described in the next sections are:

- Monitoring framework
- Virtual Machine and GPU support in Kubernetes environments
- CHARITY Edge Storage (CHES)
- Resource-aware Adaptation Mechanisms
- Transformation of the flight simulator UC to a cloud-native XR application
- Multi-threading optimization of rendering and physics simulation for realistic VR applications
- Adaptive rendering for high QoE
- Point Cloud Encoding/Decoding
- Immersive Virtual Tours builder platform
- Evaluation of 360 video streaming based on different camera types, and in terms of different metrics (e.g. Glass to Glass latency, Streaming Frame Rate, etc)
- Mesh Merger

The *monitoring* of the available network and computational resources plays a fundamental role for their assignment within a system according to specific requests, i.e. for the orchestration, and for the applications performance management, i.e. QoE or latency. Regarding performance, sometime, the applications should adapt their behaviour during their execution to guarantee a target QoE or reduce it in case of loss of resources. The monitoring is based on the open-source Prometheus framework. Such technology is configured and integrated to satisfy the CHARITY requirements. This is one of the main activity of the Task 3.1, the task committed to the efficient exploitation of computing resources. The approach followed for monitoring and the architecture of the monitoring system is detailed in Section 2.

Virtual Machine and GPU support in Kubernetes environments is another activity about the advanced exploitation of computing resources (Task 3.1). XR applications can greatly benefits of such mechanisms. VMs are essential to deal with intricate third-party libraries, but their usage creates challenges to orchestrate them together with containers. On the other side, containers enable Cloud Native architectures and micro-services, which facilitate the development and deployment of XR applications. GPUs are fundamental for almost all the XR applications. Therefore, their support should

also be considered to take into account a comprehensive XR orchestration process. For example, an XR orchestration solution should be able to recognize the GPU requirements of specific components to plan its deployment in an optimal way. This activity and the experiments conducted are described in details at the begin of Section 5, as part of the XR enabling technologies.

The *CHARITY Edge Storage (CHES)* is a solution for the optimized edge storage services to the CHARITY framework and its hosted applications. The goals of the CHES are ambitious; it should work on hardware with limited resources (e.g. a Raspberry Pi), and, at the same time, should provide reliable, robust, and fast access to the information. It is based on Lightweight Kubernetes (K3s), MinIO and Prometheus technologies. The CHES is the main activity of the Task 3.2. The current status of the development is detailed in Section 3.

The *Resource-aware Adaptation Mechanisms* are designed according to the MAPE K-Loop [7] approach. It consists in adapting the running applications according to the available resources by acting on applications' variability points (changing the frame rate, changing the resolution, and so on). Such adaptation can be achieved by dynamically modifying the configuration of the application. In CHARITY, a variant of the standard MAPE K-Loop approach is proposed which leverages cloud-native rolling update functionality to seamlessly reconfigure an application an runtime while maintaining service continuity. This concept is explained in details Section 4. This is the main activity of Task 3.3.

In CHARITY, we are also modifying the architecture of some UCs such that these XR applications become cloud-native. The case of UC3-1 Manned-unmanned Operations Training Application is particularly complex and requires a lot of effort. Such technical effort has been described in Section 5.1.

Many VR applications require both high-quality rendering and accurate physical simulation to provide a realistic virtual environment. One of the UC of CHARITY, UC2-1, regards VR simulation for medical training. The idea is to improve the performance of this VR medical simulation platform by employing *multi-threading to speed up rendering and physics simulation* (as detailed in Sections 5.3 and 5.4). The multi-threading exploitation of rendering and physics for the realistic simulation of virtual environments is also part of the activities for the efficient exploitation of computational resources (Task 3.1).

VR immersive applications, to be comfortable, satisfying, and convincing, require low latency and high bandwidth. In CHARITY, we aim to integrate in two UCs, the UC2-1 VR Medical Training Simulator and the UC3-1 Manned-unmanned Operations Training Application, an adaptive rendering algorithm to reduce the computational burden and, consequently, the motion-to-photon latency. This activity is described in Section 5.5.

The *Point Cloud Encoder/Decoder* is the main component of the UC1-3 Holographic Assistant. The Holo Assistant must efficiently transmit a huge amount of 3D data from the cloud to the edge (the holographic display). This UC is described in detail in D1.2. The current status of the development of this innovative PC encoder/decoder is given in Section 5.6. This activity is conducted in the ambit of the Task 3.4, devoted to the development of an adaptive data compression/decompression for high demanding rendering applications.

Another activity of the Task 3.4 is the development of a *virtual tour platform* (UC2-2 VR Tour Creator Application) to create interactive immersive VR experiences. This platform, called *Cyango*[2], supports 360 videos, panoramas, 3D models, standard images and videos and basic 3D meshes. The technological advancements of the Cyango platform are described in Section 5.7. In this context, it is also important to understand how streaming protocols operate under different scenarios and when different cameras are used. In this regard, it is important to evaluate the performance of these streaming protocols in terms of different metrics, such as glass to glass latency, streaming frame rate, display refresh rate. Experiments about such aspects are reported and analyzed in depth in Section 5.8.

[2] https://www.cyango.com

The *Mesh Merger* is a data service built on a geometry processing algorithm which runs on the server to enable the UC3-1 Collaborative Game. This algorithm integrates the different pieces of geometry of the environment so that the game players can interact with a virtual environment. Initially, the Mesh Merger is used to set up the virtual environment to resolve collisions, during the game to align the changes of the real environment with the virtual one. For example, if a chair inside a room is moved during the game, and one gamer acquires this change through her smartphone, the Mesh Merger should integrate this environment change in the virtual environment. The Mesh Merger data service is described in Section 5.9. This activity is also conducted in the context of Task 3.4.

1.2 Relationships between the CHARITY Framework and the WP3 Tasks

An overview of the mapping between the CHARITY architecture components and the Work Packages / Tasks is given in Figure 1. The architecture is subdivided into three planes: i) the Domain Specific XR Service Monitoring and Reaction Plane, ii) the XR Service E2E Conducting Plane, and the iii) XR Service Deployment Plane. Further details about the architecture are provided in the respective deliverable, D1.3.

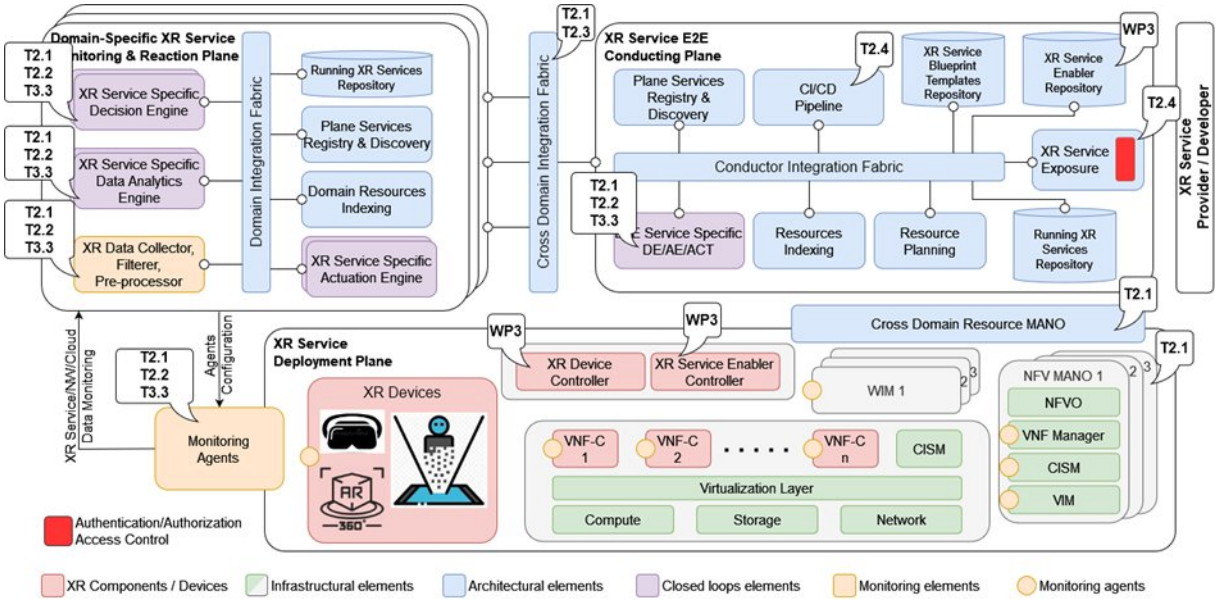


Figure 1: CHARITY Architecture components and project WPs/Tasks mapping.

The *XR Service Deployment Plane* is the plane where XR services are executed. It thus hosts the different Virtual Network Functions (VNFs) that compose the different XR services. The main responsibility of this plane is to manage the computational, network, and storage resources of the infrastructure. Two of the main components of the XR Service Deployment Plane are the *XR Device Controller* and the *XR Service Enabler Controller*. The XR Device Controller is in charge to control the XR devices. This allows to separate the data plane from the control plane. Similarly, the XR Service Enabler Controller is in charge of control specific XR services instead of devices. The XR Device Controller and the XR Service Enabler Controller are developed as part of the activities of the WP3, the last one in particular.

The *Domain-specific Monitoring and Reaction Plane* is responsible for monitoring the service inside a technological or administrative domain. It keeps track of the resource usage and of the XR services running in the domain, and it makes decisions according to the monitored data. In particular, is responsible for implementing XR-specific orchestration mechanisms following a closed-loop model to support the lifecycle management of XR services at the domain level.

The *XR Service E2E Conducting Plane* is responsible for preparing and supporting the resource

orchestration within the respective domains based on the XR Service Blueprint Templates Repository (which stores the templates of the XR services), the XR Service Enabler Repository (which holds the implementation details of the services referenced in the XR Service Blueprints), and the Running XR Services Repository (which provides the ability to track the status of these services).

A detailed description of the different planes and components of the CHARITY architecture can be found in Deliverables D1.3 and 4.2 (Section 3). The roles and details about the different components of the XR E2E Conducting Plane can be found in Section 5 of Deliverable 4.2; more information about the orchestration mechanisms can be found in Section 5, 5.1 and 5.2 of Deliverable 4.2.

The WP3 tasks are connected to the CHARITY architecture as described in the following (see Figure 1, Table 1 and Table 2):

- **Task 3.1 Efficient exploitation of CPUs, GPUs and FPGAs on edge devices.** This task is focused on providing efficient solutions for exploiting computational resources to support the project needs. The main activities are related to the monitoring and the resource indexing, as well as technological and algorithmic solutions for enabling the exploitation of the different and heterogeneous computational resources belonging to CHARITY. The monitoring framework is strictly connected with Tasks 2.1, 2.2, and 3.3.

- **Task 3.2 Efficient storage and caching for AR, VR and Holographic applications.** In the ambit of this task several components for the realization of a distributed edge storage framework spread across heterogeneous edge and cloud nodes, with intelligent data management, high quality performance (QoE), and high-security levels (in collaboration with Task 2.3) are under development. These components, parts of the XR Service Deployment Plane, are: the CHarity Edge Storage (CHES) which is a distributed hybrid storage component and the CHES Registry component that realizes a localized Docker registry in order to support the faster application deploying and limit the network flooding caused by large image downloads during deployment.

- **Task 3.3 Network and infrastructure awareness for efficient exploitation of resources:** This task explores the Dynamic Software Production Line (DSPL) paradigm to adapt XR services dynamically and automatically to network and environment changes. Task 3.3 also designs and develops specific Monitoring, Analytics, Decision and Actuation Engines for both domain and cross-domain levels. This work is related to the realization of the XR Service Specific Analytics Engine, the XR Service Specific Decision Engine, and the XR Service Specific Actuation Engine components, which are parts of the Domain-specific Monitoring and Reaction Plane as well as of the XR Service E2E Conducting Plane.

- **Task 3.4 Adaptive rendering and contextualized data compression / decompression:** The R&D activities conducted in this task relate to the development of the algorithms that will be integrated in data services for XR applications such as the *Point Cloud Encoder/Decoder (PC E/D)*, used by UC 1-3 Holographic Assistant, or the *Mesh Merger*, developed to support UC3-1 Collaborative Gaming. This task is also devoted to R&D activity on immersive video experiences.

To make this document self-containing and more readable, we report below two tables adapted from D4.1. Table 1 contains the name of the component of the CHARITY Framework together with the name of the tasks related to its development, Table 2 contains the names of the algorithms/mechanisms that are at the base of some specific plane/components, and the tasks within which they were studied and developed.

Table 1: CHARITY Component List

Component Name	Architectural Layer	Tasks

Component Name	Architectural Layer	Tasks
Monitoring Agents	Monitoring & Reaction Plane	T3.1, T3.3
XR Service Specific Analytics Engine	Monitoring & Reaction Plane	T2.1, T2.2, T3.3
XR Service Specific Decision Engine	Monitoring & Reaction Plane	T2.1, T2.2, T3.3
XR Service Specific Actuation Engine	Monitoring & Reaction Plane	T2.1, T2.2, T3.3
Running XR Services Repository	Monitoring & Reaction Plane	T2.1, WP3, WP4
Plane Services Registry & Discovery	Monitoring & Reaction Plane	T2.1, WP3, WP4
E2E Service Specific DE/AE/ACT	XR Service E2E Conducting Plane	T2.1, T2.2, T3.3
XR Service Enabler Repository	XR Service E2E Conducting Plane	T2.4, WP3
Running XR Services Repository	XR Service E2E Conducting Plane	T2.1, WP3, WP4
Resource Planning	XR Service E2E Conducting Plane	T3.1
Resource Indexing	XR Service E2E Conducting Plane	T3.1
XR Device Controller	XR Service Deployment Plane	WP3
XR Service Enabler Controller	XR Service Deployment Plane	WP3

Table 2: CHARITY proposed mechanisms and algorithms

Component Name	Component Description	Architectural Layer	Tasks
Prometheus and Monitoring agents	Resource monitoring tool Agent to facilitate VNF monitoring.	Monitoring Agents / Monitoring & Reaction Plane	T2.2, T3.1, T3.3
Adaptative Network Traffic Mechanisms	Mechanisms to dynamically route network traffic accordingly to infrastructure conditions.	DE/AE/ACT Monitoring & Reaction Plane / E2E Conducting Plane	T3.3
XR Service Enabler Repository	Repositories for container images, VM images and metadata.	XR Service Enabler Repository / XR Service E2E Conducting Plane	T2.4, WP3
CHES (CHARITY Edge Storage)	A distributed hybrid storage component spread across heterogeneous edge and cloud nodes with intelligent decisions on data placement, data caching and considerations on performance (QoE) and security.	XR Service Deployment Plane	T3.2
CHES Registry	A sub-component that realizes a localized Docker registry in order to support the faster application deploying and limit the network flooding caused by large image downloads during deployment.	XR Service Deployment Plane	T3.2
Mesh Merger	Data service which create a common mesh of a virtual environment to interact with it, merging collision meshes coming from different user	XR Service Deployment Plane	T3.4

Component Name	Component Description	Architectural Layer	Tasks
	devices.		
3D Point cloud encoder/decoder	Data service component to compress/decompress point cloud for efficient transmission.	XR Service Deployment Plane	T3.4
Decentralised storage / network performance	Measuring the performance of DHT-based decentralised storage platforms such as IPFS and pub-sub based federation networks.	XR Service Deployment Plane	T3.2

For the complete list of components and algorithms/mechanisms, the interested reader is referred to Appendices A, B, and C of Deliverable D4.2. The corresponding tables of D4.2 provide also additional information for each component/algorithm, like the names of the partners involved in the development.

2 Monitoring

The adaptive scheme devised by CHARITY implies the migration of components to offer maximum performance in XR services. The monitoring is a key part between dynamically adaptive network-aware services and efficient exploitation of resources. The continuous monitoring of components and network allows to anticipate performance failures that will affect the QoE, so monitoring, prediction and migration are the key cycle in the CHARITY project after the initial deployment of the applications.

The monitoring framework (Figure 2) designed for CHARITY expands the functionalities of Prometheus, the leading open-source monitoring solution, turning it into a dynamic and reactive tool by adding a custom module to manage communications and configurations. Previously called Monitoring Agent (in D3.1), it has been renamed as Monitoring Manager for a more accurate meaning regarding its functions.

The large community behind Prometheus has sponsored the development of a tool to expand the server capacity, Thanos, a high availability storage module able to collect data from multiple Prometheus servers. It adapts the single cluster environment to the needs of the multi-provider, multi-domain and multi-cluster environment considered in the CHARITY project. Thanos concentrates all the monitoring data in a single point, allowing to run multi-cluster requests for monitoring the combined performance of the XR components.

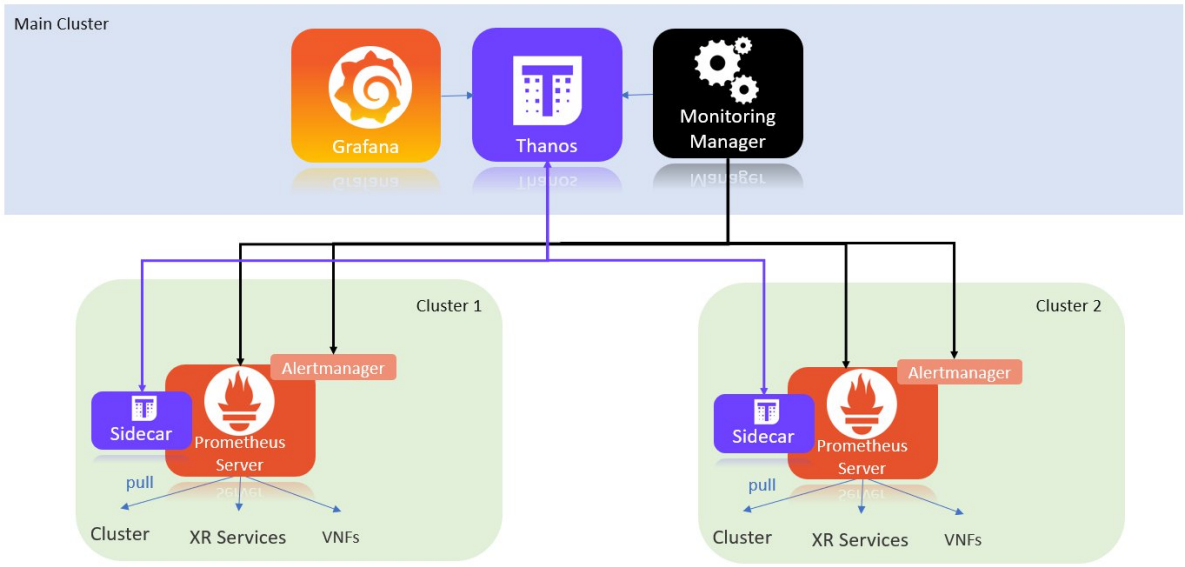


Figure 2: Monitoring Framework.

The monitoring architecture, presented in Deliverable D2.1, responds to the preliminary requirements of XR applications to be deployed on a multi-cloud platform: reduce complexity focusing on prevention and reactivity in ecosystems with heterogeneity of technologies. To translate these formal needs into functional values, it is necessary to identify the components of the architecture of each UC, the links between them, and the needs of each of the developing partners.

2.1.1 Monitoring Manager

CHARITY's monitoring platform use a set of open source-based tools, Prometheus, Grafana and Thanos, to achieve the goals of monitoring, alerting, data storage and visualization. However, the working functionality of this tools is that they are particularly focused for a single-cluster platform with basic adaptability, and we aim to achieve a multi-cluster platform with dynamic adaptation. To achieve this, a new monitoring component is necessary, focused on prevention and reactivity, and capable of following changes in the network over time and migrations. The Monitoring Manager oversees carrying

out the necessary tasks, receiving instructions from the HLO, updating the configurations of the Prometheus servers, and responding to the requests of the Data Analytics Engines. Table 3 provides an overview of these communications.

Table 3: Monitoring Manager communications.

Source	Destiny	Description
HLO	Monitoring Manager	Change configuration and alerting
Monitoring Manager	Prometheus server	Change configuration and alerting
Monitoring Manager	Thanos	Change configuration
HLO	Monitoring Manager	Monitor new component
Monitoring Manager	Prometheus server	Monitor new component
Monitoring Manager	Thanos	Collect data from a new component
HLO	Monitoring Manager	Stop monitoring a component but don't delete Thanos stored data
Monitoring agent	Prometheus server	Stop monitoring a component
HLO	Monitoring Manager	Stop monitoring a component and delete Thanos stored data
Monitoring Manager	Thanos	Stop monitoring a component
Data Analytics Engine	Monitoring Manager	Request Data
Monitoring Manager	Data Analytics Engine	Scrape forecasted data

Figure 3 shows the communications of the Monitoring Manager with the rest of the monitoring framework and the interactions with other components of the CHARITY architecture. Flows for changes in monitoring configuration start from the High Level Orchestrator. Two types of flows are considered, the first is the deployment flow, when the services are configured for the first time in the monitoring framework. The second one is on runtime, when the migration of a service requires to update the configuration of two Prometheus servers, the one that monitors the cluster that the service initially occupied and the Prometheus of the cluster to which the service has been migrated. The migration triggers are detailed in the Monitoring Data section.

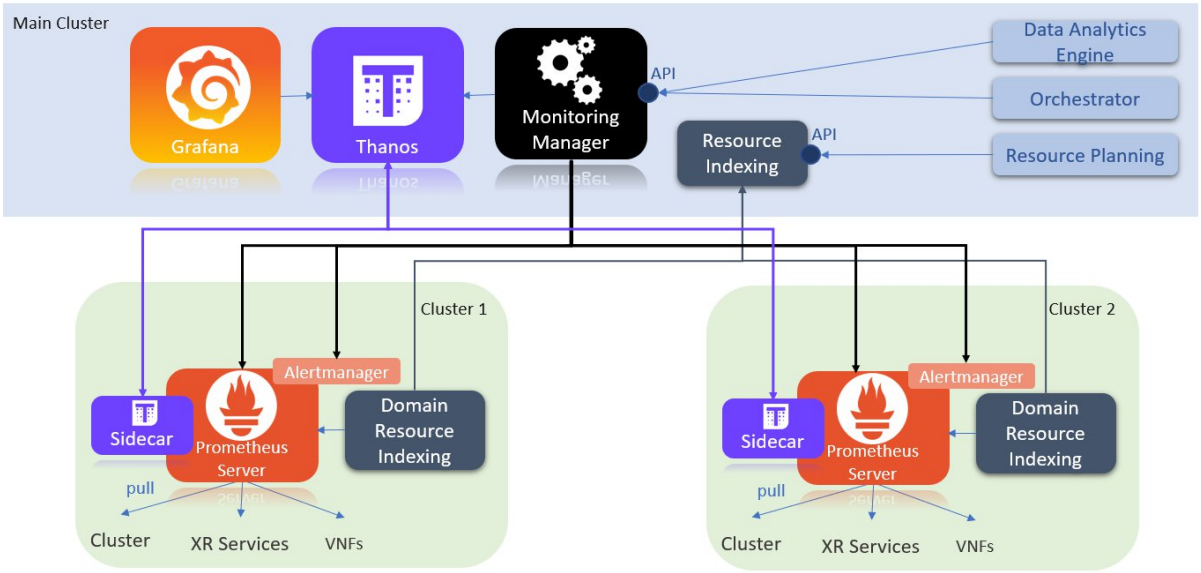


Figure 3: Monitoring Architecture defined in D2.1.

The Monitoring Manager has three functionalities:

- Manage request through a REST API.
- Download Prometheus servers configuration and update YAML configuration files.
- Trigger the configuration reload to apply the changes.

2.1.2 Monitoring Data

Prometheus servers deployed with each new cluster are in charge of monitoring VNFs, XR services, network performance between nodes and the available resources of the cluster itself. This observance of the environment is in CHARITY platform the trigger of the reactive flows that turn deployed application architectures into dynamic ones.

Three architecture components employ the monitoring data gathered:

- **HLO**: It uses cluster performance metrics to make decisions about deployments and migrations. This is described in Resource Indexing section, section XX.
- **Data Analytics Engine**: The monitored data of each metric feeds the forecasting instances, that predict future values.
- **Prometheus**: The monitoring servers allow to define alerting rules over the observed data, triggering its own alerting system to notify that a certain value has been reached.

Migration triggers: Alarms vs Alerts

In CHARITY project, we differentiate two types of performance notifications, that differ in the origin of the data, the components that provides that data, and what mitigates the migration they trigger, as stated in table Table 4. While the alarms use real data collected by the monitoring system, the alerts are based on predictions calculated by forecasting instances. It is necessary to differentiate the notifications (Figure 4) because one is triggered by the current performance of the monitored component, which means that the migration it triggers aims to correct a certain condition, exceeding a limit value established by the UC owner. However, in the case of the alerts, the data used are predictions, estimates of values that this metric will reach in a certain time. Therefore, the migration does not seek to reverse an existing situation, but rather to prevent it from occurring.

Table 4: Migration triggers

	Data Type	Data provider	Migration
Alarms	Monitored	Prometheus	Fix a performance failure
Alerts	Predicted	Data Analytics Engine	Avoid a performance failure

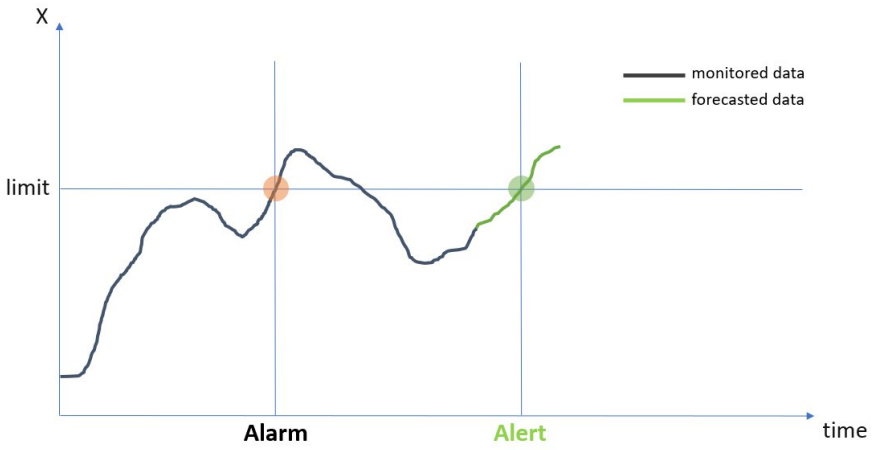


Figure 4: Alarm and alert notifications.

The use of two types of notifications allows CHARITY platform to use them as a verification factor to validate one of the objectives of CHARITY project, anticipation of performance failures. This is key for XR applications, that in today's highly competitive market must guarantee to their users an high quality graphics and smooth interactions. Therefore, the metric thresholds established in the AMF are the values that the UC considers critical and that overcoming them means entering a range in which a failure of services could occur, and therefore reduce the user's QoE.

The correct functioning of the alarm system ensures that predictions allow us to anticipate anomalies in the user experience migrating the services so that they always have the necessary resources for their correct functioning in the cluster in which they are deployed. This situation will occur if in no case the limits are reached, and therefore no alarm is activated.

2.1.3 UC metrics

The analysis of each UC and the KPIs collected in deliverable D1.3 allows to define the preliminary set of values to be monitored according to the needs of each UC owner. These metrics and their formats will be consulted and processed by CHARITY components, hence, it is necessary to establish common values, as stated in Table 5.

Table 5: Metrics definition

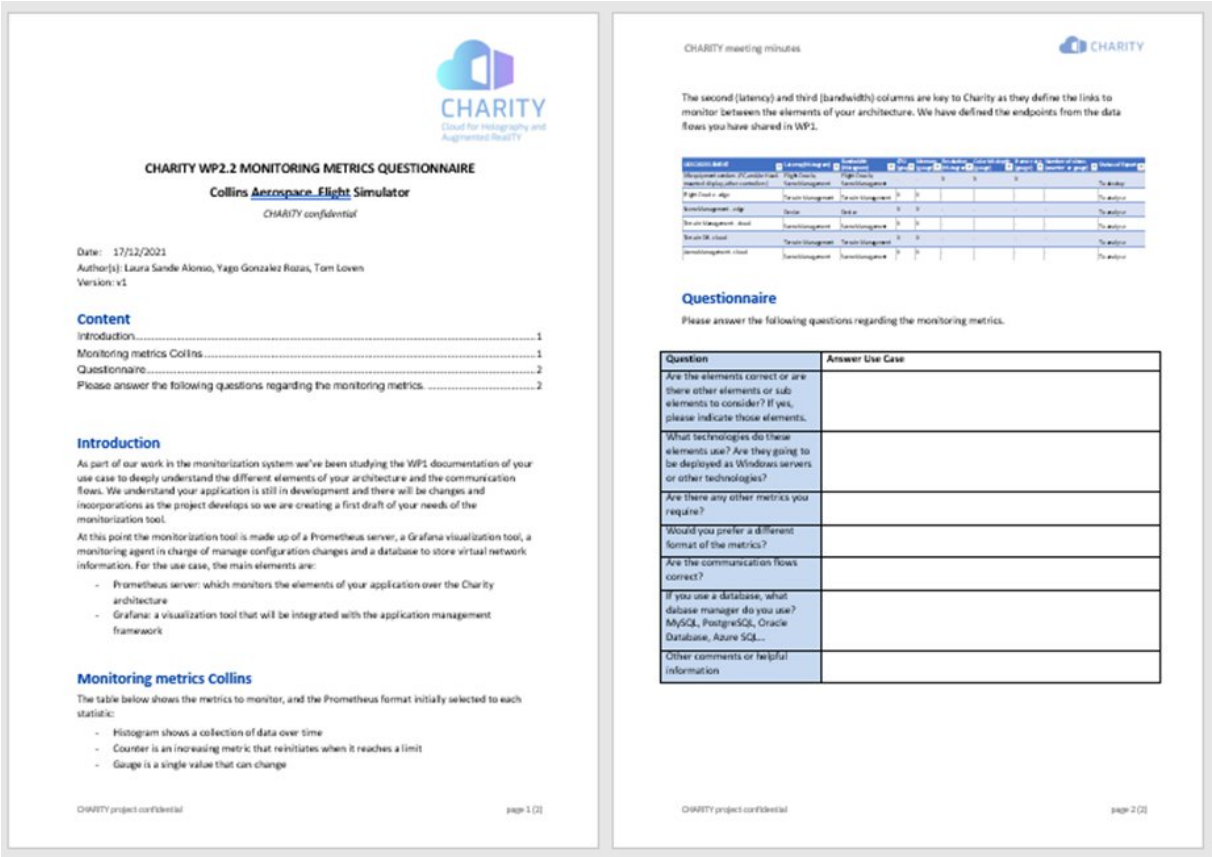
METRICS	DEFINITION	OUTPUT NAME	OUTPUT UNITS	FORMAT	EXAMPLE
Latency	Time it takes for a request to reach the destination and return, including the operation time of the destination to respond to the request	latency	miliseconds	three decimals	125.123
RTT	Round trip time. Time it takes for a request to reach the destination and return. It doesn't include the operation time of the destination to respond to the request	rtt	miliseconds	three decimals	125.123
Bandwidth	Maximum capacity that can be transmitted over a link	bw	Mbps	three decimals	1000.000
CPU	Percentage of used CPU	cpu	percentage	positive integer	0-100
GPU	Percentage of used GPU	gpu	percentage	positive integer	0-100
Memory	Percentage of used memory	memory	percentage	positive integer	0-100
Resolution	Number of pixels a screen is capable of displaying	resolution	Megapixel	three decimals	4.096
Color bit depth	Number of bits needed to represent the color of a pixel	colorbitdepth	bits per pixel	positive integer	24
Frame-rate	Frequency at which a device displays images	framerate	Frames per second - fps	positive integer	240
Petitions per second	Number of requests per second	petitionspersecond	Requests per second	positive integer	1000

A preliminary collection of the monitoring requirements of the use cases, from the performance of each of the components to the performance of the links between them opened communications with each UC to find out their preliminary needs. Ad-hoc surveys were created. These surveys contained a table of metrics that affected the case and a series of questions with which to delve into the types of data they need and the technologies of the elements they are developing (see an example in Figure 5).

The results of these surveys allowed to convert the requirements into a list of values to be monitored, with already defined formats. This allows also to design the exporters that will expose the data collected by the Prometheus server, the core of the monitoring system. The Prometheus server pulls

metrics from elements monitored through the HTTP endpoint each one uses to communicate. To expose these metrics, the elements use *exporters*, which collect the monitoring information, convert it to the format used by Prometheus and expose it to the outside.

The extensive use of Prometheus benefits from the existence of a community that maintains numerous exporters developed by third parties, which are already identified in the tables in the following sections focused on each use case. However, XR applications involve the appearance of new elements that require the development of new custom exporters, and for such task Prometheus offers detailed documentation and compatibility with the most common programming languages. Therefore, the collection of information, made through a questionnaire (as in Figure 5), needs to be made prior to the development of the monitoring system is a key step for the following phases, since it allows efforts to be focused on understanding the elements, their languages and the need or not to develop custom exporters, that can be similar between different use cases.



CHARITY

CHARITY WP2.2 MONITORING METRICS QUESTIONNAIRE

Collins Aerospace Flight Simulator

CHARITY confidential

Date: 17/12/2021
Author(s): Laura Sande Alonso, Yago Gonzalez Rozas, Tom Loven
Version: v1

Content

Introduction 1
Monitoring metrics Collins 1
Questionnaire 2
Please answer the following questions regarding the monitoring metrics. 2

Introduction

As part of our work in the monitorization system we've been studying the WP1 documentation of your use case to deeply understand the different elements of your architecture and the communication flows. We understand your application is still in development and there will be changes and incorporations as the project develops so we are creating a first draft of your needs of the monitorization tool.

At this point the monitorization tool is made up of a Prometheus server, a Grafana visualization tool, a monitoring agent in charge of manage configuration changes and a database to store virtual network information. For the use case, the main elements are:

- Prometheus server: which monitors the elements of your application over the Charity architecture
- Grafana: a visualization tool that will be integrated with the application management framework

Monitoring metrics Collins

The table below shows the metrics to monitor, and the Prometheus format initially selected to each statistic:

- Histogram shows a collection of data over time
- Counter is an increasing metric that reinitiates when it reaches a limit
- Gauge is a single value that can change

CHARITY project confidential page 1 (2)

CHARITY meeting minutes

CHARITY

The second (latency) and third (bandwidth) columns are key to Charity as they define the links to monitor between the elements of your architecture. We have defined the endpoints from the data flows you have shared in WP1.

Questionnaire

Please answer the following questions regarding the monitoring metrics.

Question	Answer Use Case
Are the elements correct or are there other elements or sub elements to consider? If yes, please indicate those elements.	
What technologies do these elements use? Are they going to be deployed as Windows servers or other technologies?	
Are there any other metrics you require?	
Would you prefer a different format of the metrics?	
Are the communication flows correct?	
If you use a database, what dabase manager do you use? MySQL, PostgreSQL, Oracle Database, Azure SQL...	
Other comments or helpful information	

CHARITY project confidential page 2 (2)

Figure 5: Questionnaire for Collins Aerospace.

Prometheus allows the use of four metric formats, two of them for individual values and the other two for storing a set of values during a certain period. *Counter* is an integer value that is incremented by one or reset to zero, while *gauge* allows the numeric value to increment and decrement. *Histogram* allows to collect values between certain margins over a period of time to later perform statistical analysis. The use of *summary* is similar to that of histogram, with the difference that it does not require a bucket definition, so it allows obtaining frequencies of more adjusted values than those of a histogram. In the questionnaires made to the UCs, these four possibilities were offered for the values to be monitored and they were asked to choose the formats according to the needs of each of their elements. In the following tables they will be defined as Counter -C-, Gauge -G-, Histogram -H- and Summary -S-.

The information discussed in this section serves as background for the following sections, which include the tables with the monitoring needs of both use cases and CHARITY own architecture, compiling metrics, formats and the existence of exporters already developed that expose the data.

2.1.3.1 UC1-1 - UC1-2: HOLO 3D - Holographic concert and holographic meeting

In the case of the holographic systems for concerts and meetings devised by Holo3D, we find key metrics related to image quality, as well as the performance of communications to eliminate delays and offer a real-time experience.

Table 6: UC1-1 and UC1-2 - elements and metrics

USE CASE ELEMENT	Links	Latency	RTT	Bandwidth	CPU	GPU	Memory	Resolution	Color bit depth	Frame-rate	Petitions per second	Exporter	Third party exporter
Musician (PC with camera, microphone)	Charity edge	HG	HG	HG	-	-	-	H	G	**G**	-	CUSTOM	-
Client (person watching the hologram on a holographic display)	Charity edge	HG	HG	HG	-	-	-	H	G	**G**	-	CUSTOM	-
Windows server	X	HG	HG	HG	G	-	G	-	-	-	-	EXISTING	Windows server
Speaker (PC with camera and microphone)	Charity edge	HG	HG	HG	-	-	-	H	G	**G**	-	CUSTOM	-
Client (person watching the hologram on a holographic display)	Charity edge	HG	HG	HG	-	-	-	H	G	**G**	-	CUSTOM	-
Windows server	X	HG	HG	HG	G	-	G	-	-	-	-	EXISTING	Windows server

2.1.3.2 UC1-3: SRT - Holographic assistant

The elements of the SRT holographic assistant are developed on Windows servers, which already have existing exporters to expose data in Prometheus format. The only custom exporter to be created is the one that involves the end user of the application. The creation of this type of exporter is common to all use cases, since Prometheus cannot monitor screens, virtual reality headsets or cockpits. Its monitoring will be carried out on another element of the architecture of the use case that communicates with these final elements.

Table 7: UC1-3 - elements and metrics

USE CASE ELEMENT	Links	Latency	RTT	Bandwidth	CPU	GPU	Memory	Resolution	Color bit depth	Frame-rate	Petitions per second	Exporter	Third party exporter
PC with holographic 3D device and Eyetracker	Charity edge	HG	HG	HG	-	-	-	H	G	**G**	-	CUSTOM	-
Windows server	X	HG	HG	HG	G	-	G	-	-	-	-	EXISTING	Windows server
SRT_SW_CLIENT	SRT_SW_CONTENT	HG	HG	HG	G	-	G	H	G	**G**	-	EXISTING	Windows server
SRT_SW_CONTENT	SRT_SW_BEHAVIOUR, SRT_SW_PCGEN	HG	HG	HG	G	G	G	-	-	-	-	EXISTING	Windows server

SRT_SW_PCGEN	CHARITY_SW_PCENC	HG	HG	HG	G	G	G	-	-	-	-	EXISTING	Windows server
CHARITY_SW_PCENC	SRT_SW_CLIENT	HG	HG	HG	G	G	G	-	-	-	-	EXISTING	Windows server
SRT_SW_BEHAVIOUR	Google API	HG	HG	HG	G	-	G	-	-	-	-	EXISTING	Windows server

2.1.3.3 UC2-1: ORAMA - Medical training

Surgical training through virtual/extended reality implies high synchronization between all the participants in the session to accurately simulate the collaborative work that takes place in an operating room between the end users of the application and the responses of the virtual elements to collisions with users.

Table 8: UC2-1 - elements and metrics.

USE CASE ELEMENT	Links	Latency	RTT	Bandwidth	CPU	GPU	Memory	Resolution	Color bit depth	Frame-rate	Petitions per second	Exporter	Third party exporter
VR equipment	Charity edge	HG	HG	HG	-	-	-	H	G	G	-	CUSTOM	-
LSpart_1	LSpart_2	HG	HG	HG	G	G	G	-	-	-	-	EXISTING	Windows VM
LSpart_2	LSpart_1	HG	HG	HG	G	G	G	-	-	-	-	EXISTING	Windows VM

2.1.3.4 UC2-2: DOTES - Virtual tours

Virtual tour applications are widely popular; however, they are still far from providing a realistic immersive user experience as they don't focus on the limitations that the network imposes on application performance. To achieve the quality of the image and interaction with the scenarios that DOTES, UC owner, plans with its application, it's essential that the communication speeds of the network and the processing of the elements adjust to their maximum performance.

Table 9: UC2-2 - elements and metrics.

USE CASE ELEMENT	Links	Latency	RTT	Bandwidth	CPU	GPU	Memory	Resolution	Color bit depth	Frame-rate	Petitions per second	Exporter	Third party exporter
Cyango Story - front-end	Cyango API Livestream service	HG	HG	HG	-	G	-	H	G	**G**	-	EXISTING	Nginx server
Cyango Cloud Studio - front-end	Cyango API	HG	HG	HG	-	G	-	H	G	**G**	-	EXISTING	Nginx server
Charity media converter	Cyango API	HG	HG	HG	G	G	G	-	-	-	-	CUSTOM	-
Cyango API	3D engine Cyango Story Cyango Cloud Editor File hosting Transcribe service Cyango Database 3dEngine Image engine	HG	HG	HG	G	-	G	-	-	-	-	EXISTING	Nginx server

File Hosting	Cyango API Video Engine	HG	HG	HG	G	-	G	-	-	-	-	EXISTING	AWS S3
3D engine	Cyango front-end Cyango API	HG	HG	HG	G	G	G	-	-	-	-	EXISTING	Nginx server
Image Engine (Image processor)	Cyango API	HG	HG	HG		G					-	EXISTING	Nginx server
Video Engine (replace by charity media converter)	File hosting Cyango API	HG	HG	HG		G					-	EXISTING	Now: AWS Lambda Future: nginx server
Livestream service	Cyango Story Cyango API	HG	HG	HG		G					-	EXISTING	Nginx server
Cyango Database	Cyango API	HG	HG	HG		-					-	EXISTING	Now: mongoDB Future: nginx server
Transcribe Service	Cyango API	HG	HG	HG		-					-	EXISTING	Nginx server

2.1.3.5 UC3-1: ORBK - Mixed reality

The extended reality application devised by ORBK focuses on the user's interaction with the virtualized scenario, the virtual objects introduced and that all this happens with the minimum delay between all the users of that application session. The mesh collider that is being developed in the CHARITY project is key to the performance between real image and virtualized elements, and to achieve the KPIs of the next generation XR applications, a proactive adaptive architecture like CHARITY is needed.

Table 10: UC3-1 - elements and metrics.

USE CASE ELEMENT	Links	Latency	RTT	Bandwidth	CPU	GPU	Memory	Resolution	Color bit depth	Frame-rate	Petitions per second	Exporter	Third party exporter
Game client	Game Server	HG	**HG**	HG	-	-	-	H	**G**	G	-	CUSTOM	-
Game Server	Game client, Mesh collider, Game Servers Status DB	HG	**HG**	HG	G	-	G	-	**-**	-	-	CUSTOM	-
Game Servers Status DB	Game Server	HG	**HG**	HG	G	-	G	-	**-**	-	-	EXISTING	CloudWatch
Specialized XR Service by Charity (Mesh collider generator service)	Game Server	HG	**HG**	HG	G	G	G	-	**-**	-	-	CUSTOM	-
Mesh Merging Service - developed by CNR	Game Server	HG	**HG**	HG	-	G	-	-	**-**	-	-	CUSTOM	-

2.1.3.6 UC3-2: Collins Aerospace (CAI) - Flight simulator

To date, the simulation of high-speed scenarios has been limiting in terms of collaborative applications due to the difficulties of synchronization between users and the performance of the different microservices in charge of predicting the images to be displayed in the participants. In the case of

Collins Aerospace, an edge architecture is proposed to reduce interaction times and control over the requests received by each element to show the highest image quality to always maintain synchronization between its users.

Table 11: UC3-2 - elements and metrics.

USE CASE ELEMENT	Links	Latency	RTT	Bandwidth	CPU	GPU	Memory	Resolution	Color bit depth	Frame-rate	Petitions per second	Exporter	Third party exporter
Cockpit (flight stick, thrustor, pedals)	Flight Oracle, Scene Management	HG	HG	HG	-	-	-	H	**G**	G	-	CUSTOM	-
Flight Oracle	Terrain Management	HG	HG	HG	G	-	G	-	-	-	-	CUSTOM	-
Scene Management	Device	HG	HG	HG	G	G	G	-	-	-	-	CUSTOM	-
Terrain Management	Scene Management	HG	HG	HG	G	G	G	-	-	-	-	CUSTOM	-
Terrain DB	Terrain Management, Image Generator	HG	HG	HG	G	-	G	-	-	-	-	EXISTING	PostgreSQL
Arena Management	Scene Management	HG	HG	HG	G	-	G	-	-	-	-	CUSTOM	-
Flight Dynamics/Physics?	Flight Oracle, Cockpit, View Builder	HG	HG	HG	G	G	G	-	-	-	-	CUSTOM	-
Image Generator	Flight Oracle, Terrain DB	HG	HG	HG	G	G	G	-	-	-	-	CUSTOM	-
Frame Caché	View Builder, Resolution Upscaler, Image Generator	HG	HG	HG	G	G	G	-	-	-	-	EXISTING	Redis
View Builder	Flight Dynamics, Frame caché, WARP, web RTC Client	HG	HG	HG	G	G	G	-	-	-	-	CUSTOM	-
WARP	View Builder	HG	HG	HG	G	G	G	-	-	-	-	CUSTOM	-
web RTC Client	PC-HMD	HG	HG	HG	G	G	G	-	-	-	-	CUSTOM	-
PC, HMD	webRTC	HG	HG	HG	G	-	G	H	-	G	-	CUSTOM	-
Resolution Upscaler	Frame Caché	HG	HG	HG	G	G	G	-	-	-	HG	CUSTOM	-

2.2 Resource Indexing

The Resource Indexing collects performance related cluster metrics and makes it available for the HLO, which is responsible for choosing the most suitable cluster based on the resources required by each new deployment or for migration to a new cluster.

The Resource Indexing is made up of instances per cluster, as shown in Figure 6, that collect data from the cluster in which they are deployed. These instances communicate with the main Resource Indexing, an architectural component with information on all clusters available in all domain and for all providers.

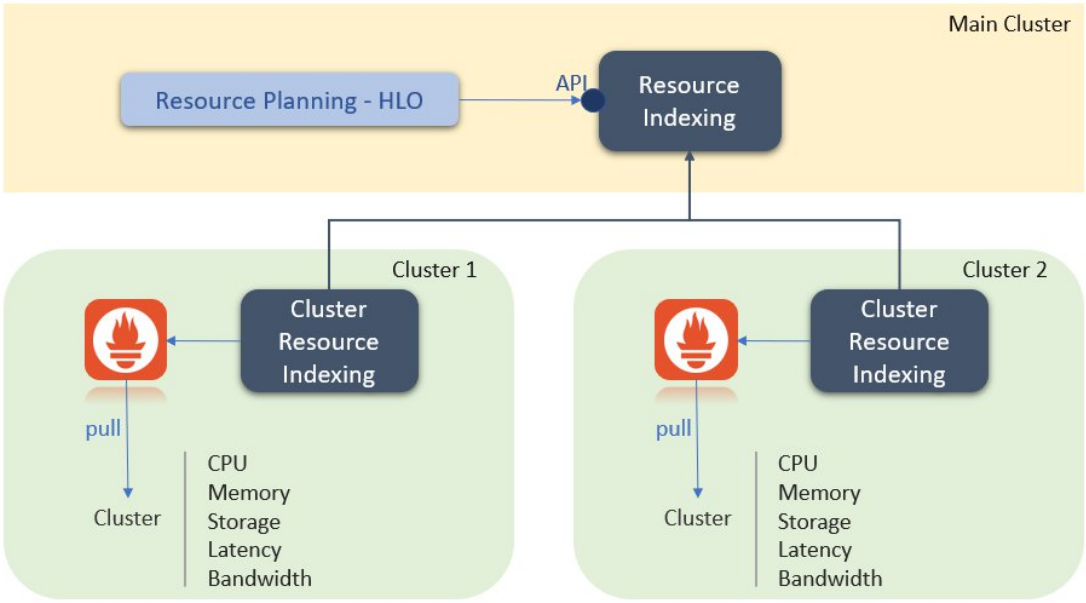


Figure 6: Resource Indexing components.

Each Cluster Resource Indexing is made up of a database, where the last value of each of the metrics is stored, and two update tools. The first one makes periodic HTTP requests to Prometheus with queries in PromQL format, processable by the monitoring server, and stores it in the database. The second one updates periodically the main Resource Indexing.

The cluster metrics the HLO needs to evaluate to take deployment decisions are: CPU, memory, storage, bandwidth between clusters and latency between clusters. The first prototype of the Resource Indexing collects the first three. The table 12 collects the Prometheus metrics used by the Resource Indexing prototype. Network performance metrics will be incorporated in the coming months.

Table 12 - Resource Indexing prototype metrics.

Metric		Prometheus Metric	Prometheus Query
CPU	total	machine_cpu_cores	sum(machine_cpu_cores)
	used	container_cpu_usage_seconds_total	sum(rate(container_cpu_usage_seconds_total))
	available	machine_cpu_cores container_cpu_usage_seconds_total	sum(machine_cpu_cores)- sum(rate(container_cpu_usage_seconds_total))
Memory	total	machine_memory_bytes	sum(machine_memory_bytes)
	used	container_memory_working_set_bytes	sum(container_memory_working_set_bytes)
	available	machine_memory_bytes container_memory_working_set_bytes	sum(machine_memory_bytes)- sum(container_memory_working_set_bytes)
Storage	total	container_fs_limit_bytes	sum(container_fs_limit_bytes)
	used	container_fs_usage_bytes	sum(container_fs_usage_bytes)
	available	container_fs_limit_bytes container_fs_usage_bytes	sum(container_fs_limit_bytes)- sum(container_fs_usage_bytes)

The communication between the HLO and the Resource Indexing is through a REST API, where two types of queries are defined, according to the needs of the HLO. The first one returns the values of all

clusters and all domains, while in the second the values returned are from the clusters belonging to the domain specified in the input parameters.

Some basic experimentation has been done regarding the communication between a main cluster and a couple of standalone clusters to check if all of the tools needed for the Resource Indexing work as expected. To test it, we deployed each standalone cluster with a subset of a couple of servers, and adapted the Prometheus tool to obtain the necessary information. Then, we tested the Cluster Resource Indexing, first to check if the information was stored correctly in the database, and then to see if this information was sent to the Main Cluster, in which we checked that the information was correctly received. Finally, we simulated the HLO calls to the Resource Indexing through the Rest Api, particularly the ones related to availability, to check if the obtained data was useful to the HLO.

3 CHARITY Edge Storage (CHES)

3.1 Component descriptions

The CHARITY Edge Storage Component (CHES) is responsible for providing optimized edge storage services to the CHARITY framework and its hosted applications. These services include data storage, retrieval and migration tasks, security and privacy protection capabilities, QoS and QoE violation prevention and mitigation, as well as other data-related services that serve the runtime requirements of CHARITY. More specifically, the edge storage component has to provide a reliable, fast, stable and secure shared storage engine, accessible by all devices and users in an edge-cloud. Furthermore, it needs to be extremely lightweight since it is created for edge devices with extremely limited resources, like Raspberry Pies or other micro-computer devices.

Edge nodes generally have limited computation, storage, network, or power resources. The distributed, dynamic and heterogeneous environment in the edge and the diverse application's requirements pose several challenges. The edge storage component needs to overcome some inherent edge challenges like:

- Coordination of unreliable devices and network
- Hardware and software incompatibilities that arise due to the plethora of different devices
- Integration of different data storage formats and data types
- Limited resources of the edge devices
- Security and privacy concerns
- QoE insurance

CHES component is based on the Kubernetes (K8s)[3], MinIO[4] and Prometheus[5] technologies, combining and optimizing them in order to better serve the needs of CHARITY. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. As a storage solution, an open-source framework created by IBM is utilized, called MinIO. MinIO is an inherently decentralized and highly scalable Peer-to-Peer solution, allowing us to deploy it freely on usable nodes. It is designed to be cloud native and can run as lightweight containers managed by external orchestration services such as Kubernetes. It supports a hierarchical structure to form federations of clusters and it has been proven as a valid candidate for an edge data storage system [1]. MinIO writes data and metadata together as objects, eliminating the need for a metadata database. In addition, MinIO performs all functions (erasure code, bitrot check, encryption) as inline, strictly consistent operations. The result is that MinIO is exceptionally resilient. Moreover, it uses object storage over block storage so it is in fact a combination of the two systems, preserving the lightweight distributed nature of block storage while providing the plethora of metadata and easy usage of the object storage. Unlike other object storage solutions that are built for archival use cases only, the MinIO platform is designed to deliver the high-performance object storage that is required by modern big data applications. In addition, MinIO provides both a web-based GUI and an AWS S3 compatible API library. The Kubernetes Dataset Lifecycle Framework provided by IBM's Datashim[6] is employed on top of MinIO, allowing the edge storage component to be used as a file system folder, which is useful for applications that we cannot or do not want to integrate with the Restful API of MinIO. A detailed description of the Kubernetes Dataset Lifecycle Framework is provided in Section 3.1.1. Finally, Prometheus is responsible for collecting monitoring data about the real-time performance of the nodes and the component as a whole to analyze the behaviour of different applications and optimize the cluster architecture, the options, and the data distribution.

[3] https://kubernetes.io/

[4] https://min.io/

[5] https://prometheus.io/

[6] https://datashim.io/

Additionally, a sub-component, called CHES-Registry, was implemented using CHES as its file storage backend, to move application images closer to the edge and limit network traffic and delays during the operations of deployment, migration or scaling. CHES Registry hosts the Docker images and employs Kubernetes containerization to provide its services, creating a new pod in the CHES namespace that is able to connect to the MinIO storage backend. In addition, CHES Registry allows the secure communication between the registry and its clients using the HTTPS protocol and a basic authentication scheme.

3.1.1 Kubernetes Dataset Lifecycle Framework

Hybrid edge/cloud environment is rapidly becoming the new trend for organizations seeking the perfect mix of scalability, performance and security. As a result, it is now common for an organization to rely on a mix of on-premises data centers (private cloud), and cloud/edge solutions from different providers to store and manage their data. Nevertheless, many obstacles arise when applications have to access the data. On the one hand, developers need to know the exact location of the data and, on the other hand, manage the correct credentials to access the specified data-sources holding their data. In addition, access to cloud/edge storage is often completely transparent from the cloud management standpoint and it is difficult for infrastructure administrators to monitor which containers have access to which cloud storage solution. Even if containerized components and micro-services are widely promoted as the appropriate solution for efficiently deploying and managing storage over a hybrid edge/cloud infrastructure, containerization makes it more difficult for the workloads to access the shared file systems. Currently, there are no established resource types to represent the concept of data-source on Kubernetes. As more and more applications are running on Kubernetes for batch processing, end users are burdened with configuring and optimizing the data access [2].

To tackle the aforementioned issues, the Dataset Lifecycle Framework (DLF) is employed, which is an open-source project that enables transparent and automated access for containerized applications to data-sources. DLF enables users to access remote data-sources via a mount-point within their containerized workloads and it is aimed to improve usability, security, and performance, providing a higher level of abstraction for dynamic provisioning of storage for the users' applications. By integrating DLF on Kubernetes pipelines, it is possible to mount object stores as Persistent Volume Claims (PVCs), which are pieces of storage in the cluster, and present them to pipelines as a POSIX-like file system. In addition, DLF makes use of Kubernetes access control and secret so that pipelines do not need to be run with escalated privilege or to handle secret keys, thus making the platform more secure.

In more technical details, DLF orchestrates the provisioning of PVCs required for each data-source, which users can refer to their pods (the smallest deployable unit in Kubernetes), allowing them to focus on the actual workload development rather than configuring/mounting/tuning the data access. DLF is designed to be cloud-agnostic and due to Container Storage Interface (CSI)[7], it is highly extensible to support various data-sources. CSI is a standard for exposing arbitrary block and file storage systems to containerized workloads on Container Orchestration Systems (COS) like Kubernetes. With the adoption of COS, the Kubernetes volume layer becomes truly extensible. Using CSI, third-party storage providers are able to write and deploy plugins exposing new storage systems in Kubernetes without interacting or changing the core Kubernetes code. This provides Kubernetes users more options for storage and makes the system more secure and reliable. On the infrastructure side, DLF also enables cluster administrators to easily monitor, control, and audit data access.

DLF introduces the Dataset as a Custom Resource Definition (CRD)[8], which is a pointer to existing S3 or NFS data-sources. A Dataset object is a reference to a storage provided by a cloud-based storage solution, potentially populated with pre-existing data. In other words, each Dataset is a pointer to an

[7] https://kubernetes-csi.github.io/docs/

[8] https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions

existing remote data source and is materialized as a PVC. The Dataset is a declarative construct that abstracts access information and provides a single reference for data in Kubernetes. Users only need to include this reference in their deployments to make the data available in pods, either through the file system or through environment variables [3].

Figure 7 illustrates an example configuration of a Dataset CRD for data stored in COS. The mandatory fields are the *bucket*, *endpoint*, *accessKeyID*, and *secretAccessKey*. The *bucket* entry creates a one-to-one mapping relationship between a Dataset object and a bucket in the COS. The *accessKeyID* and *secretAccessKey* fields refer to the credentials used to access this specific bucket.
DLF is completely agnostic to where/how a specific Dataset is stored, as long as the *endpoint* is accessible by the nodes within the Kubernetes cluster, in which the framework is deployed.

```
apiVersion: com.ie.ibm.hpsys/v1alpha1
kind: Dataset
metadata:
  name: example-dataset
spec:
  local:
    type: "COS"
    accessKeyID: "{ACCESS_KEY_ID}"
    secretAccessKey: "{SECRET_ACCESS_KEY}"
    endpoint: "{S3_SERVICE_URL}"
    bucket: "{BUCKET_NAME}"
    readonly: "false"
```

Figure 7: Dataset CRD.

Creating a CRD is just the first step to add custom logic in the Kubernetes cluster. The next step is to create a component that has embedded the domain-specific application logic for the CRD. Essentially, a service provider needs to develop and install a component which reacts to the various events which are part of the lifecycle of a CRD and implements the desired functionality.

DLF utilizes the Operator-SDK, an open-source component of the Operator Framework[9], which provides the necessary tooling and automation in the development of these components in an effective, automated, and scalable way. Operator-SDK is utilized to create the Dataset Operator in DLF. Its main functionality is to react to the creation (or the deletion) of a new Dataset and materialize the specific object. Specifically, when a Dataset gets created, the software stack invokes the necessary Kubernetes CSI plugin and creates a PVC that provides a file system view of the bucket in the COS.

Figure 8 demonstrates in an abstract view, the Dataset Lifecycle Framework with the various components employed in an example of a two-node K8s cluster.

[9] https://operatorframework.io/

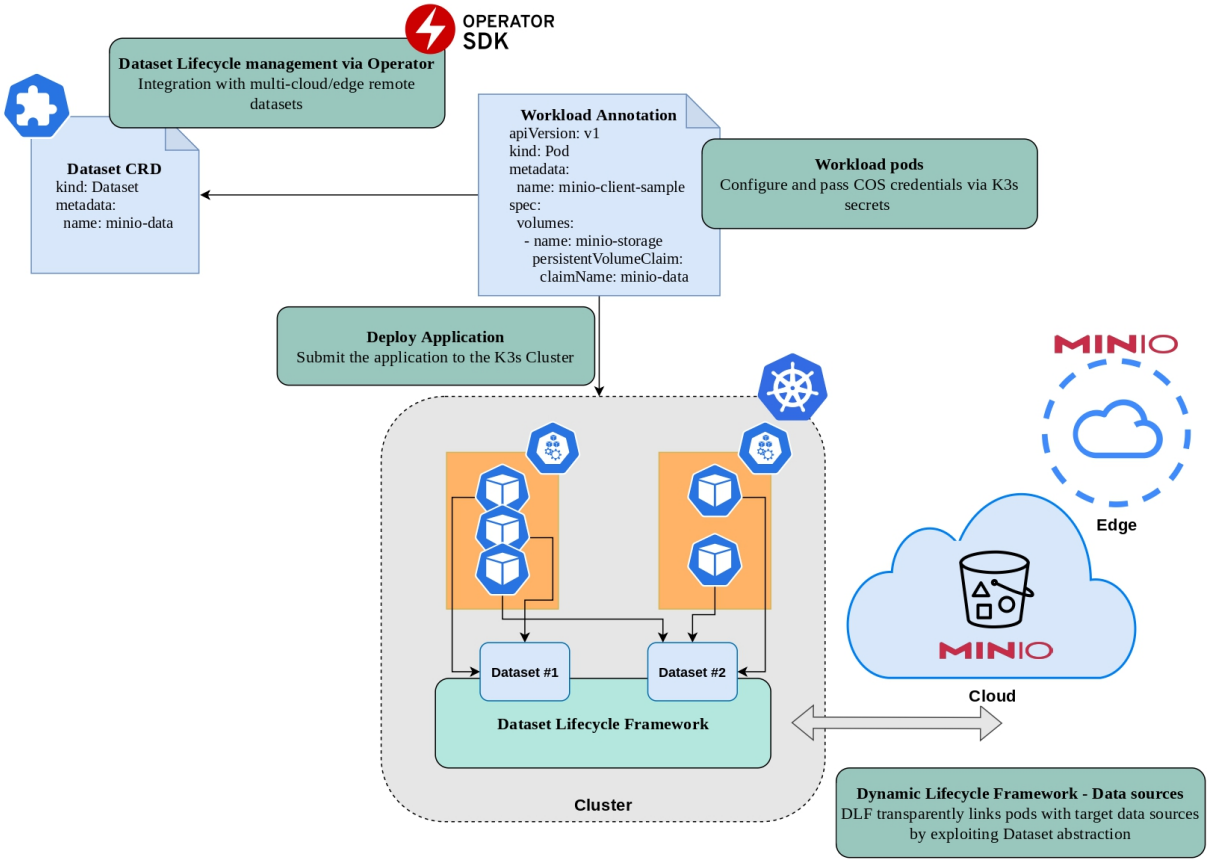


Figure 8: Conceptual overview of the Dataset Lifecycle Framework (DLF).

3.2 Package information

3.2.1 CHES Storage

CHES is a package including Kubernetes deployment files in YAML format, installation scripts in bash script format, and a configuration file in JSON format that contains all options needed to configure the component.

All files of the package are available on the official CHARITY GitLab page[10] and can be obtained with the following command:

```
$ git clone https://gitlab.charity-project.eu/hua/edgestoragecomponent.git
```

In detail, we have one YAML file called chesDeployment.yaml which is the Kubernetes deployment file for the storage server (master). This file will install all necessary services, authentication keys, roles and images on the Kubernetes cluster, reading information from the configuration file (.conf). It will use the Kubernetes architecture, deploying most services on the Kubernetes master. Of course, the actual MinIO instances that store the data will be deployed on the nodes having the label “ches-worker” set to “true”. The second yaml file is called chesClientDeployment.yaml and it will allow nodes to use CHES as a file system folder by mounting the PVC that is connected to the CHES storage service.

The bash scripts are again two, chesInstalldeploy.sh that configures and deploys the chesDeployment.yaml on the Kubernetes master, and chesClientDeploy.sh that configures and deploys the chesClientDeployment.yaml on the client nodes. These scripts are just applying the options selected in the configuration file to the YAML files and then run the necessary commands to deploy

[10] https://gitlab.charity-project.eu/hua/edgestoragecomponent

the YAML files on the Kubernetes cluster. There is a third bash script called InstallScript.sh which is configuring and deploying the chesDeployment.yaml file in a single K8s cluster node installation, without requiring any additional configuration steps.
Finally, a YAML file called dlf_kube.yaml is used for the deployment of the Dataset Lifecycle Framework and a bash script named Undeployches.sh which undeploys the CHES containers and jobs. A complete list of the files included is presented in Table 15.

Table 15: List of package files for Edge storage component.

Filename	Description
`chesDeployment.yaml`	Kubernetes deployment file for CHES master
`chesClientDeployment.yaml`	Kubernetes deployment file for CHES client(s)
`chesInstalldeploy.sh`	Bash script for deploying the CHES servers
`chesClientDeploy.sh`	Bash script for deploying the CHES client(s)
`InstallScript.sh`	Bash script for deploying the CHES servers on single node clusters
`configuration_file.conf`	JSON file containing the configuration options for CHES
`dlf_kube.yaml`	Kubernetes deployment file for the Dataset Lifecycle Framework
`Undeployches.sh`	Bash script for undeploying the CHES containers and jobs

3.2.1.1 Kubernetes Dashboard

Along with the CHES component, the Kubernetes dashboard is provided, which is a web-based Kubernetes user interface. In general, Kubernetes dashboard is used to deploy containerized applications to a Kubernetes cluster, troubleshoot the containerized applications, and manage the cluster resources. In addition, the dashboard can get an overview of applications running on a cluster, as well as for creating or modifying individual Kubernetes resources (such as Deployments, Jobs, DaemonSets, etc). Dashboard also provides information on the state of Kubernetes resources in the cluster and on any errors that may have occurred. The associated files are located in the same repository with CHES.

In detail, the installation of Kubernetes dashboard includes four files, two deployment YAML files and two bash scripts. A bash script named InstallDashboard.sh is used for deploying the Kubernetes dashboard in a K3s cluster. A complete list of the files included, is presented Table 14.

Table 16: List of files included in the Kubernetes Dashboard.

Filename	Description
`InstallDashboard.sh`	Bash script for deploying the Kubernetes Dashboard
`recommended.yaml`	Kubernetes deployment file for Kubernetes dashboard
`dashboard_account_roles.yaml`	Kubernetes deployment file for creating a minimal RBAC configuration, i.e. a Service Account and a ClusterRoleBinding
`UndeployDash.sh`	Bash script for undeploying the Kubernetes Dashboard

3.2.2 CHES Registry

CHES Registry is a sub-component that realizes a localized Docker image registry, taking Docker and VM images near the edge devices/nodes to support the faster application deploying and limit the network flooding caused by large application image downloads during deployment. This functionality acts as a proactive caching mechanism by optimizing the download delays and the network traffic. The port of the CHES Registry as well as its credentials are pre-configured using the generalized configuration file that is packed with the edge storage solution.

CHES Registry is based on the Docker registry technology in order to store and distribute container images. It combines the official Docker registry image[11] with Kubernetes orchestration, a MinIO object storage backend, and a set of automated deployment and configuration scripts. This enables CHES Registry to automatically deploy and scale the Docker registry as needed while centrally controlling the configuration options such as communication protocols, SSL certificates, credentials, connection ports and other. This configuration also enables us to fine-tune the back-end storage, placing the images at the optimal physical locations according to the needs of each use case. As a result, CHES Registry streamlines the storage and distribution of container images, offering enhanced control, scalability, and optimized edge deployment capabilities. The application images are handled as objects, stored in a MinIO bucket and accessed either using the S3 API it provides, its web interface or the DLF functionality the LDR has added on top of MinIO, making the buckets available as mountable virtual disks.

Figure 9 illustrates the CHES Registry sub-component. The associated files are separated into a different folder, in order to separate them by functionality, make documentation and maintenance easier and decouple their installation process.

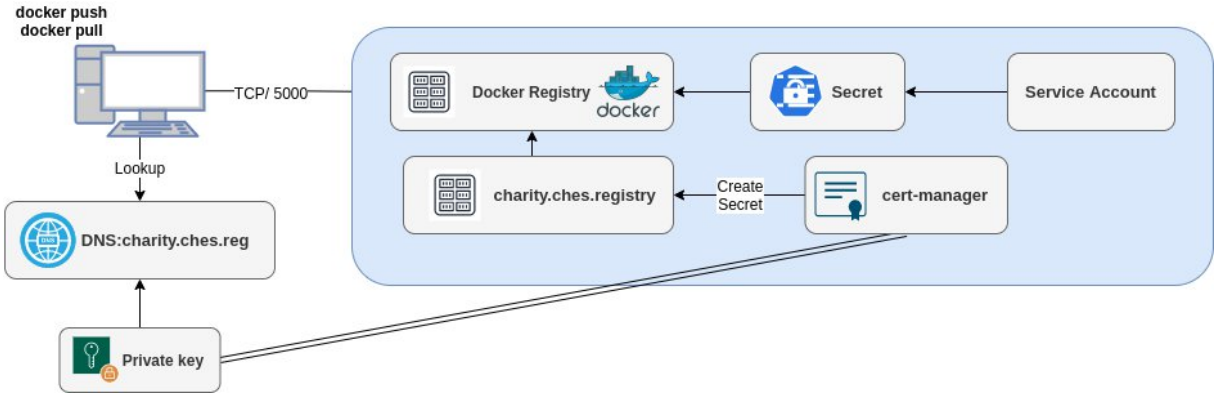


Figure 9: CHES Registry.

CHES Registry can be downloaded by running the command:

```
$ git clone https://gitlab.charity-project.eu/hua/edgestoragecomponent.git
```

In detail, the installation of CHES Registry includes six files, four deployment YAML files and two bash scripts. The YAML files are deploying all the necessary containers and jobs that need to be executed to setup and configure the registry, in order to be functional and accessible by other containers hosted in the same K8s cluster. A complete list of the files included is presented in Table 15 .

Table 17: List of files included in the CHES Registry repository.

Filename	Description

[11] https://hub.docker.com/_/registry

add_certs.yaml	Kubernetes deployment for a daemon job that adds the appropriate SSL certificates to new containers
add_to_hosts.yaml	Kubernetes deployment for a daemon job that adds the appropriate configurations to the hosts files of new containers
deployment.yaml	Kubernetes deployment for the Docker registry container
registry_setup.sh	Bash script for deploying the CHES LDR containers and jobs
registry_uninstall.sh	Bash script for undeploying the CHES LDR containers and jobs
test_deploy.yaml	Kubernetes deployment for a test container that loads a docker image from the deployed CHES LDR

3.2.3 Prometheus

Prometheus, as previously said, is a popular open-source monitoring and alerting tool that enables users to monitor the performance and health status of their systems and applications. It is specifically designed for highly dynamic environments, such as cloud-native applications and microservices architectures. Prometheus collects time-series data from various sources, including its own client libraries, exporters, and third-party integrations, and stores them in an efficient and scalable manner. The associated files for the Prometheus setup are located in the same repository with CHES.

The deployment of Prometheus includes one YAML file (prometheus.yaml) that configures the scraping job and the target. A scraping job refers to the process of periodically collecting metrics from a target using HTTP, allowing Prometheus to monitor and analyze the performance and health of that target over time. The target refers to the specific endpoint from which metrics are collected through scraping. This is achieved through a URL that exposes metrics in a format Prometheus understands.

3.2.4 Semi-automated Deployment and off-loading

In the context of the presented solution, a set of bash and YAML scripts have been developed that handle all the configuration, installation and deployment processes that need to be contacted before and after the MinIO workers are deployed. These configurations include firewall rules, DNS settings, package installations and security checks that take into account the setup environment, the architecture and resources of the physical machines and the software involved. These tasks enable the semi-automatic deployment of the edge storage solution, forming complex pipelines that in most other cases are performed manually by a system administrator. This ensures that scaling can be performed seamlessly on each cluster, regardless of the underlying physical machines that act as nodes. In addition, off-loading of data can be achieved by "ordering" more instances of the MinIO worker to be deployed on more nodes and adding them in the same MinIO cluster in real-time.

3.3 User Manual

3.3.1 CHES Storage

We have three ways to utilize CHES, the first way is through the MinIO Web GUI which is described in details on the official MinIO documentation[12]. A sample MinIO storage deployment can be seen in Figure 10.

[12] https://docs.min.io/docs/minio-quickstart-guide.html

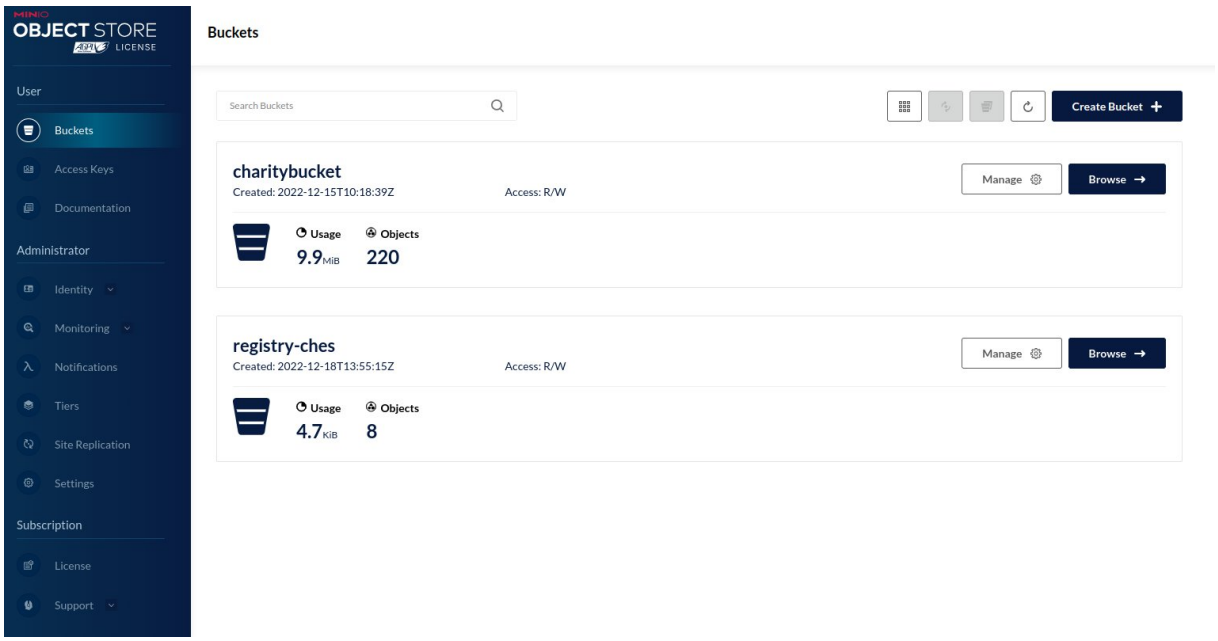


Figure 10: The MinIO web-based interface.

The second way is through the MinIO client which is a command line tool that is also documented in detail on the official MinIO website[13]. A connection to a remote host can be seen as an example in Figure 11.

```
antonis@antonis-dell:~$ mc alias set CHES_Storage "http://31.171.240.140:9011" "chesAccesskeyMinio" "chesSecretkey"
Added `CHES_Storage` successfully.
antonis@antonis-dell:~$ mc ls CHES_Storage
[2022-12-15 12:18:39 EET]     0B charitybucket/
[2022-12-18 15:55:15 EET]     0B registry-ches/
```

Figure 11: Connection to MinIO using the client command tool.

Additionally, using the integrated Datashim's DLF, CHES can be accessed through the K8s deployment manifest files by mounting the PVC it creates as a system volume. Detailed reference of the usage of PVCs can be found in the Kubernetes API documentation[14]. An example of the deployment manifest file is illustrated in Figure 12.

[13] https://docs.min.io/docs/minio-client-complete-guide.html

[14] https://kubernetes.io/docs/concepts/storage/persistent-volumes/

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ches-client-sample
spec:
  volumes:
    - name: ches-storage
      persistentVolumeClaim:
        claimName: ches-dataset
  containers:
    - name: ches-test
      image: nginx
      volumeMounts:
        - mountPath: "/data/ches"
          name: ches-storage
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: ches-worker
            operator: In
            values:
            - "true"
```

Figure 12: An example of mounting a PVC created by the Datashim integration if the PVC is called "ches-dataset".

The Kubernetes dashboard which is a web-based Kubernetes user interface is illustrated in Figure 13.

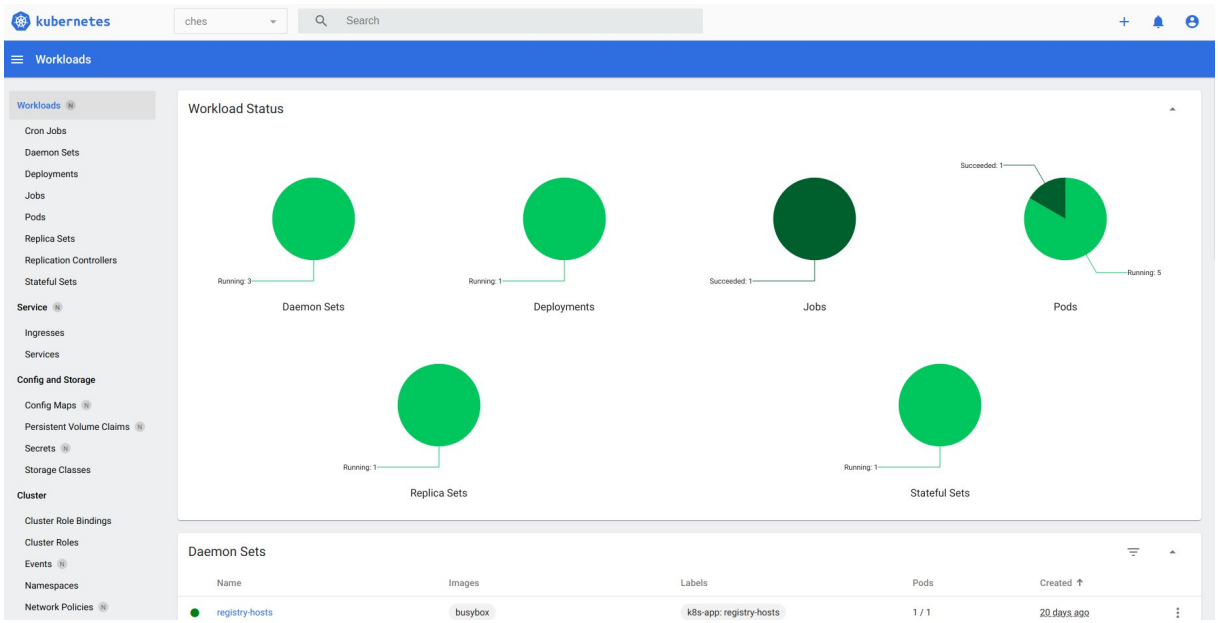


Figure 13: Kubernetes Dashboard.

To utilize Prometheus, it is necessary to configure the YAML file with the appropriate endpoint and target settings, as depicted in Figure 14.

```
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: "minio-job"
    metrics_path: /minio/v2/metrics/cluster
    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.
    static_configs:
    - targets: ['charityconsole.ches.charity-project.eu:9011']
```

Figure 14: Prometheus configuration YAML file.

Subsequently, the user can access the Prometheus console and utilise query commands to retrieve a multitude of metrics. As an example, Figure 15 exemplifies the outcomes generated by the *minio_bucket_objects_size_distribution* query.

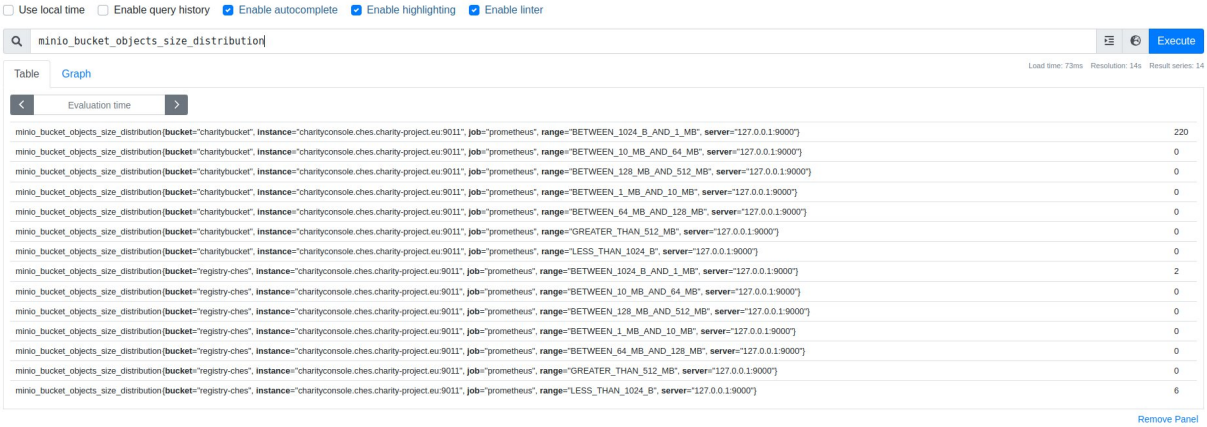


Figure 15: Prometheus console example metric.

Moreover, the integration between Prometheus and the MinIO console offers valuable information, as demonstrated in Figure 16.

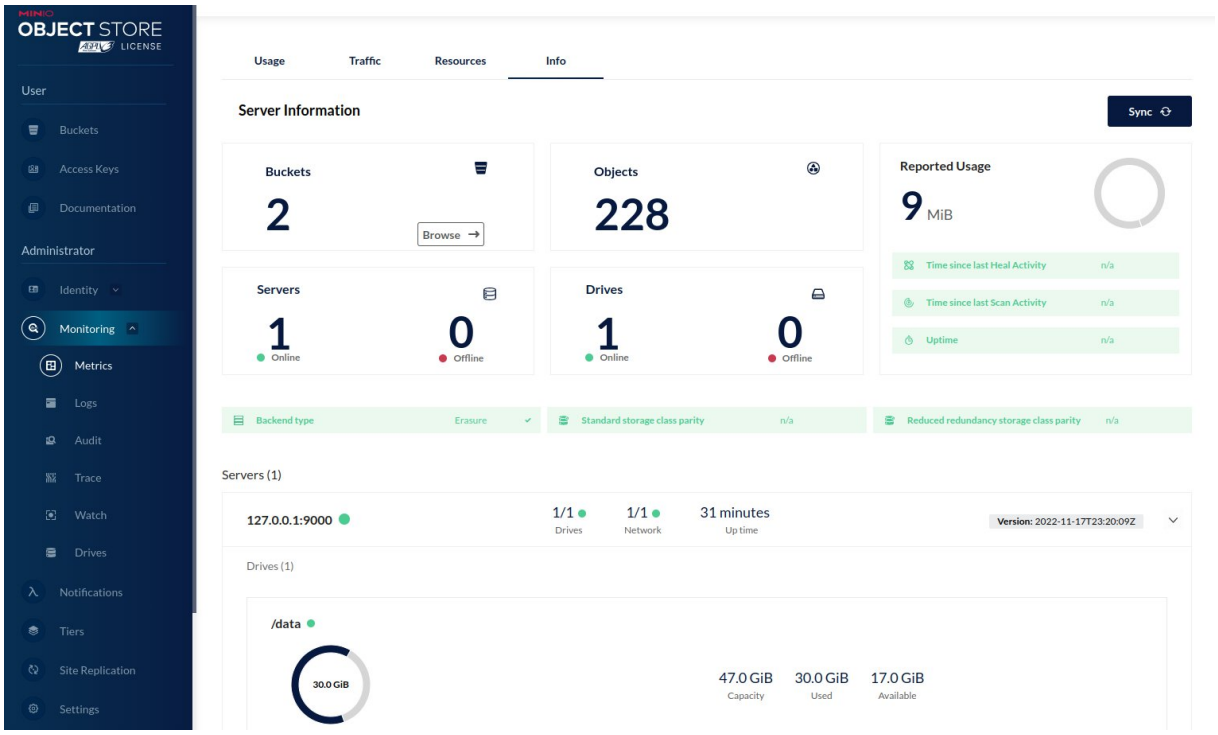


Figure 16: Prometheus-MinIO Console integration.

3.3.2 CHES Registry

CHES Registry can be accessed through the Docker Registry APIs. These APIs are described in the official Docker documentation[15]. Catalog API is the simplest of the APIs provided, displaying a list of the available images pushed in a registry. An example of the catalog API is illustrated in Figure 17. In this case it hosts an example *hello-world* image.

{"repositories":["hello-world"]}

Figure 17 Catalog API example.

Finally, a list of available images can be obtained from a terminal (using the curl utility), as illustrated in Figure 18.

```
cloudsigma@GVA2-CHA-SRVR01:~/edgestoragecomponent/CHES-LocalizedDockerRegistry$ curl https://charity.ches.registry:5045/v2/_catalog
{"repositories":["hello-world"]}
cloudsigma@GVA2-CHA-SRVR01:~/edgestoragecomponent/CHES-LocalizedDockerRegistry$
```

Figure 18: Example of the catalog API for CHES Registry hosted in a K8s cluster.

3.4 Licensing

This component, including all originally created source files, scripts and other resources are released as free software under the terms of the GNU General Public License version 3 or later, as published by the Free Software Foundation.

MinIO is provided under GNU Affero General Public License version 3 which enables us to use it as an open-source component providing that we also use a GNU public License.

Prometheus, Datashim and K8s are protected under Apache License which gives us full usability of their open-source components.

3.5 Results obtained in relation to the objectives (KPIs)

The work conducted in Task 3.2 aims in achieving the objectives along with the requirements and targeted KPIs. More specifically, the KPIs that will be met from Objective 2 (*Provide holistic support for the orchestration of advanced media solutions*) are:

- **KPI-2.2 Storage formats: at least one (block, file, object)**
  - As already mentioned, as a storage solution, an open-source framework created by IBM is utilized, called MinIO. This framework uses object storage over block storage so it is in fact a combination of the two systems, preserving the lightweight distributed nature of block storage while providing the plethora of metadata and easy usage of the object storage.
    - Extensive research has been conducted in the field of storage solutions in edge computing infrastructures. A scientific journal entitled "*A Lightweight Storage Framework for Edge Computing Infrastructures/EdgePersist*" [49] has been published in Software Impacts (Elsevier) presenting the proposed edge storage solution (CHES).
- **KPI-2.3 Edge storage hit rate: higher than 70%**
  - The native "disk cache" feature of MinIO has been investigated. Disk caching feature refers to the use of caching disks to store content closer to the tenants allowing users to have the following: i) object to be delivered with the best possible performance

[15] https://docs.docker.com/registry/spec/api/

and ii) dramatic improvements for time to first byte for any object. Experimental results reveal a hit rate exceeding 84%.

- An online proactive caching scheme based on deep recurrent neural network models is investigated for research purposed, to predict time-series content requests and update edge caching accordingly.

- **KPI-2.4 Blockchain for edge storage transaction rate: more than 4 transactions per second**
  - A blockchain database, namely BigchainDB[16] is being explored. More specifically, BigchainDB supports both blockchain (decentralization, immutability, and owner-controlled assets) and database properties (high transaction rate, low latency, indexing, and structured data querying). One design goal of BigchainDB is the ability to process a large number of transactions each second. Each BigchainDB instance is a virtual concept consisting of three parts: i) a MongoDB database, ii) a BigchainDB server and iii) a Tendermint communication node which uses a Byzantine Fault Tolerant middleware for networking and consensus. Experimental results demonstrated that MinIO is able to achieve a higher transaction rate (*4.3*) compared to BigchainDB (*3.2*) for a specific class of experiments. The performance evaluation was executed through Locust[17], an open-source load testing framework that enables the definition of user behaviour and supports running load tests distributed over multiple machines and simulates millions of simultaneous user requests. Overall, the experimental results demonstrated that MinIO presents the best performance in both read and write operations. To further evaluate the storage systems, we also measured the RAM usage, the CPU usage, the disk latency and the disk IO time for a single user's request and for all users' request. Again, MinIO achieved the best performance. A scientific journal in the context of performance of storage systems in edge computing infrastructures entitled "*Performance Analysis of Storage Systems in Edge Computing Infrastructures*" has been published in Applied Sciences (MDPI) to the Special Issue Cloud, Fog and Edge Computing in the IoT and Industry Systems.
  - In addition, we conducted extensive experiments within a distributed computing environment, utilizing a configuration consisting of four nodes, and once again, we observed consistent outcomes. Specifically, MinIO demonstrated a superior transaction rate in comparison to BigchainDB and also achieved a better performance in both read and write operations. This reaffirms the robustness and efficiency of MinIO across varied deployment scenarios, further underscoring its potential as a high-performance data storage solution. A scientific conference paper entitled "*A Study on the Performance of Distributed Storage Systems in Edge Computing Environments*" has been submitted to the 9th ACM/IEEE Conference on Internet of Things Design and Implementation (IoTDI 2024), showcasing the aforementioned results.

3.6 Relation to research questions

There are a number of research questions regarding the edge storage, which are actively being researched at the moment. These questions include the intelligent data placement in computing networks, the pro-active and intelligent caching of data, the minimization of resource waste and the maximization of resource efficiency and the harmonization of IoT network diversity. The present research work and the designed component provides solutions to most of these open research questions by providing a complete edge storage solution that takes into account the present issues in IoT edge networks and the vast number of data transactions that continuously happen between them.

Pro-active and intelligent caching of data are two questions that also trouble the academic community

[16] https://www.bigchaindb.com/

[17] https://locust.io/

and the industry for a very long time. It concerns the replication or migration of data before they are needed to have them ready for usage when they are finally needed. This minimizes the wait time of operations since the I/O and network operations, which usually take much more time to be completed than processing does, are performed before they are needed. In order to achieve that, an edge storage system needs to be able to predict the need for a specific data packet early enough to be able to complete the data operations before the need arises. Modern approaches are using machine learning in order to profile the applications and the users of a system, extracting patterns of behaviour that hint at the future data operations. The presented solution is using Kubernetes as an orchestrator, which enables us to define certain node affinity and node selection rules that aid the selection of storage workers and the placement of the data inside an edge cluster. The affinity rules are relaxed rules that are instructing Kubernetes to prefer nodes that are meeting most of the affinity rules specified. On the other hand, selection rules are strict and instruct Kubernetes to deploy the storage workers on nodes that fulfil all of the selection rules. These rules can be dynamically set either by a network administrator or by an automated mechanism such as an intelligent agent or a machine learning model that can estimate the most efficient placement of storage workers.

Harmonization of IoT network diversity concerns the definition of a uniform way of handling the various IoT devices that can be a part of an edge cluster. An IoT edge network is like a living organism. The parts that comprise it can change at any given time either because they do not wish to be part of the network anymore, due to hardware or software malfunction, scaling out and in operations or for any other reason that removes or adds new devices over the device-edge-cloud continuum. The presented solution is using K3s as an orchestrator which is compatible with most devices that run windows or Unix based operating systems. This enables the administrators to create generalized deployment scripts that handle the deployment, configuration, un-deployment and re-deployment of the storage workers. These generalized scripts are highly configurable and can be edited in real time by higher level scripts and automated mechanisms adding more layers of intelligence and automation to these deployment and configuration processes. Additionally, DLF provides a uniform way of accessing the data, using the local file system of each device, eliminating the need of customized solutions for each new device that becomes a member of the device-edge-cloud continuum.

3.7 Evaluation of CHES

3.7.1 Evaluating CHES through Resource Utilization and Quality of Service Metric Analysis

The CHARITY Edge Storage Component aims at improving the Quality of Experience (QoE) of the end-users by migrating data “close” to them, thus reducing data transfers delays and network utilization. To evaluate the effectiveness of the storage component, a number of resource utilization and Quality of Service (QoS) metrics are collected using the Prometheus system. The data are collected on the edge, by Prometheus agents running on edge nodes that handle the data storage. These data are stored in the Prometheus database of each edge cluster. The data are collected at regular intervals of 5 minutes throughout the functional period of the component, i.e. for the whole duration that the edge storage component is active and waiting for serving data requests.

The evaluation metrics employed are divided into two categories:

- Resource consumption: CPU available (total, used), RAM available (total, used), HDD available (total, used), Network available (total, used)
- Performance: Throughput, Data request response time, and Network time

The resource consumption metrics of the first category are all being passively collected by the Prometheus agents placed on storage nodes. The performance metrics of the second category on the other hand, require a client-side approach so they are actively collected only during benchmarks and tests.

The evaluation is conducted using two CHES deployments, one in a local and one in a remote edge cluster. The behaviour of CHES is evaluated using a collection of small to medium binary files ranging from 15KB to 10MB. All these files are forming the evaluation dataset that is stored in various MinIO buckets, created and managed by CHES in the local and remote edge cluster. These buckets are then mounted onto new pods, using the DLF, and these new pods are taking the role of clients, sending data requests to the CHES and recording performance metrics for these requests.

Figure 19 illustrates the percentage change of various resource utilization metrics -CPU Usage, Memory Usage, Available Memory, Disk Write Latency, Disk IO time- during intense data transactions and during normal functionality of the node.

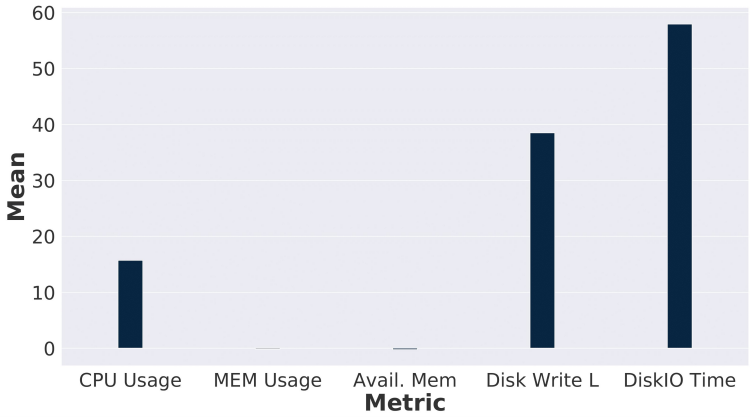


Figure 19: Percentage change of various resource utilization metric.

As the results suggest, CHES is not overusing the RAM of the node, although it is slightly increasing the usage of the CPU and the disk operations, as expected. This proves that CHES is lightweight enough to be deployed on most edge devices. More specifically, the RAM related metrics are near to zero, meaning almost no change, the CPU metric is slightly increased while the disk metrics are increased by a larger degree, proving intense I/O activity.

Client-side metrics collected to assess the impact of CHES on QoE, are presenting a clearer picture of how CHES improves the response times of various data requests. Figure 20 and Figure 21 show the comparison between read, write and delete operations for the local and the remote CHES respectively.

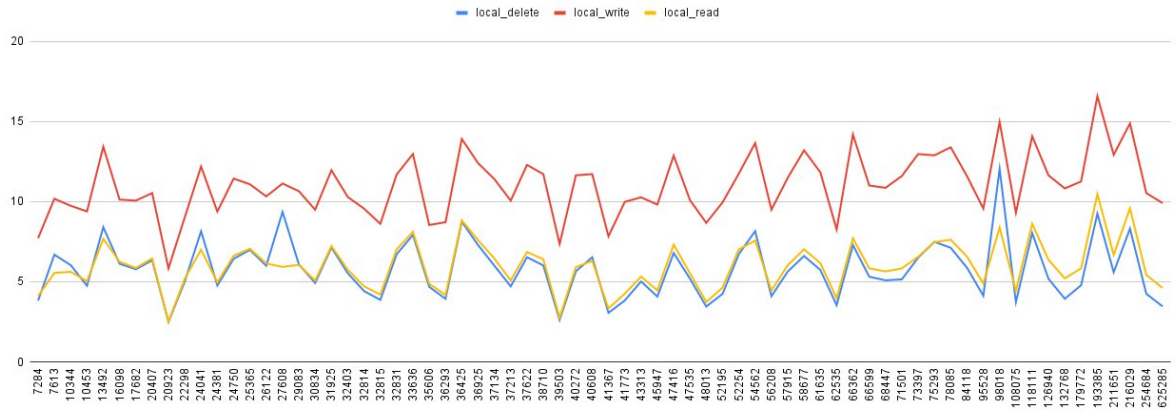


Figure 20: Read, Write and Delete operation response times in milliseconds for the local CHES deployment.

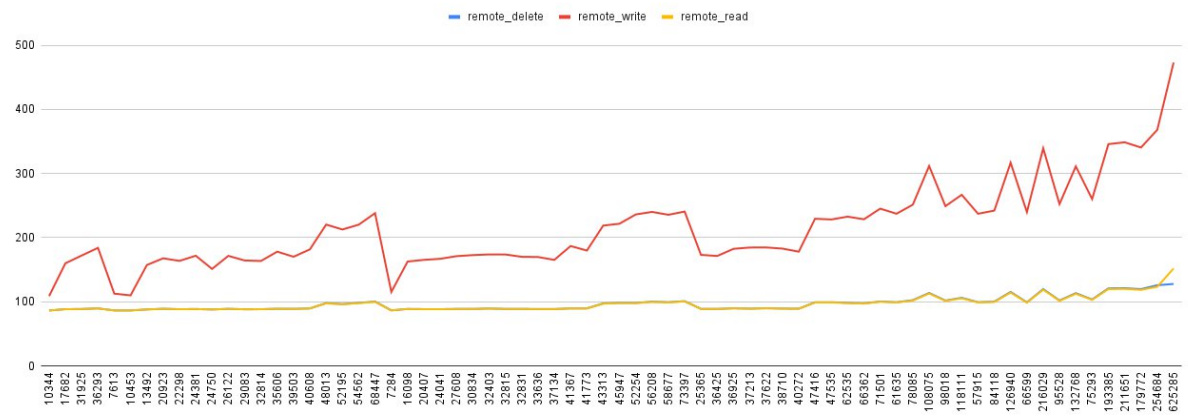


Figure 21: Read, Write and Delete operation response times in milliseconds for the remote CHES deployment.

Due to the object store nature of MinIO, it can be observed that write operations are more time consuming compared to read and delete operations. On the other hand, read and write operations do not differ much compared to each other, the only difference is the network delay for the final file transfer, which is pretty small taking into account that present evaluation tests were conducted using file transfers of multiple small to medium files.

The comparison between the different operations are similar but at a different scale; for the local CHES, response times vary between 3 to 17 ms while for the remote CHES, response times vary between 84 to 450 ms. This is becoming more obvious when putting the response times into direct comparison, as illustrated in Figure 22. The request response time for the local CHES is under 20 ms for all file operations which is significantly lower than the remote CHES. In summary, all data operations were significantly enhanced during runtime when the data storage was placed near the edge devices.

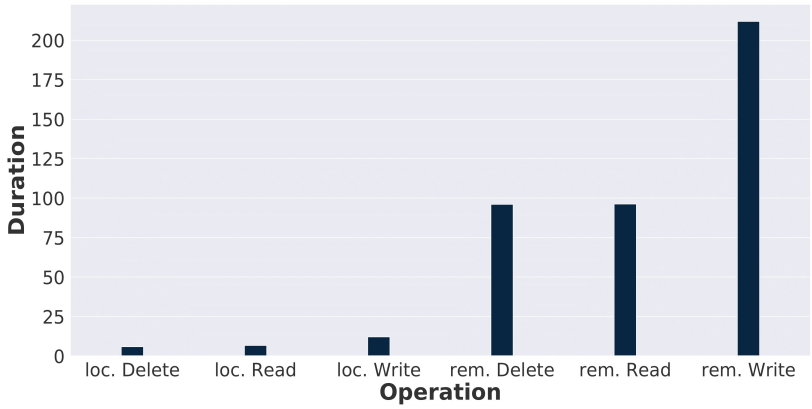


Figure 22: Comparison of response times for various operations for the remote and local CHES deployments.

In conclusion, the above experiments prove two things: a) the lightweight nature of the edge storage component, making it a perfect fit for edge device deployments and b) the great reduction in data request response times, which on some edge use cases is a necessity for their basic functionality.
Detailed results can be found at the scientific conference paper entitled "*Towards a Distributed Storage Framework for Edge Computing Infrastructures*" [29] which was presented at the 2nd Workshop on Flexible Resource and Application Management on the Edge (FRAME 2022).

3.7.2 Assessing CHES's Performance Perspectives

One of the main challenges in the development of applications at the edge is the efficient data sharing between the edge nodes, and it can be accomplished within individual application frameworks or through an external storage service. Despite significant improvements in offering an efficient edge storage solution, there are still some issues to be addressed related to the functional and non-functional requirements of cloud/edge-based applications, including low data retrieval latency, high availability and integrity, dealing with a potential shortage of storage resources at an edge node, supporting rapid application component deployment or automatic restart/replacement of unresponsive components, and dealing with the high heterogeneity presented in edge environments. These requirements can be achieved by optimizing resource usage, allocation, and data management plans on edge devices.

The plethora of available storage systems and underlying technologies have left researchers and practitioners alike puzzled as to what is the best option to employ in order to manage and process, in the most efficient way, the massive amount of data generated by IoT/edge devices. Therefore, we focused on highlighting the advantages and disadvantages of various edge-enabled storage systems. Thus, we present a performance analysis between CHES (MinIO storage), IPFS[18] and BigchainDB. The evaluation metrics employed are divided into two categories: resource consumption and performance. More specifically, three aspects were taken into consideration: i) transaction rate, ii) response time, and iii) resource utilization. To enhance the validity of our findings, each experiment was conducted over five iterations, thereby enhancing the reliability of our results and mitigating potential biases.

Figure 23 illustrates the transaction rate achieved by each storage solution. The results indicate that CHES achieves the highest transaction rate followed by BigchainDB, while the IPFS exhibits the worst results. For instance, the transaction rate obtained by CHES is 3.3 and 1.3 times larger compared to the IPFS and BigchainDB, respectively.

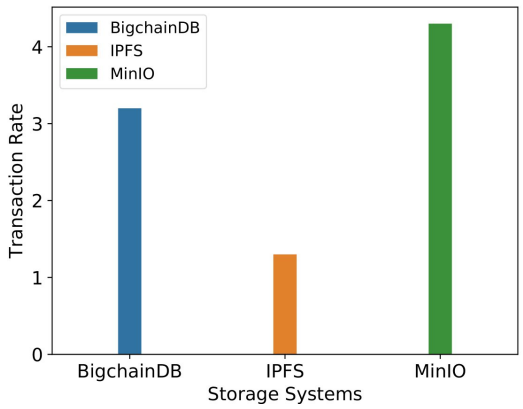


Figure 23: Transaction rate achieved by each storage solution.

Figure 24 demonstrates the average response time in milliseconds of each storage solution. Figure 24a and Figure 24b visualize the average response time of a single request of read and write operations, respectively. On the other hand, Figure 24c and Figure 24d illustrate the average response time for all users' requests. The standard deviation of the response time is also illustrated in each figure in a stacked bar plot manner on top of each average response time. Overall, as indicated in the above figures, CHES (MinIO) presents the best performance in both the read and write operations.

[18] https://ipfs.tech/

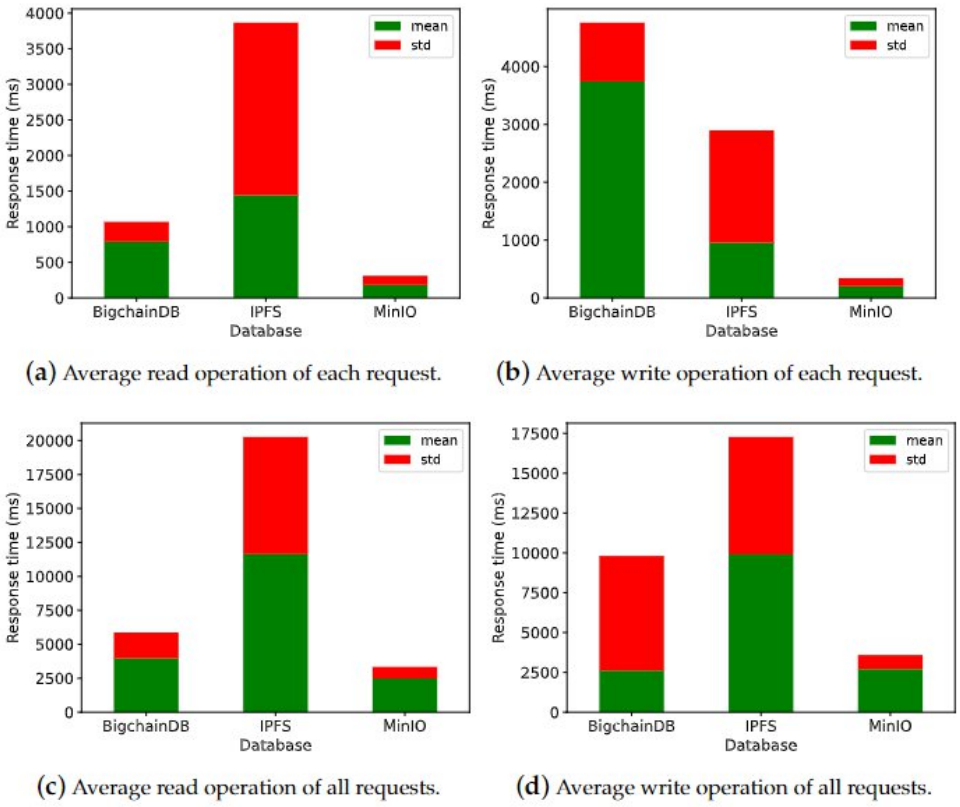
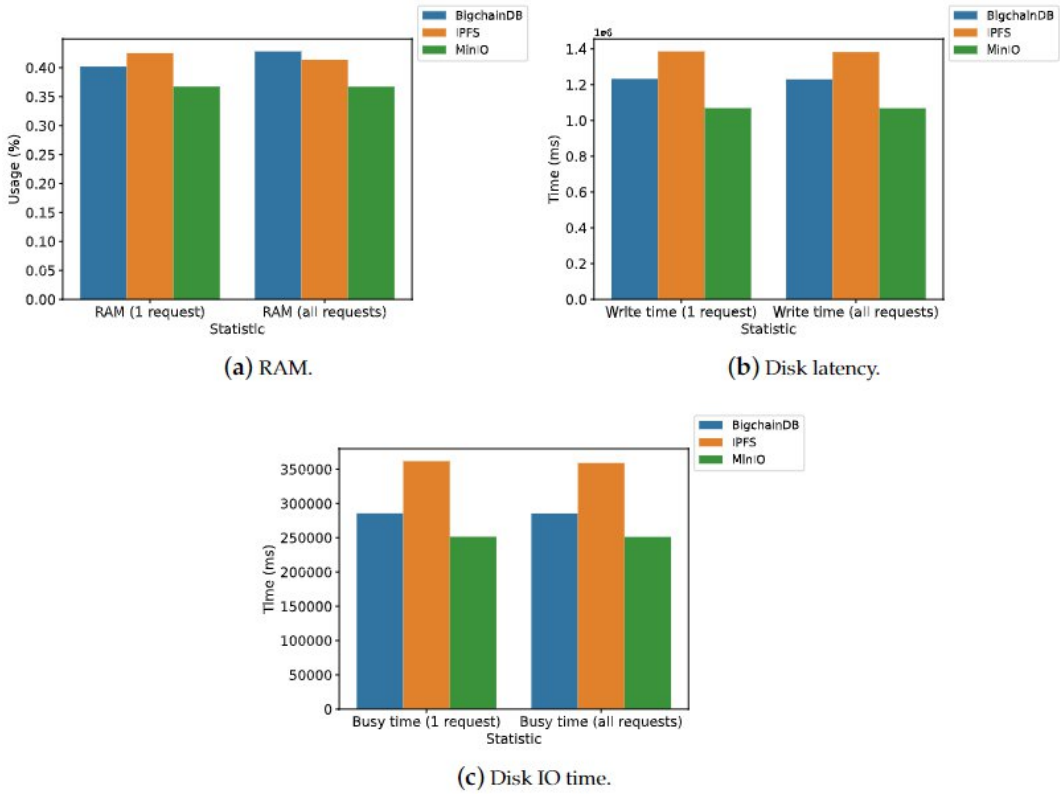


Figure 24: Performance of read/write operations of each storage solution.

To further evaluate the storage systems, we also measured the RAM usage, the disk latency, and the disk IO time for a single user's request and for all users' requests, similar to the previous figures. The CPU was also recorded but not plotted because its usage was negligible. This proves that the storage is lightweight enough to be deployed on most edge devices. Figure 25 and Figure 26 illustrate the statistics for the read and write operations, respectively. Figure 25a and Figure 26a indicate the percentage of the RAM usage where, as depicted, CHES consumes the least amount of RAM in each case. In addition, BigchainDB follows CHES, only in the case of a single request, with the IPFS is ahead of BigchainDB in all users' requests. In the rest of the figures where the disk latency and and the disk IO time are presented, CHES achieves the best performance followed by BigchainDB, while the IPFS yields the worst performance results. The disk metrics are increased by a larger degree, proving intense I/O activity.

Detailed results can be found at the scientific journal entitled "*Performance Analysis of Storage Systems in Edge Computing Infrastructures*" [28] which has been published in Applied Sciences (MDPI) to the Special Issue Cloud, Fog and Edge Computing in the IoT and Industry Systems.

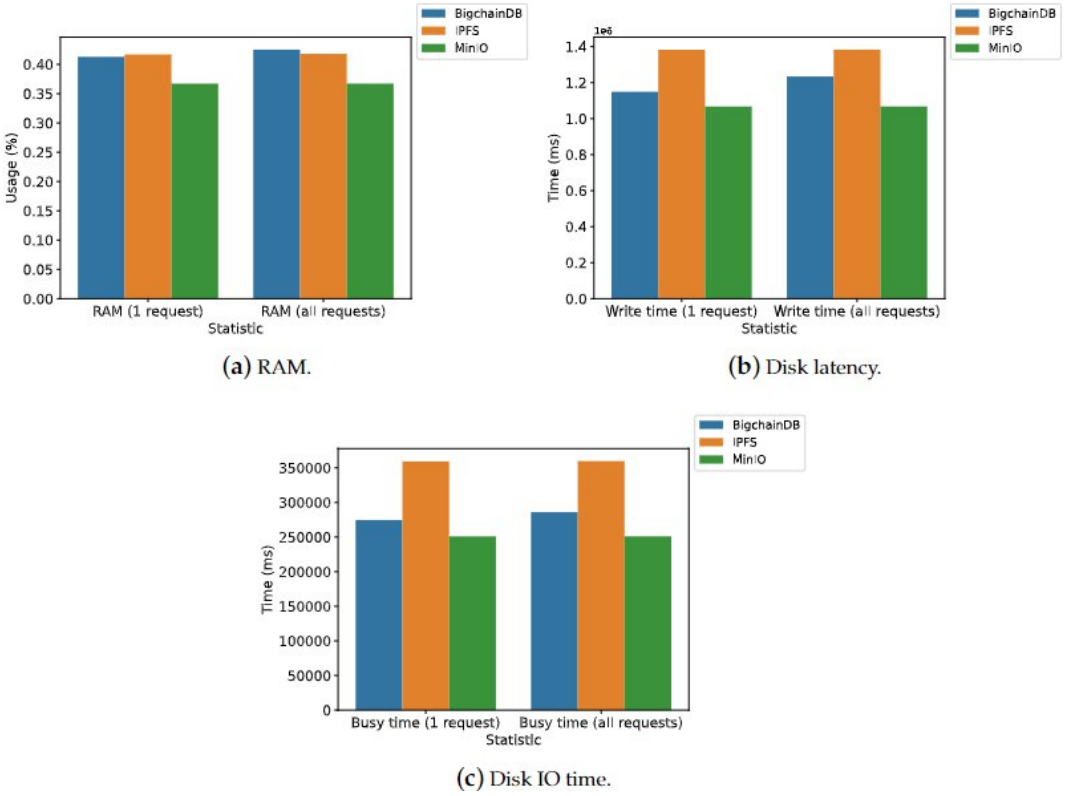


(a) RAM.

(b) Disk latency.

(c) Disk IO time.

Figure 25: Statistics for the read operation of each storage solution.



(a) RAM.

(b) Disk latency.

(c) Disk IO time.

Figure 26: Statistics for the write operation of each storage solution.

3.7.3 Evaluating CHES Registry sub-component

The seamless delivery of XR applications on resource-constrained edge devices, poses unique challenges due to limited network bandwidth, latency constraints, and intermittent connectivity. Additionally, the size of XR application images is often significant, and downloading these images from remote repositories can put a burden on the limited network bandwidth and introduce significant latency. CHES Registry sub-component serves as a crucial component, addressing the need to bring application images closer to the edge while minimizing network traffic and image download durations.

The feasibility and efficiency of the CHES Registry are evaluated through the examination of two specific use case scenarios: UC2-1 VR Medical Training and UC3-1 Collaborative Gaming. During the pilot evaluations of the VR medical training application, retrieving the 10GB-sized LSPart1 VM image from a remote repository led to considerable network congestion, causing delays in image download and concurrent network operations. This issue was addressed by pre-positioning the VM image within the CHES Registry on the same edge node before initiating a new VR session request. This change, which involved deploying the new VM from a local repository rather than a remote one, significantly reduced deployment times. In initial tests without CHES Registry pre-loading, the application took over 10 minutes to deploy, and in some cases, even up to 20 minutes. With CHES Registry pre-loading, deployment times dropped to 1-2 minutes. These results indicate that CHES Registry achieved deployment times up to 10 times faster than raw Kubernetes deployment. As the compressed LSPart1 VM image size is further reduced to 2.82GB, it will provide even more optimal deployment times. In the case of the Collaborative Gaming Use Case, the CHES Registry solution proved instrumental in minimizing network load and reducing deployment time for new game servers. This was achieved by strategically placing and hosting game server Docker images near the edge nodes in anticipation of their usage.

Overall, the evaluation reveals a significant reduction in application deployment time, indicating the positive impact of the proposed solution. Detailed results can be found at the scientific conference paper entitled *"Streamlining XR Application Deployment with a Localized Docker Registry at the Edge"* [48] which was presented in the European Conference on Service-Oriented and Cloud Computing (ESOCC 2023).

4 Resource-aware Adaptation Mechanisms

While the cloud offers extreme scaling opportunities through the dynamic allocation of physical resources to meet demand, it comes at a cost. Apart from the design challenges of engineering an elastically scalable architecture, the financial costs of cloud resources require careful monitoring. Although an application may be able to physically scale to meet demand, it may not be able to do so economically - unconstrained growth leads to unconstrained costs and if the returns do not exceed the investment, then cost can serve as a scalability brake. Edge computing resources such as those increasingly offered through metropolitan points of presence by hyperscalers and the forthcoming rollout of hyper-local edge infrastructure throughout 5G radio networks, offer new architectural options for domains such as real-time media streaming which require the flexibility of the cloud with the low latency typically associated with locally dedicated hardware. In comparison to traditional cloud deployment, edge resources are far scarcer requiring a more measured approach to scalability as there may simply be insufficient physical resources available in proximity to the user for optimal operation.

Across the cloud and edge, software engineers will increasingly find themselves challenged with designing software that needs to scale and dynamically adapt its tactics to suit the computational and network resources currently available within the environment in which it operates. With a Service Based Architecture approach increasingly favoured in modern architectures, there is a growing challenge with respect to how we equip services with sufficient adaptability to adjust their operation in line with the ebb and flow of physical resources available, and affordable, in their local environment.

4.1 Dynamic Software Adaptation

Software should be designed for change so that maintenance and reuse efforts can be minimised. Designing for variability has the significant advantage of enabling architects and engineers to delay key decisions until late in the development cycle or even until run time through site configuration. The longer we can accommodate a delayed decision, the more information we may have to hand when having to make the decision as requirements are adjusted in line with customer needs and environmental realities. These delayed design decisions are known as variability points [6] and the successful integration and curation of variability points has been the subject of intensive research for decades [7]. Variability points serve a key role in the design and construction of software product lines in which organizations seek to reassemble collections of reusable components into distinct members of a product family through leveraging a wide array of architectural, engineering and run-time variability point strategies ranging from abstract, interchangeable design stereotypes to run-time command line parameters [8].

While there is much active research into Software Product Line Engineering (SPLE) to attain development and deployment reuse efficiencies at industrial scale [9], the approach necessitates a highly planned, rigorous, and disciplined approach to variability management throughout the software design and implementation phases. It facilitates the reuse of software across multiple products in the same family by carefully designing variability points that can be leveraged during the software build and deployment process. An extension of this approach, known as the Dynamic Software Product Line (DSPL) paradigm, merges SPLE with techniques to adapt software at runtime to produce a collection of variability points that may be manipulated through configuration or runtime binding to alter the behaviour of deployed software [7].

Configurability lies at the heart of modern software development, and it is rare for software to be developed to such a narrow purpose and exact set of parameter values that no deployment configuration is required. Indeed, configurability is desirable as it can improve the versatility of software and often enable functional behaviour or adaptation to environmental setups that were not envisioned at the time of initial software deployment. While some software is equipped with runtime dynamic configurability and zero downtime, most of the software at least supports static

configurability. This could be facilitated through environment variables, command-line parameters or configuration held in a file or repository of some form. Such configurability essentially exposes a collection of variability points which can be manipulated to affect the behaviour of the software and the principle is the same irrespective of whether the software was developed in-house, open-source or closed-source acquired from a third party.

The number and nature of variability points exposed will vary from one application to the next and can range from debugging trace activation to port numbers and timeout values, from sampling rates to thread numbers. In fact, the very configurability of software often results in a software configuration space explosion [10] that causes challenges for the testability of software (Linux has well over 10,000 configurable features [11]). In the hands of a knowledgeable user however, configuration is a powerful tool to adapt and tune software to its environment and user needs.

4.2 A structure for adaptation

In [12], the authors put forth a vision of autonomic computing in which software systems could self-manage according to specific goals. Each component would be designed as an autonomic element which would manage its own internal behaviour and relationships with other autonomic elements through integration of an autonomic manager in each element. This manager would take responsibility for monitoring the operation of the element and its interactions and adjust the operation of the element as required (e.g., enable/disable features).

The autonomic manager comprises of what has come to be known as a MAPE-K loop – Monitor, Analyze, Plan and Execute according to available Knowledge. In DSPL, the autonomic manager becomes the adaptation manager as shown in Figure 27.

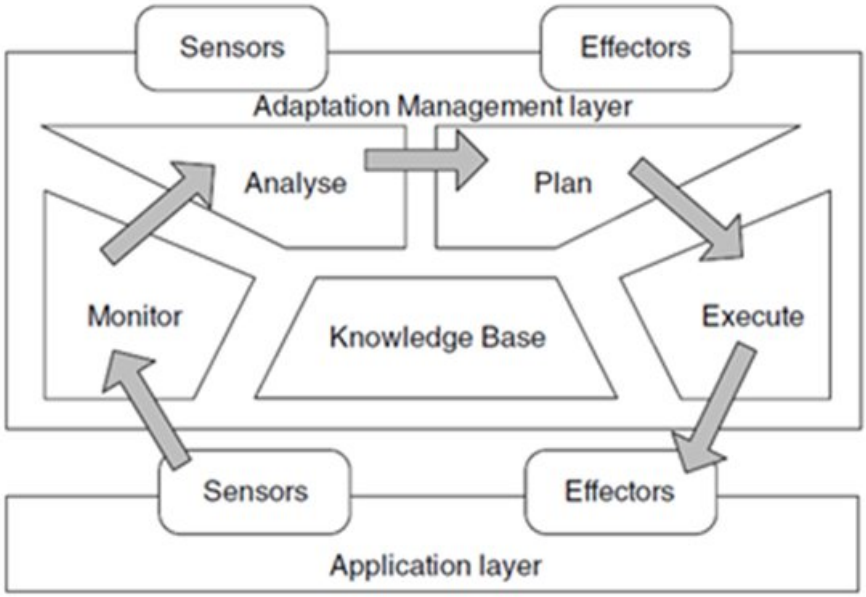


Figure 27: MAPE-K Loop [7].

The Monitoring step is concerned with capturing data regarding the properties which will drive the adaptation choices. The Analysis step examines the monitored data and performs any necessary pre-processing before making it available to the Planning step which decides, if adaptation is required, which variant of the system is more suited to the current conditions. Once the variant has been identified then the Execution step performs the adaptation.

4.2.1 Context Monitoring & Analysis

Applications and their environment need to be monitored to observe when the operation of software needs to be adapted. To record the properties being monitored, the adaptation manager can maintain flat context variables [13] or a more sophisticated hierarchical ontology [14] maintained as a dynamically updated property set that can be undergo analysis using pre-defined rules or queries to check for conditions that would warrant an adaptation.

4.2.2 Planning

"Our claim is that a major reason for the lack of context-aware, adaptive mobile applications is the inherent complexity of building them. Not only need the developers understand the main functionality of the application and how this can be provided on a mobile device, but they also have to conceive different application variants, specify how applications are linked to the execution context variables, and consider which variant should be activated under which context conditions. This complexity may easily appear like an insurmountable barrier to the developer" [13].

As mentioned previously, the potential system variant explosion arising from variability points in a software application can overwhelm the testing efforts. If left to an adaptation manager to explore unbridled, automated manipulation of variability points at runtime can lead to operational profiles that were not tested or foreseen by the developers. In the field of DSPL, the approach of static goal evolution involves an approach in which a software system has a fixed adaptation policy and system variants [7]. In the event the system needs to adapt to a new goal (operate at a reduced media streaming resolution for example), then the system is stopped, modified and restarted. Verification of such systems is greatly simplified as the state space is highly constrained. This suggests a model in which the Planning step of the MAPE-K loop can collapse to the selection of a particular variant in response to a given goal.

4.2.3 Execution

To initiate adaptation, it is required to reconfigure the software using some form of runtime reconfiguration mechanism. How this may be accomplished naturally depends on the design and capabilities of the software. Approaches based on capabilities of the software architecture range from dynamic aspect weaving essentially rewiring the software assembly on the fly [15] to service re-routing in a service-oriented architecture. In [11], the authors examined self-adaptation within a micro-service architecture for a media streaming platform in which they proposed leveraging the rollout functionality available in the Kubernetes platform which can perform rolling upgrades of a given micro-service without service interruption.

4.3 Challenges

In CHARITY we seek to enable the self-adaption of software systems to significant fluctuations in the resource availability within the execution environment. Based on an analysis of the state of the art and considering the needs of CHARITY, we identify several challenges.

- **Avoid design time intrusions**.
  We seek to avoid prescriptive, opinionated approaches which step into the architecture and design of such systems requiring scaffolding and algorithms to be integrated. We adopt this position for several reasons. Firstly, most software is legacy software and seeking developers to modify this software retrospectively creates a significant barrier to adoption. Secondly, updates to the adaptation design and capabilities places an onus on developers to integrate these changes into their software resulting, over time, in version mismatches and requiring constant vigilance to maintain backwards compatibility. Thirdly, not all the components and services employed in a given software system are modifiable. They may be commercial or otherwise unavailable for modification and, even when the source is available, it may have

been written by a third party (e.g., open source) and difficult to modify without subsequent upgrade and maintenance concerns.

- **Prevent platform instability**.
  CHARITY seeks to support a micro-service architecture which can involve chains of services working together. When performing an adaptation, we need to be careful that the integrity of the chain is maintained.
- **Accommodate user-level adaptation**.
  CHARITY also aims to support media streaming software services that operate at scale. At any given point in time, there will be a mix of users using a particular service that may necessitate different priorities. For example, in a flight training simulation system, users co-operating in a key team operational training exercise may take priority over individual users experimenting with the controls of the flight simulator. Alternatively, we may want to maintain a high Quality of Experience (QoE) for existing users but lower the QoE for new users entering a resource-stressed environment. Supporting this model of operation will require that CHARITY supports a multi-tenant architecture where applications can simultaneously operate in different modes and priorities.
- **Transparency & Tractability**
  It is imperative that system adaptations are predictable and visible to avoid instability or loss of confidence.
- **Cattle Not Pets**
  The cloud-native metaphor of pets versus cattle [47] promotes that a running software service should not be treated as a pet – unique, carefully managed and whose loss causes upheaval. Instead, it should be treated as cattle – easily and seamlessly replaceable with another fulfilling the same role and managed in bulk.
  An infrastructure that supports the dynamic adaptation of software applications could request that candidate applications be themselves adaptable at runtime and expose APIs that can be used to have the application manage its runtime resource usage itself. This would be a more straightforward proposition than seeking to adapt applications that have no inherent capability to do so themselves dynamically. However, changing the state of services within an application in this fashion breaks one of the bedrock tenets of cloud native software design – services should be transparently disposable and replaceable. If we have altered the inner workings of one or more services within an application through adaptation APIs that it exposes, then what happens if that service terminates unexpectedly (and needs to be restarted) or needs to be migrated to another server? Cloud native demands this versatility yet how does the replacement get to the state that the service it is replacing was in? This would require state tracking and synchronisation which incurs significant overhead and almost certainly would require intrusive customization of the application to support this – not to mention customization of the adaptation infrastructure itself that would be required to invoke application-specific APIs using whatever communication protocol the application supports.
  This led us to shy away from using application-specific APIs for dynamic application adaptivity and instead strive for a solution that would be reusable across third party applications and adhere to the tenets of cloud native computing. To support cattle not pets.

4.4 Adaptation Infrastructure

As discussed previously, variability points are used in Software Product Line (SPL) engineering to delay decisions until such point as we are better informed as to how software needs to adapt to its use and environment. Run-time adaptation through manipulation of variability points at run-time is used in Dynamic SPL (DSPL). In CHARITY we propose to implement a DSPL model which utilizes existing variability points in a software application to facilitate provisioning of different service editions where a service edition is an application instance with a distinct runtime configuration. This runtime configuration would be selected in accordance with the observed environmental conditions.

In this model, the function of any given service edition does not change during its lifetime (i.e., the software itself is not expected to self-adapt) but rather different configurations of it are selected according to the environmental circumstances. This model is depicted below in Figure 28 in which we show three services – each with multiple editions – that exchange information to operate an overall software application.

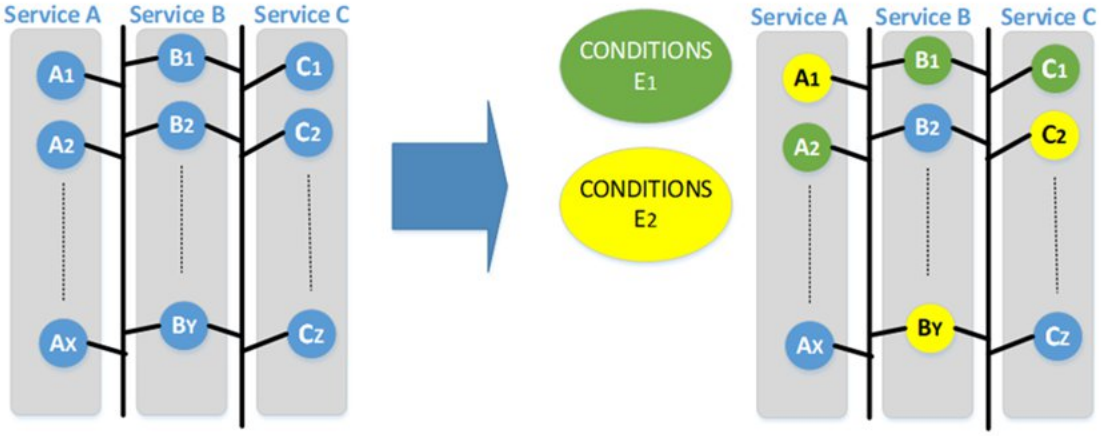


Figure 28: Service Editions used to satisfy different environment conditions.

In effect, we propose to use static goal evolution [7] in which we constrain the variability state space to explicitly configured variants and thus prevent the application from entering into unforeseen (and untested) states. There are a range of challenges involved here:

1. How do we enable multiple editions of a single service to operate alongside each other.
2. How do we decide which editions to use under given circumstances and wire these together into a safe and coherent service chain.
3. How do we route traffic between services without them needing to be made aware of multiple editions.
4. How do we monitor the environment.

As we will discuss in the following sections, we propose an evolution of the MAPE-K loop introduced previously for runtime adaptation in DSPL. In CHARITY we propose to position Monitoring and Container Management platforms between the Adaptation Management and Application Layers as shown below in Figure 29.

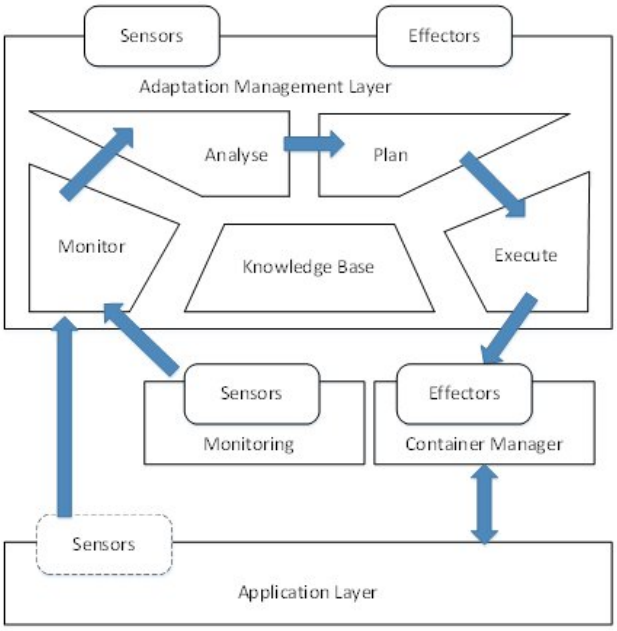


Figure 29: MAPE-K look modified to enable extraction of sensors and executors from the application layer.

Note that in Figure 29, the sensors shown in the Application Layer are facilitated but not obligatory.

The following sections outline how we plan to meet these in CHARITY according to the previously identified research and technical challenges.

4.4.1 Configuration Containment

One of the fundamentally transformative benefits of Docker containers for software development has been the ability to create separate self-contained environments for experimentation and production. On a single host, we can deploy multiple containers hosting applications that, if run collectively outside the container confines on a single node, would come into conflict with each other – for example, conflicting version requirements of common software packages; conflicting requests to use the same ports, environment variables or journal files. Containers allow us to run multiple copies of the same application side by side without coming into conflict. This ability to contain the application's environment to just that application allow us to painlessly run multiple copies of the same application side-by-side with different configurations. Configurability through feature flags and configuration options at application launch is a widely used technique in software development to offer a variety of deployment variations to suit the needs of the given environment (whether business or operational) [4]. Docker containers enable us to leverage the power of this configurability in a production environment.

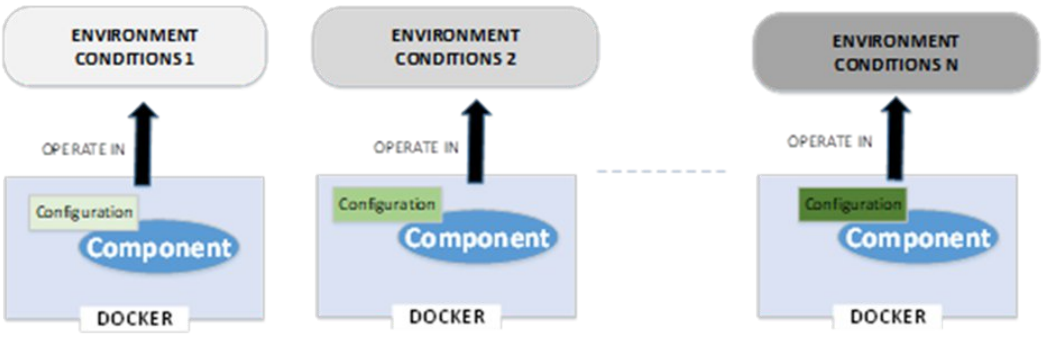


Figure 30: Run differently configured copies of a single application simultaneously.

Given a particular set of environmental conditions (e.g., GPU availability, network latency, user request profile) then we may find that a change to the configuration of a given component to alter its mode of

operation (e.g., disable feature, reduce sampling rate) may produce a more stable application that operates more in tune with the environment in which it finds itself.

While some software is equipped with runtime dynamic configurability and zero downtime, most will support static configurability through environment variables, command-line parameters or configuration held in a file or repository of some form. Such configurability essentially exposes a collection of variability points which can be manipulated to affect the behaviour of the software - irrespective of whether the software was developed in-house, open-source or closed-source acquired from a third party. By leveraging the environment isolation properties of containers, we can launch multiple instances of a service in different configurations. As we shall see, coupled with the ability of Kubernetes to orchestrate the launch of groups of services, containers bestow a powerful ability to seamlessly replace whole subsets of a service-based application to deliver a coherent application variant – involving multiple individual service variants working in concert - in a safe and predictable manner.

4.4.2 Service Dependencies

In a distributed service-based architecture, some service relationships will have stricter constraints than others. Some relationships will be predicated on extremely low latency communications, use of shared host resources like network space (localhost), storage volumes and shared memory. Some services need to start together, scale together and stop together. We require service groupings to accommodate these circumstances and such a concept can be manifested with *Kubernetes Pods*.

Pods enable us to group containers together into a single meta-container that is deployed on a single host. The Pod lifecycle controls all the containers within. Below in Figure 31 we depict the high-level Pod concept.

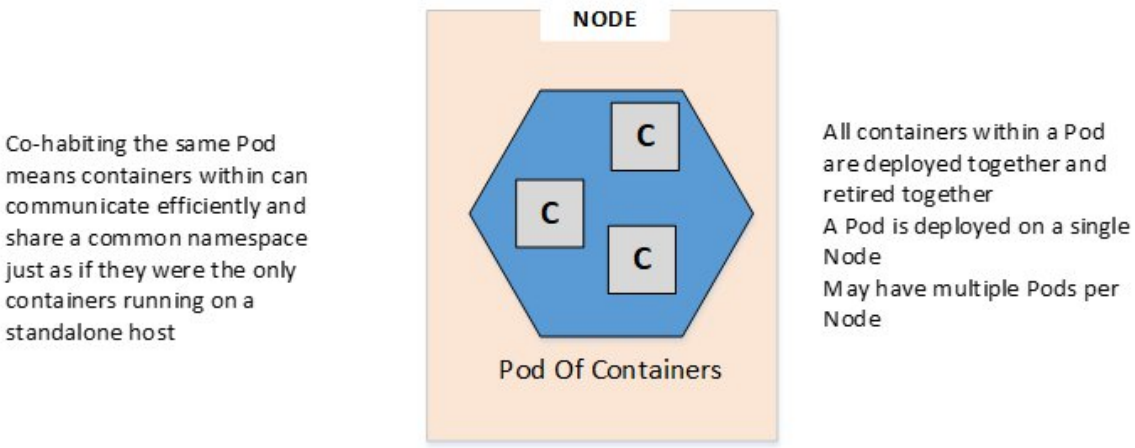


Figure 31: Co-dependent Containers are deployed as a unit in a single pod.

With Pods, we have a means of collecting tightly related services into containers within a single deployment unit. This gives us a powerful and elegant mechanism to deploy, redeploy, reconfigure, and retire such service groupings as a unit.

4.4.3 Service Routing

With more focused and cohesive segmentation of responsibilities into separate services, service-based architectures rely extensively on inter-service communication to collectively perform their work. In Microservice-based architectures, the mechanics of enabling services to communicate with each other robustly requires careful and detailed design and planning. Apart from peer discovery, there are significant challenges involved in establishing and monitoring communication links. Transferring

control from one process to another – irrespective of the distance between them – requires coordination in the event of link failure. We must facilitate failover between multiple copies of services and indeed decide on the efficient distribution of traffic when multiple peers are available to accept it. When deploying a new service edition (same service software with different configuration) we need a means to allow both editions to co-exist for some short duration of time and to seamlessly handover traffic from the old service edition to the new one.

Our initial investigations and research focused on the use of a Service Mesh [5] to offer an overlay that could be used to manage routing, and route changes, of communication between services so that we could switch between service editions. While this initially looked promising it did become clear that it was not without its shortcomings. It would increase the complexity of the design through the introduction of an additional infrastructure layer that would require deployment, configuration, and maintenance. Additionally, it would require a custom synchronisation layer to co-ordinate proxies in the service mesh such that switches between service editions occur together when multiple containers/pods are involved.

As we progressed our thinking and our familiarity with Kubernetes, an alternative approach suggested itself in the form of Kubernetes Services. A Service can be used to expose a single IP address and seamlessly route traffic to multiple Pods – ideal for load balancing traffic between identical pods or transparently handling a restarted pod that has been assigned a new IP address. Below in Figure 32 we see a high-level depiction of a Kubernetes Service.

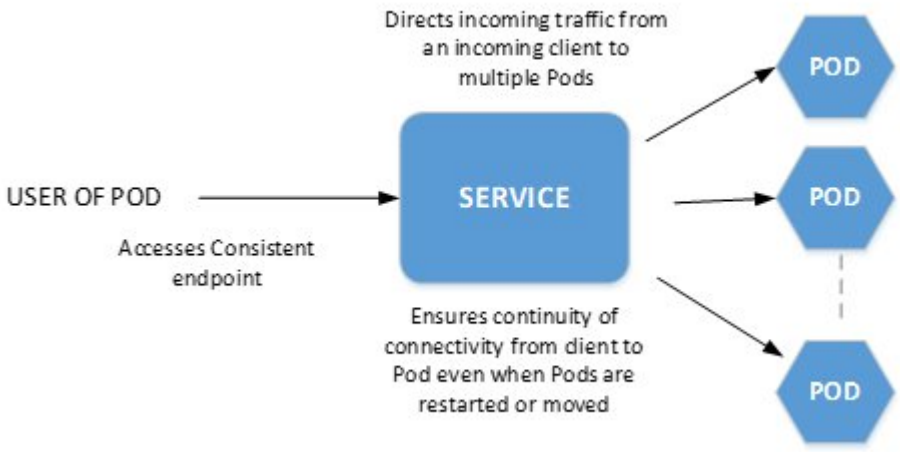


Figure 32: A Kubernbetes Service conceals pod churn from the clients.

4.4.4 Application Quality Modes

Consider an application comprised of three microservices as shown below. The services deliver a response or perform a particular action in accordance with a request. We refer to the sequence of services involved in delivering on this response as a Service Chain.

Figure 33: Simplified Application with Microservice Architecture.

For an interactive XR streaming application, the QoE is typically measured according to the response or action performed in response to the triggering request on several dimensions.

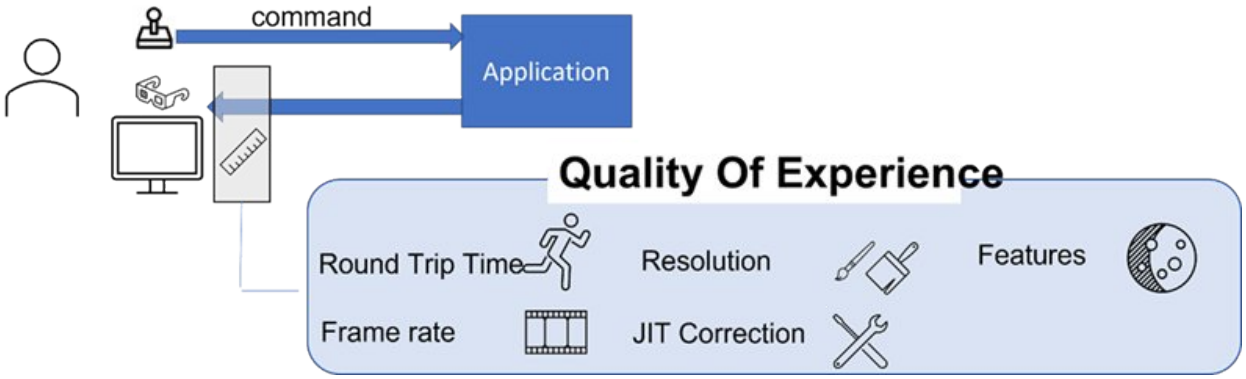


Figure 34: XR Application Quality of Experience is often multi-faceted.

The Round-Trip Time (motion-to-photon, glass-to-glass) captures how long it takes for the application to deliver updated imagery in response to the triggering user interaction. The Frame Rate captures how many frames per second the application is delivering to the user device. Resolution captures how many pixels per frame are being rendered. Just in Time (JIT) Correction is the term we will assign to processing carried out on the generated media stream to try and compensate for insufficient frame rates, delays, or insufficient resolution. Such processing generally involves algorithmic guess work to repair incomplete media streams on the fly through pixel or frame rate upscaling. Features typically involve visual flourishes such as sophisticated weather effects, reflections and shadows but could also include some AI-driven augmentation such as object recognition and framing to assist the end user.

In an ideal world, we may want a sub-20ms RTT, 90 FPS, 4K resolution, no need for JIT correction and full feature set enabled. In an ideal world we have unlimited resources. The application provider knows that resources are not unlimited and that networks get congested. We propose to offer the application provider the facility to specify configurations of their application that would offer acceptable, but less than ideal, Quality of Experience specifications. The objective is to allow the application to remain operational in resource contested environments. To explore this concept, we present the application provider with the facility to specify three modes of target QoE – High, Medium, Low – representing the different levels of QoE we want to be able to deliver. We will term these QModes. While the objective of QModes is to capture different levels of physical resource consumption by the application running in a virtualized environment, what constitutes a given QMode only makes sense within the context of a particular application. QModes may be differentiated for example, by the set of rendered features (e.g., accurate weather effects, reflections, shadows), by the number of simultaneously active users, the resolution and/or frame rate delivered to the HMD, or even the placement and operation of service components across the device-edge-cloud. For a given application deployed on our platform, it's QMode values map to distinct deployment configurations of the application.

In the figure below we see three different configurations of an application and the introduction of a logical switch that can choose which deployment configuration to route traffic to. In reality, not all services are affected by a given configuration change (changing the resolution of a user interface may have no effect on the operation of a backend database for example). Just because we change the application configuration, then it does not imply that all the constituent service operational profiles change.

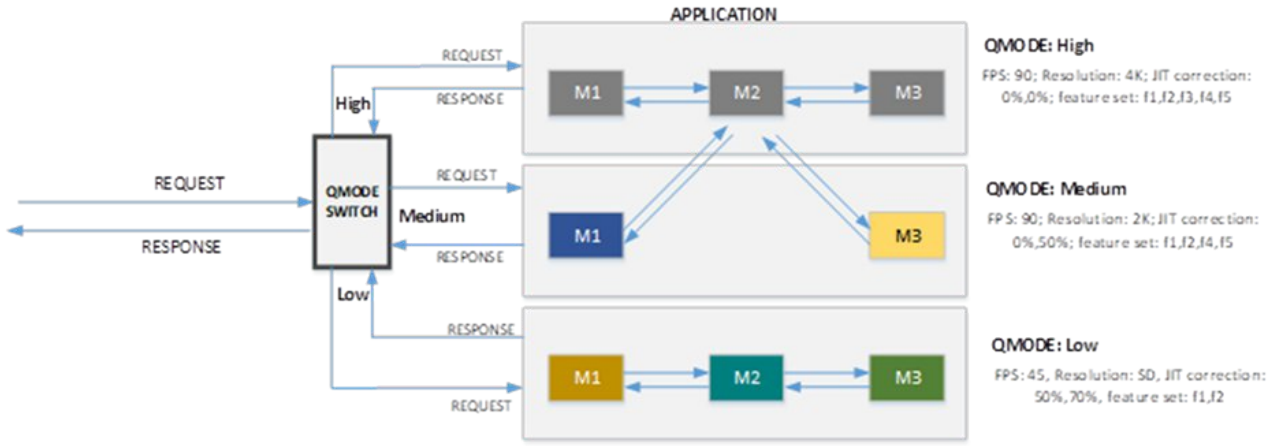


Figure 35: Logical QMode Switch and how it could be employed to divert traffic between different service configurations.

In the above model, we can see that a single service may behave identically in multiple chains.

A multi-user application will likely be operating in multiple QModes simultaneously. We view QMode as being tied to a particular traffic characteristic. Different users may be assigned different QModes according to their circumstances (e.g., SLA, device capabilities, local network congestion levels, etc.).

Conceptually, a QMode enables network routing in a similar fashion to a VLAN in that it allows us to segment and route traffic according to a tag. The choice of QMode to perform at can depend on a variety of factors. Application providers may elect to differentiate based on class of device (is it capable of high resolution, does it support frame interpolation[19], etc.), speed of network, availability of edge resources, user contract, number of local active users, etc.[20]. To be able to make this choice, however, requires that we gather and monitor this information in a centralized monitoring framework.

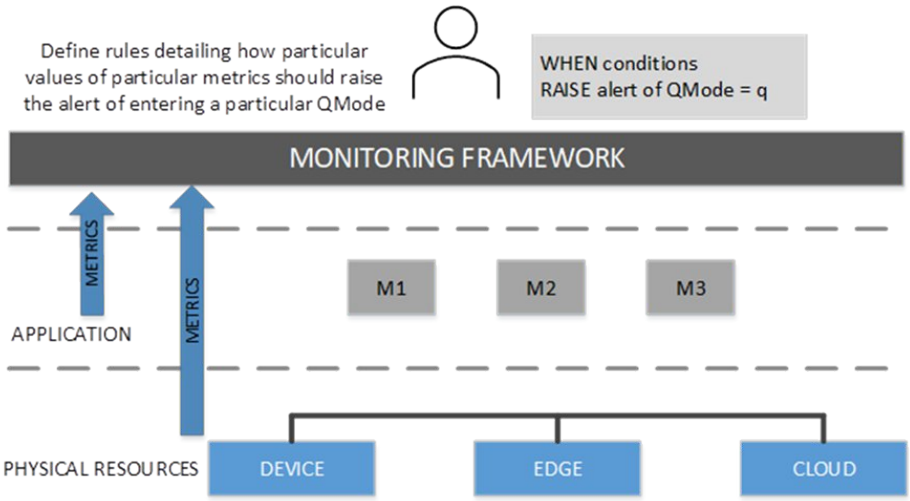


Figure 36: Monitor for conditions that warrant changes to QMode.

[19] For example, SteamVR Motion Smoothing or Oculus Asynchronous Space and Time warping.

[20] It is quite possible that three QModes would not be sufficient to capture the complexity of conditions and granularity of configuration options available to a given application provider. We have restricted ourselves to three modes to simplify concept evaluation and development.

4.4.5 Monitoring & Analysis

Adaptation requires context. The drivers for adaptation can vary according to the business and resource environment but in general, applications must adapt to resource availability. An XR application distributed across device, edge and cloud resources can depend on a delicate, geographically dispersed, web of resources. Monitoring every leveraged resource individually, seeking to detect bottlenecks and deficiencies, can overwhelm our decision making. Without intimate knowledge of an application's resourcing windows and inbuilt compensation mechanisms[21], we may elevate disparate resource stresses (such as link delays, GPU overload, database response times) to high priority problems that require countermeasures while, in fact, the application is still able to operate as a whole and deliver an acceptable quality of experience to the end user. A more sensible approach would appear to be initiating action in response to a small number of high-level red flags that holistically capture underlying problems rather than monitoring a multitude of low-level warning indicators.

The ultimate purpose of any application is to perform its work and deliver acceptable performance and experience to the end user. If the application is delivering an acceptable Quality of Experience (QoE), then we could deem the application to be performing adequately and not in need of adaptation.

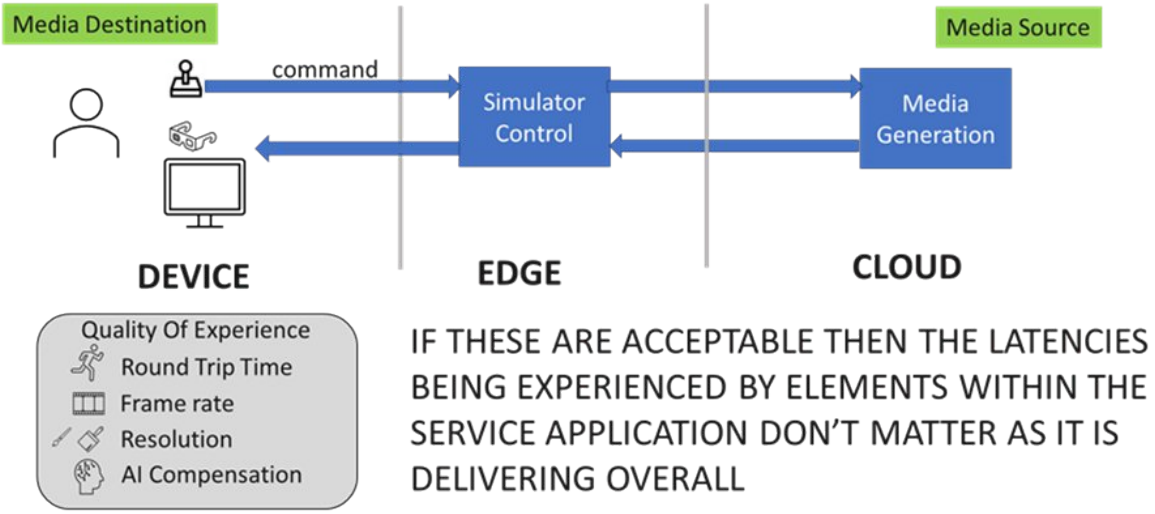


Figure 37: Monitoring High level indicators reduces decision complexity.

In Figure 37, we see representative XR metrics we can monitor for conditions that capture the overall fitness for purpose of the application:

1. Round Trip Time: as stated before, the length of time between a user action and its reflection on the visual experience
2. Frame Rate: How many frames per second we are delivering to the user device
3. Resolution: the pixel depth of the frames we are delivering
4. AI Compensation: Rate of interpolation/extrapolation we need to do locally to 'fix' sub-standard resolution or frame rate being delivered from the visual renderer. This may arise if a remote visual renderer generates lower quality media streams to reduce bandwidth needs from the cloud while it is upscaled at the edge or on the device.

[21] It may transpire, for example, that a well-resourced database equipped with advanced SSD disks can compensate for an underperforming cache relying on overly stressed RAM. Such trade-offs and compensations are generally particular to each distinct application. In addition, application providers generally dimension some latitude into their resource requirement specifications to accommodate leg room and usage peaks that may not always be used. An over-eager adaptation mechanism may seek to fix a problem that does not need fixing.

By monitoring these metrics, we can assess the application's fitness. Requesting the application provider to specify meaningful thresholds and operating windows for these metrics is reasonable – unlike requesting them to specify a combination of hardware resource availability deviations that could expose a problem.

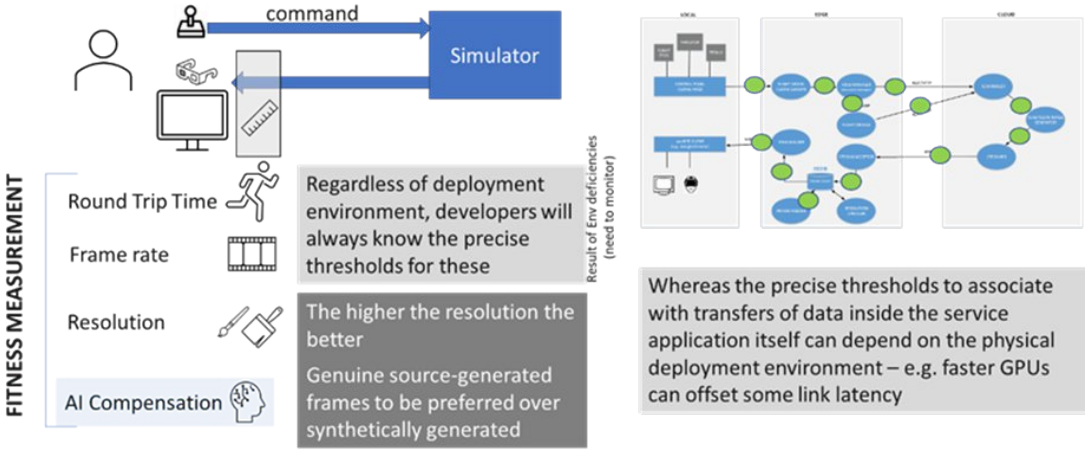


Figure 38: Monitoring the manifested user experience is more tractable and efficient.

While unacceptable levels of application fitness may highlight a problem, high-level indicators cannot inform us what the cause of it is. They inform us *when* to investigate a manifest problem rather than necessitating constant low-level monitoring and analysis to ascertain if we can deduce a problem. Root-cause investigation requires examination of far more detailed and lower-level metrics (such as individual service performance, particular link latencies or bandwidth shortcomings, and queueing backlogs) as gathered by the CHARITY monitoring platform. The driver for this level of analysis is that applications may be adapted differently depending on the root cause of the problem. For example, a deficiency in the response time from a cloud-based service to an edge node may require different adaptation than experiencing resource stresses on the edge node itself. We seek to enable application providers to fully leverage the adaptation avenues they have available to them within their application design.

This requires us to be able to retrieve metrics relevant to the application under investigation – an application that may be operating across multiple nodes over the device-edge-cloud continuum.

In Figure 39, we see the role of monitoring in application adaptation.

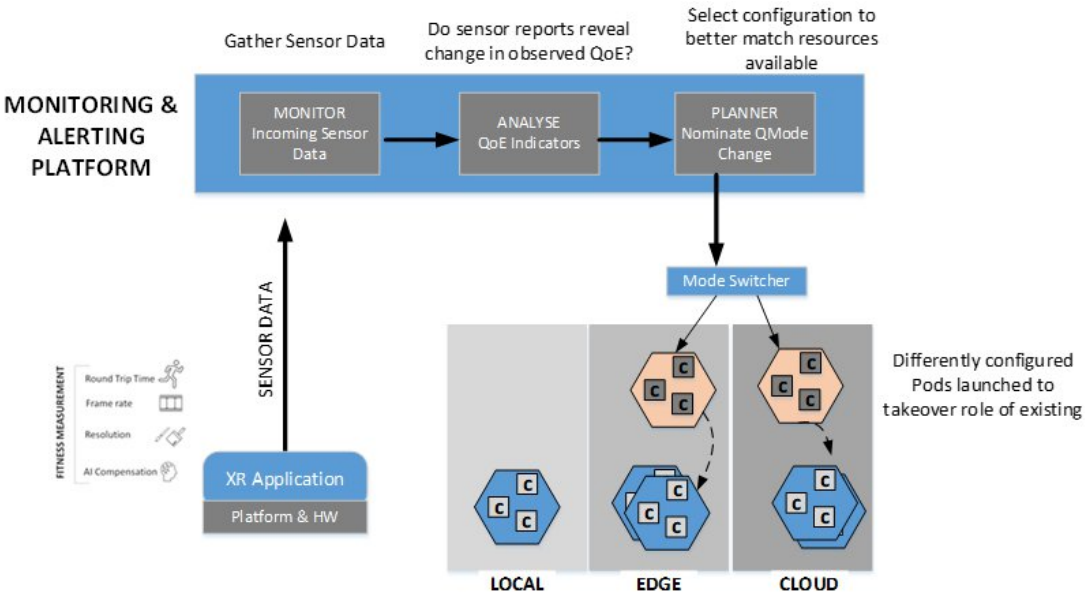


Figure 39: Monitoring, Analysing & Planning based on observed sensor data.

We employ Prometheus and its Alert Manager to trigger examinations of lower-level metrics when SLA-breaking conditions are observed with higher-level *Fitness Measurements*. The actual mechanics of how QMode updates are relayed to the Application is a topic we will return to when discussing Investigative Work later in this section.

4.4.6 Planning & Execution

When analysis of ongoing monitoring reveals the occurrence of conditions warranting a QMode change then an alert is raised and relayed to the Prometheus Alert Manager. The Alert Manager in turn publishes the alert.

The logical QMode Switch we referred to earlier routes to a particular application configuration based on the current value for the QMode associated with the application. Applied at the global system level, this would have a sledgehammer effect. We need to be lighter handed and enable QMode changes to apply to a subset of user sessions. How this is accomplished depends on the architecture of the application. In the case of the Collins Use Case (UC3-2 Manned-unmanned Operation Trainer), the application operates dedicated services for each user and there is very little shared state. In this scenario, enabling QMode changes for a single user entails a replacement of the Pods serving that user. We could envisage other applications with different architectures in which multiple users are served by a single group of services. In this scenario, enabling QMode changes for a single user cannot be easily accomplished with replacing the existing services with alternatively configured instances as this would affect all users sharing those services. Instead, we must operate a group of services per QMode and have users who are currently assigned a common QMode to share a common group of services[22].

4.4.6.1 User-Level routing

To support user/session level granularity then the switch needs awareness about the user associated with a given request.

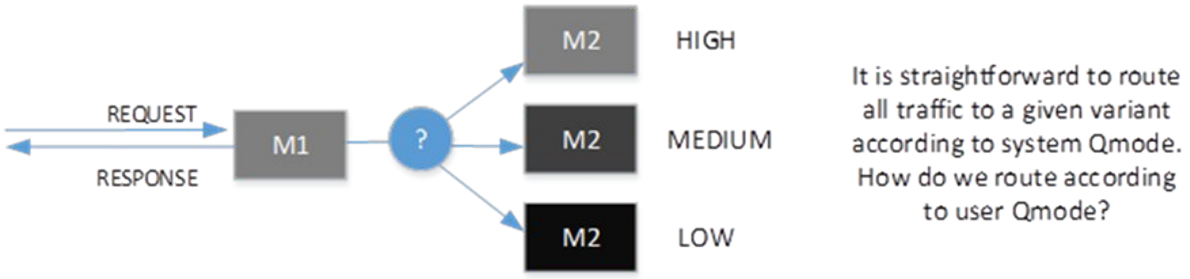


Figure 40: QMode Routing.

The proposed solution lies in associating a QMode tag with each user session and having a particular application configuration to be employed for a given QMode tag. Adapting an application fundamentally entails instantiating a variant of the application, having it run side by side with the original while it prepares itself to accept traffic, and then switching live traffic to the variant so that we can retire the original. Below we depict a snapshot in time when it has been decided to swap the user to a lower-resource-consuming variation of the application and we are ready to switch the traffic over.

[22] We have focused on integration with the Collins Flight Simulator Use Case and as such look to adapt applications that follow the model of dedicated resources per user. In many respects this is the more challenging scenario as change necessitates increased upheaval within the platform. We cannot simply re-route to already deployed Pods – instead we must always initiate a rolling update.

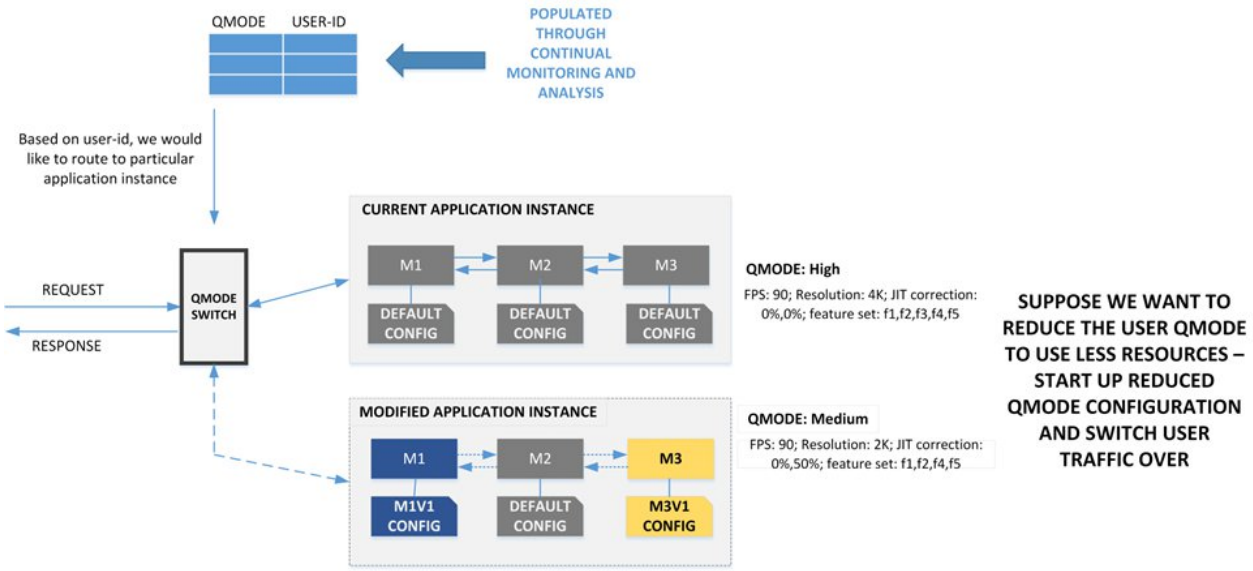


Figure 41: Application about to switch over to application variation that consumes less resources.

Note in the scenario depicted above that not all services in the application are reconfigured. We can see that M2 is unchanged.

4.4.6.2 QMode Switching

Switching essentially requires reconfiguring the services dealing with a user and re-routing the user's traffic through these newly configured services. In Figure 42 below we see the high-level sequence of actions required.

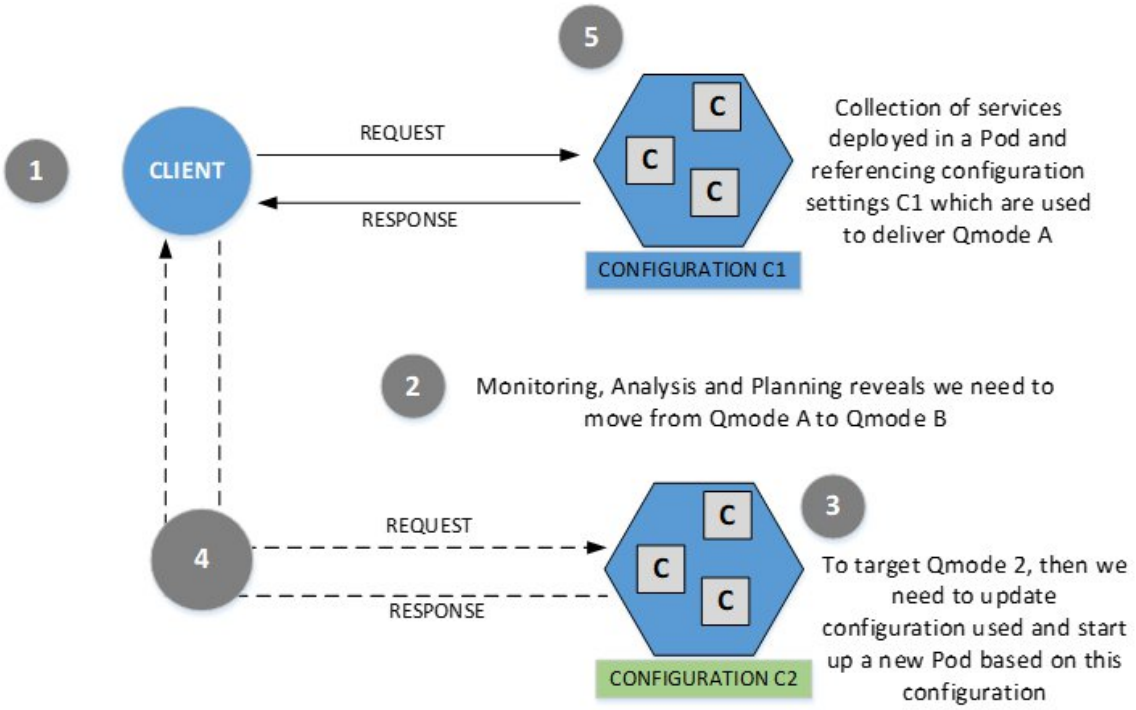


Figure 42 - QMode Transitioning.

1. We begin with client traffic being directed to existing services which reference configuration C1.

2. We observe that resource deficiencies (or improvements) in the environment warrant a reconfiguration of the service to change the resourcing footprint required. We identify a configuration option C2.
3. A new collection of services referencing configuration C2 is started and brought to readiness.
4. We redirect traffic from the original set of services (using C1) to the new set (using C2)
5. We shut down the original services and release their resources back to the environment.

4.5 Investigation& Experimentation

4.5.1 Service Mesh Routing

Previously reported investigations documented our initial experiments with the Istio Service Mesh. We had envisaged this service mesh taking responsibility for re-routing traffic between individual services by leveraging Kubernetes Pod labelling. We would inform the service mesh of changes we would like made to the current QMode so that the mesh would reroute traffic to services offering the target QMode. The Service Mesh approach is predicated on multiple variations of the same software being ready and available to accept traffic. Each variation is configured (at startup) to support a different QMode. When we need to change the QMode associated with a given user then we re-route their traffic to the appropriate variation using the service mesh.

To this end, we investigated several strategies:

1. Istio header-based routing
2. Envoy header-to-metadata filter
3. Custom Envoy filters
4. External Routing Logic
5. WASM Plugins

Details on our experiments and findings can be found in the previous release of this document. We avoid repeating them here for the sake of brevity and clarity. We abandoned the service mesh approach in favour of a pure Kubernetes approach termed Rolling Updates. The key reasons for our reassessment are as follows:

- Scalability: The scheme could suit an architecture in which single application service instances handle multiple users simultaneously. This means we do not have to deploy multiple instances for each individual user but only for the entire user base. However, in cases where each user of the system has dedicated service instances (e.g., rendering engine per user) then we are faced with a very large number of redundant services running as the number of active users grows.
- Lifecycle Management: With the previous approach, we need to take ownership of starting up, managing, and retiring service variations as users come and go.
- Orchestrator Integration: to startup, move and shutdown services requires interaction with the two-level CHARITY Orchestrator. While this is feasible, it did not appear straightforward. We would require a federated service mesh operating across cloud providers.
- Protocol support: when seeking to support a distributed application that uses heterogeneous protocols and payloads to communicate between services then the mesh approach could entail a significant degree of customization.
- Troubleshooting: more tooling requires more integration and increased instrumentation to troubleshoot.
- Performance: additional proxies require additional maintenance and monitoring along with adding additional delays to the transit of traffic.

- Recovery: if we re-route traffic and encounter problems, then we need to quickly rollback to avoid painful service outage. Observing and managing this would entail additional oversight and remediation further increasing the complexity of the approach.

For the above reasons, along with the observation that a more straightforward approach presented itself, we shelved the service mesh approach in favour of a pure Kubernetes solution.

4.5.2 Rolling Updates

Rolling updates are a key feature of Kubernetes that enable updating the version of an application or its configuration with zero downtime. This process is fundamental in cloud-native environments where continuous delivery and high availability are crucial. Rolling updates work in Kubernetes work as follows:

- Gradual Replacement: When a new version of an application/service group is ready to be deployed, Kubernetes starts by creating new pods with the new version while simultaneously removing the old-version pods. This is done incrementally, a few pods at a time, according to the defined strategy in the deployment configuration.
- Health Checks: Kubernetes checks the health of new pods before proceeding to terminate more of the old ones. If something goes wrong with the newly created pods, Kubernetes halts the rollout and prevents the termination of healthy old-version pods, ensuring service availability.
- Configurable Update Policy: We can configure the update strategy in Kubernetes deployments. Two important parameters are *maxUnavailable* and *maxSurge*. *maxUnavailable*specifies the maximum number of pods that can be unavailable during the update, and *maxSurge* specifies the maximum number of pods that can be created over the desired number of pods.
- Rollback: If the rolling update encounters an error or is not behaving as expected, Kubernetes allows us to roll back to the previous version of the application. This ensures that we can quickly revert to a known good state if the new version fails.
- Continuous Delivery: Rolling updates facilitate continuous delivery by allowing frequent and controlled updates without service interruption. They support agile development practices by enabling rapid iteration and feedback.

By using rolling updates, Kubernetes provides a robust method for application deployment, ensuring that services remain available and responsive throughout the update process. It's a powerful feature that embodies the cloud-native principles of automated, reliable, and resilient infrastructure management.

Rolling updates will be the vehicle for delivering adaptation effectors with Kubernetes managing container lifecycles as shown below in Figure 43.

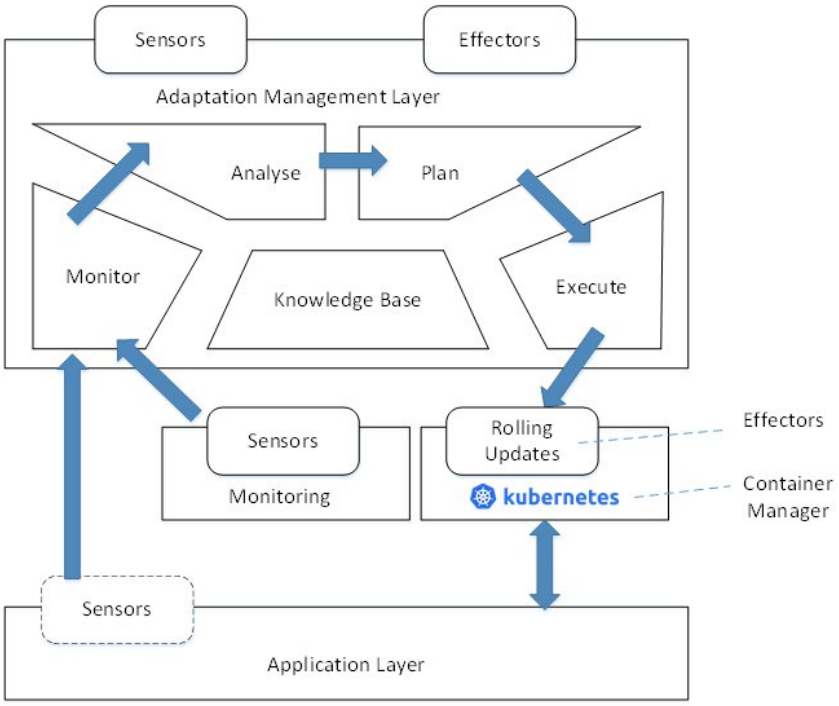


Figure 43: In our modified MAPE-K loop, Kubernetes delivers container management and effectors.

4.5.2.1 Rolling Update in Action

To test out the rolling update feature, we migrated the Collins Use Case from using docker-compose to Kubernetes. The architecture uses a Pod-per-user model to simplify scaling and ensure clean resource separation between users. The high-level model is depicted below in Figure 44.

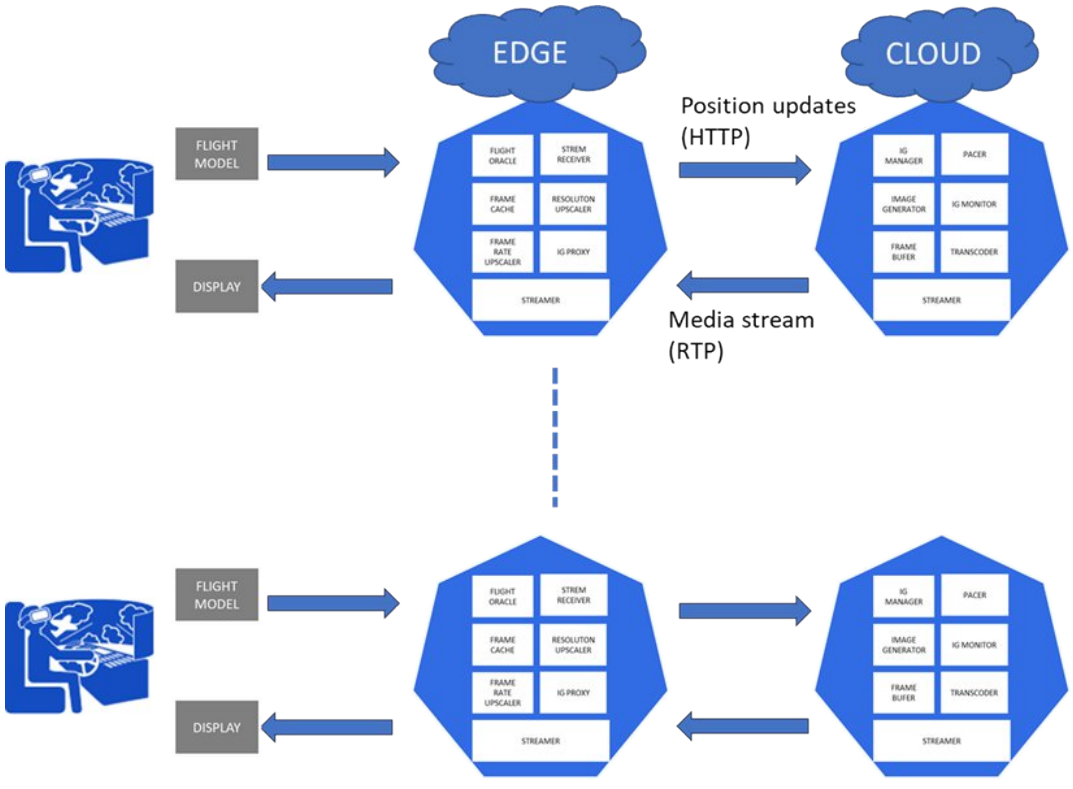


Figure 44: Dedicated pods per user in the Collins Use Case.

Changing the QMode for a user entails starting up a new pod with configuration delivering the target QMode and then switching traffic over from the original pod to the new pod. To accomplish this, we use a Kubernetes rolling update as depicted below in Figure 45.

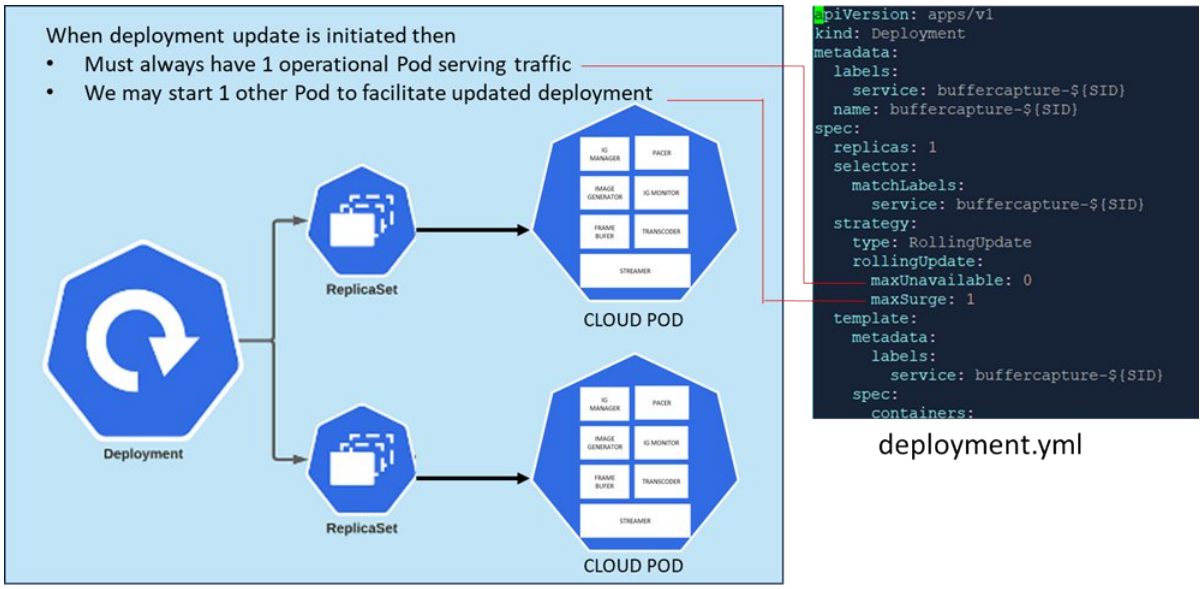


Figure 45: Rolling update of pod in Kubernetes.

We conducted experiments with a live rolling update of a cloud pod to validate the viability of using this approach. The cloud pod is challenging as it requires seamless handover of incoming HTTP traffic and outward streaming of live media streams without interruption of the user session.

4.5.2.2 Configurability

A fundamental issue is how we can collectively reconfigure a group of services. In Figure 46 below we see that such a procedure can grow complex quickly.

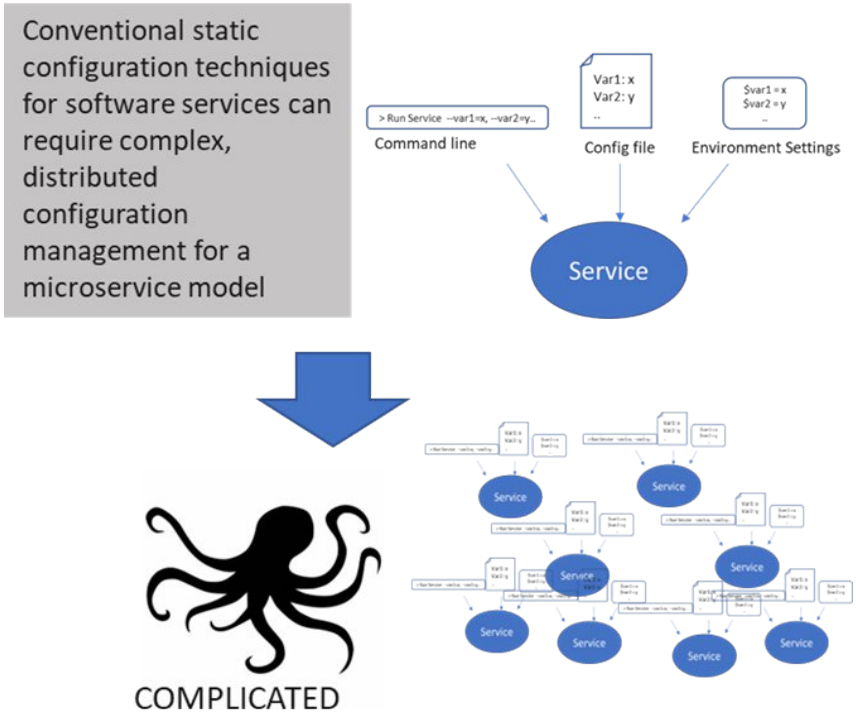


Figure 46: With many configuration routes, orchestrating change can be complex.

We began with deploying services in containers and then managing the deployment of groups of related containers using docker-compose. We leveraged the environment file capability of docker-compose to enable a single point of change in terms of configuration settings. This is depicted below in Figure 47.

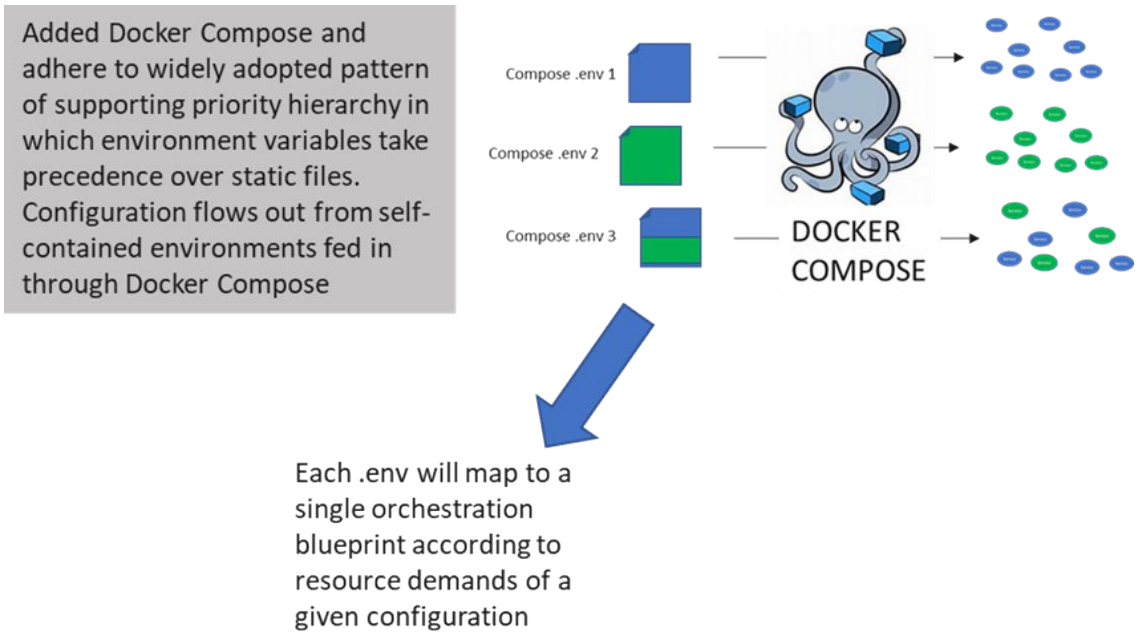


Figure 47: Centralising configuration change

In the journey to cloud-native, we migrated from docker-compose to Kubernetes. There is a somewhat similar facility available in Kubernetes known as *configmaps* which were introduced to separate configuration data from code.

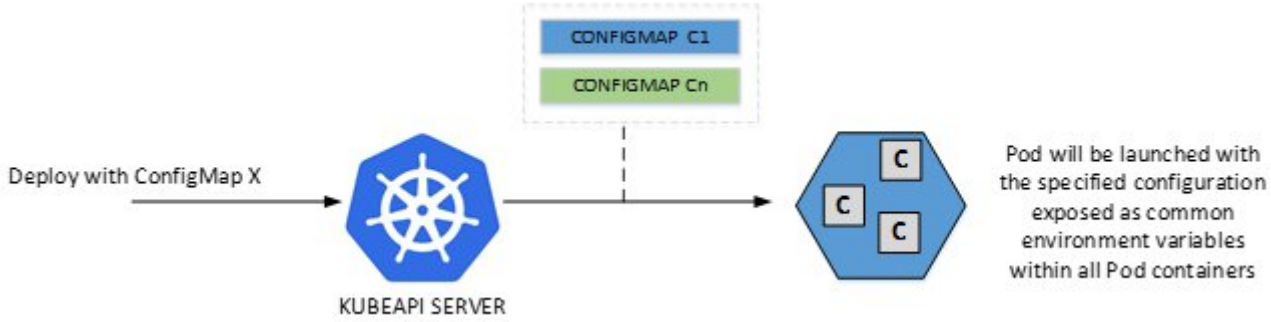


Figure 48: Kubernetes ConfigMaps can be used to reconfigure pods as required.

4.5.2.3 Supporting Diverse Configurability Channels

The Collins Flight Simulator use case provided a strong case study into how an existing application may be adapted to suit a centralised configuration scheme. The application involves numerous third-party components for which modification of the source code is not a realistic option. Figure 49 below depicts the services to be collectively deployed on the cloud for each user and summarises how each service can be configured.

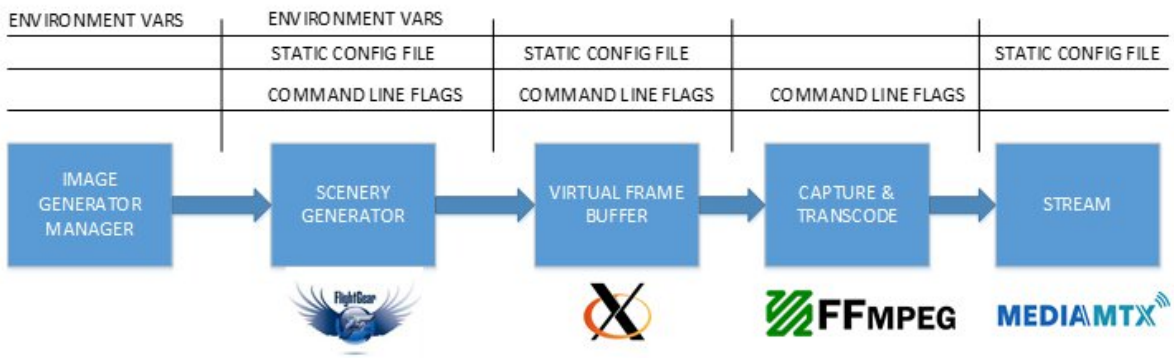


Figure 49: Collins use case configuration landscape.

The services work closely together and changing the configuration in one service can have knock on effects for others. For example, changing the resolution displayed by the Scenery Generator has repercussions for the configuration of the Virtual Frame Buffer which in turn has repercussions for the Capture & Transcode service. The configuration of all has be done in lockstep or else we risk placing the services into an inconsistent state. For services that can only be configured through command line flags, we introduced shell scripts to launch those services within containers and then made those launch scripts configurable through environment variables. For services that require static configuration files then we assemble a selection of pre-configured files and select the appropriate file according to environment variable settings at runtime. The scheme described is presented below in Figure 50.

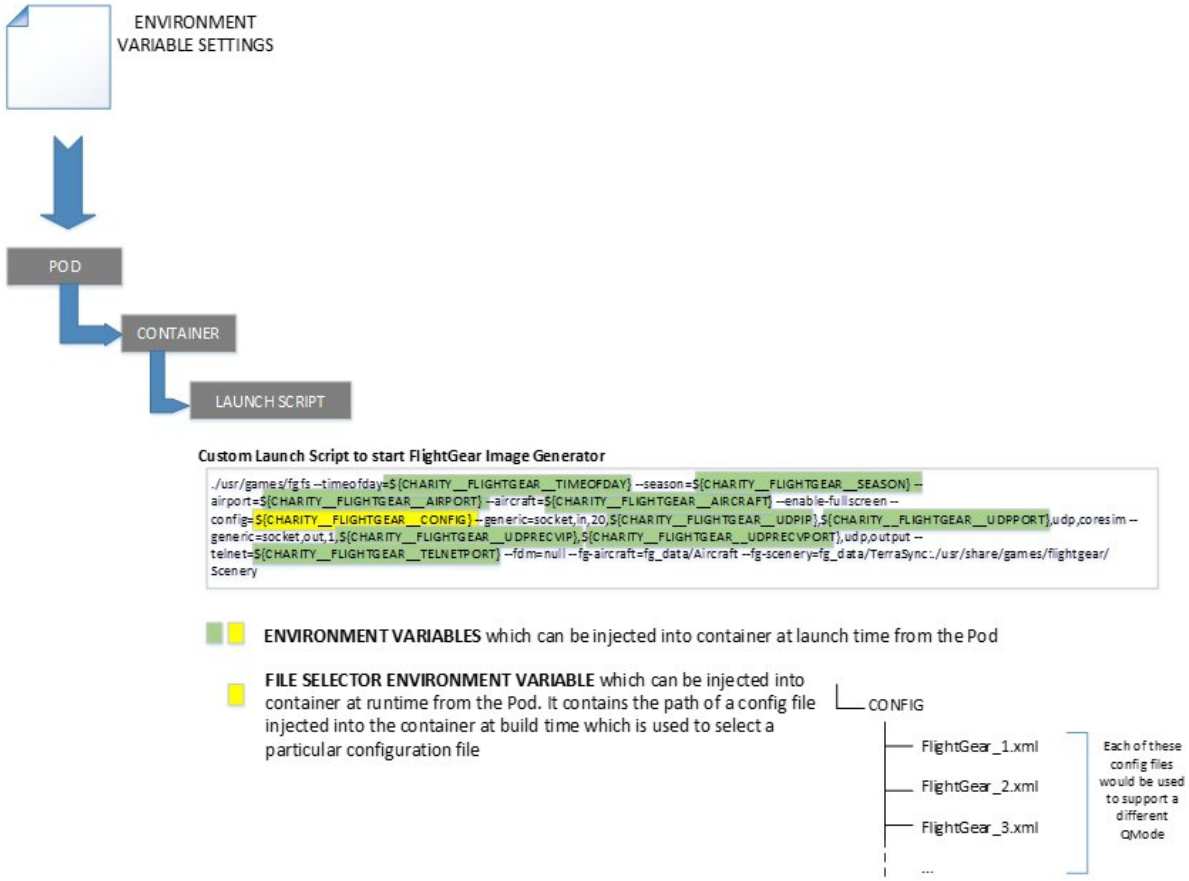


Figure 50: Example of how a single configuration set can be injected into Pod and effect change even in applications that do not directly support configuration through environment variables.

Kubernetes*configmaps* then become the Knowledgebase within our MAPE-K loop by providing a catalogue of QMode states that the application can be moved into as depicted below in Figure 51.

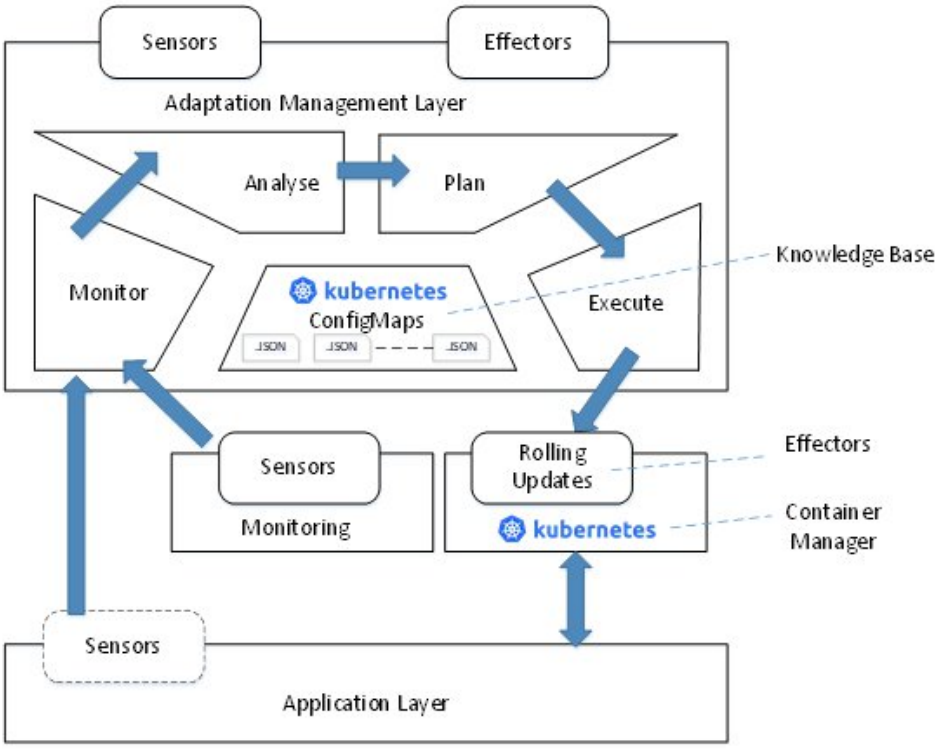


Figure 51: Kubernetes ConfigMaps form our application knowledgebase.

4.5.2.4 Adaptation Tactics

We examined a use case in depth – the Collins Aerospace Flight Simulator (UC3-2 Manned-Unmanned Operations Trainer Application) and sought to identify whether purely configurational changes could be executed which would deliver tactics that we could bring into play to deal with resource deficiencies observed from resource monitoring (see Figure 52 below).

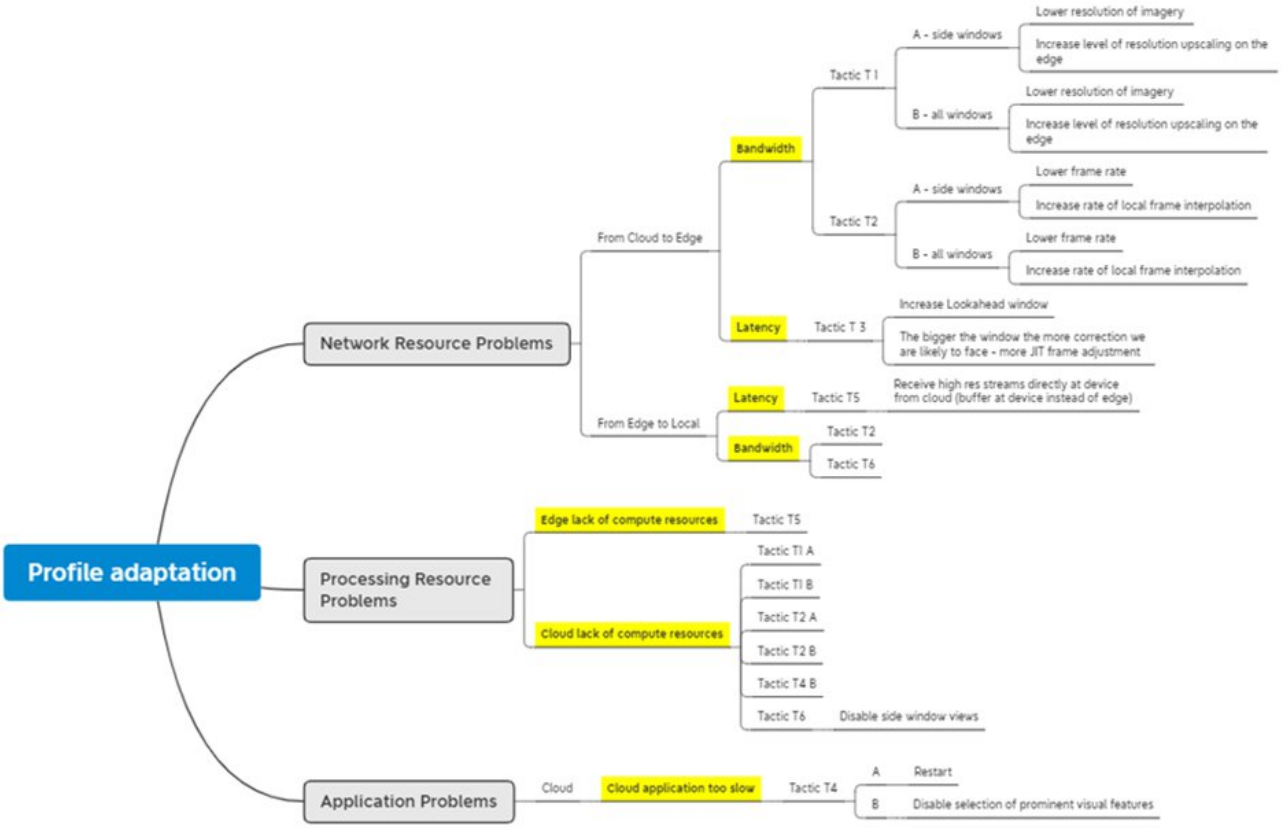


Figure 52: Configurability options to deliver adaptability tactics.

We can observe how the same tactic can be used in multiple scenarios. Tactics 1 and 2, for example, can be brought into play if we need to reduce bandwidth needs between the edge and cloud or free up compute resources on the cloud.

It became clear during analysis that changing the configuration of one service regularly requires changes to others to compensate or adapt to the new execution landscape. We cannot just lower the resolution generated on the cloud in isolation as the end user would experience a catastrophic drop in their Quality of Experience. We must simultaneously enable resolution upscaling on the edge on compensate. We see this need to deal with collateral effects of changes to how a single service operates repeated elsewhere.

Naturally, not all applications can lower their resolution on the cloud and have the necessary allowances in their design to compensate through upscaling elsewhere. Indeed, we expect other applications to have opportunities not offered by the flight simulator use case.

4.5.2.5 Resolution Modification

The resolution we adopt for cloud rendering has significant effects on the physical cloud resources that we require. Bigger resolutions require more pixels to be generated which requires more GPU, more memory and more bandwidth to transfer.

4.5.2.6 Feature Enablement

Enabling or displaying GPU intensive features can have significant repercussions for resource usage. In looking at the Collins Use Case, we identified a range of rendering effects that could be toggled at startup. Examples include:

- Generation of random buildings, roads, pylons and vegetation to increase scenery density
- The nature, density, and visibility range of clouds
- Sophistication of runway lighting
- Precipitation and smoke particle effects
- Overall shader quality (increased quality produces increased realism)

The enabling and disabling of advanced graphical effects had repercussions beyond the GPU resources needed to render them. With increased rendering activity per frame, then the visual output became more dynamic. Precipitation and smoke, for example, increased the amount of visual change between frames which reduced the amount of compression that could be achieved with video codecs (which rely on just capturing changes between frames). This resulted in increased bandwidth usage even though the resolution and frame rate remained the same.

4.5.2.7 Frame Rate

Increasing the frame rate produced on the cloud increases the bandwidth which needs to be made available to transmit the video stream. Experimenting with various frame rates (10, 20 and 60) showed the pressure that it exerts on network capacity as we will see later in this section.

4.5.2.8 Dynamically moving resource dependency between edge and cloud

What became clear during experimentation is that the most efficient place to produce high quality media streams is at the source. Reducing the quality at source with the goal of recovering this loss at the edge through upscaling is significantly more expensive in terms of overall GPU, CPU and memory consumption when viewed as a whole across the cloud and edge. The tradeoff is about bandwidth, Below we see a high-level summary of key metrics if we generate 1K resolution on the cloud

Figure 53: Generate high quality on the cloud.

Above we can see that we can generate the target stream with 20% of the available GPU computation resource and consume 2.6 MB/sec of bandwidth

If we instead generate low quality on the cloud and see to try and recover that quality on the Edge then the high level metrics are presented below in Figure 54.

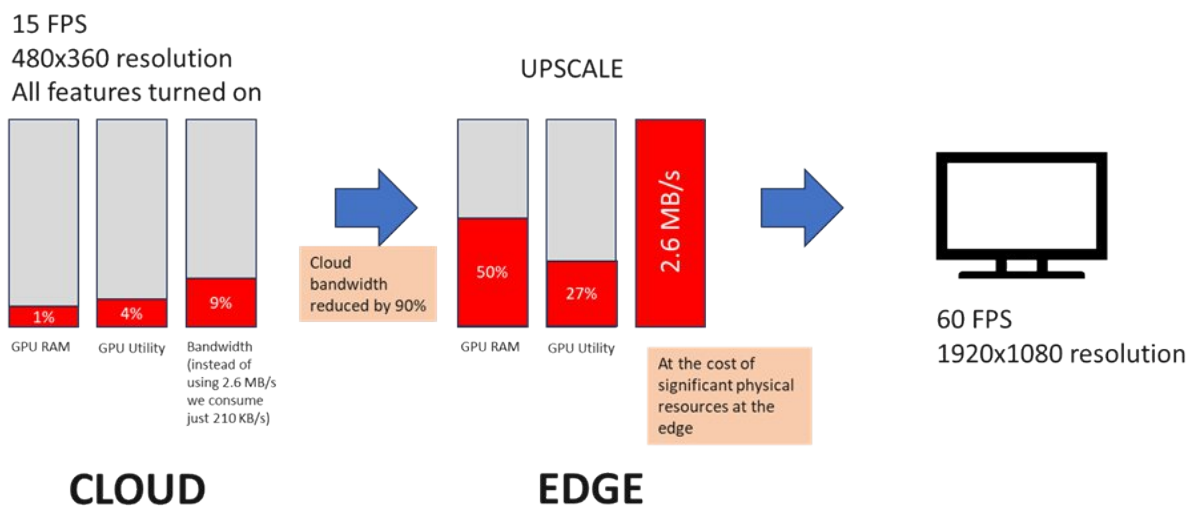


Figure 54: Generate low quality on the cloud and seek to recover quality at the edge. Significant bandwidth reductions but also significantly increased resource usage overall.

4.5.2.8.1 Effects of adaptation

We ran various experiments to observe the variations in resource consumption of the flight simulator Cloud Pods under different configurations and results are shown below in Table 18. The flight simulator can be run with just a single window (showing the scenery straight ahead) or multiple windows for left and right views[23]. Low QModes signify operation with disabled advanced graphical features (smoke, shadows, etc.) while high QModes signify operations with all features enabled.

The bandwidth reflects the amount of data being sent from the transcoder to the streamer.

Table 18: Resource usage profiles across various QModes.

QMode	GPU Memory usage (MiB)	GPU utilization	Frames Per Second (FPS)	Resolution	Bandwidth (MB/sec)
Low single window	105	2%	10	848x480	0.1
Low single window	120	4%	20	848x480	0.16
Low single window	122	13%	60	848x480	0.38
High single window	208	3%	10	848x480	0.51
High single window	208	6%	20	848x480	0.75
High single window	208	18%	60	848x480	1.2
Low multiple window	270	8%	10	848x640	0.3
Low multiple window	284	16%	20	848x640	0.35
Low multiple window	316	37%	60	848x640	0.5
High multiple windows	675	13%	10	848x640	1.35

[23] We encountered a persistent problem with the display of the right-hand window with FlightGear that we have not yet succeeded in solving. The results with multiple windows only represent two windows.

High multiple windows	675	29%	20	848x640	1.75
High multiple windows	675	33%	60	848x640	2.4
High single window	268	6%	10	1920x1080	2.1
High single window	268	16%	20	1920x1080	2.6
High single window	268	19%	60	1920x1080	3.2

We can observe from the results that all attributes – resolution, frame rate and feature enablement - play a significant role in terms of compute and bandwidth resources and thus all should be considered when formulating QMode configurations.

4.5.3 Monitoring & Alerting

A fundamental aspect of adaptation is knowing when it is required. In line with the CHARITY architecture, we used Prometheus, custom exporters, Grafana and the Prometheus Alert Manager to deploy supporting infrastructure to monitor resource usage, raise alerts when appropriate, and instigate an adaptation. This work was done in close conjunction with the Collins Use Case. Metrics are reported through custom exporters which can be deployed independently of the application or integrated into an application. Our primary focus was on the former where we deployed custom exporters for the Cloud pods in the Collins Use Case. This is summarized below in Figure 55.

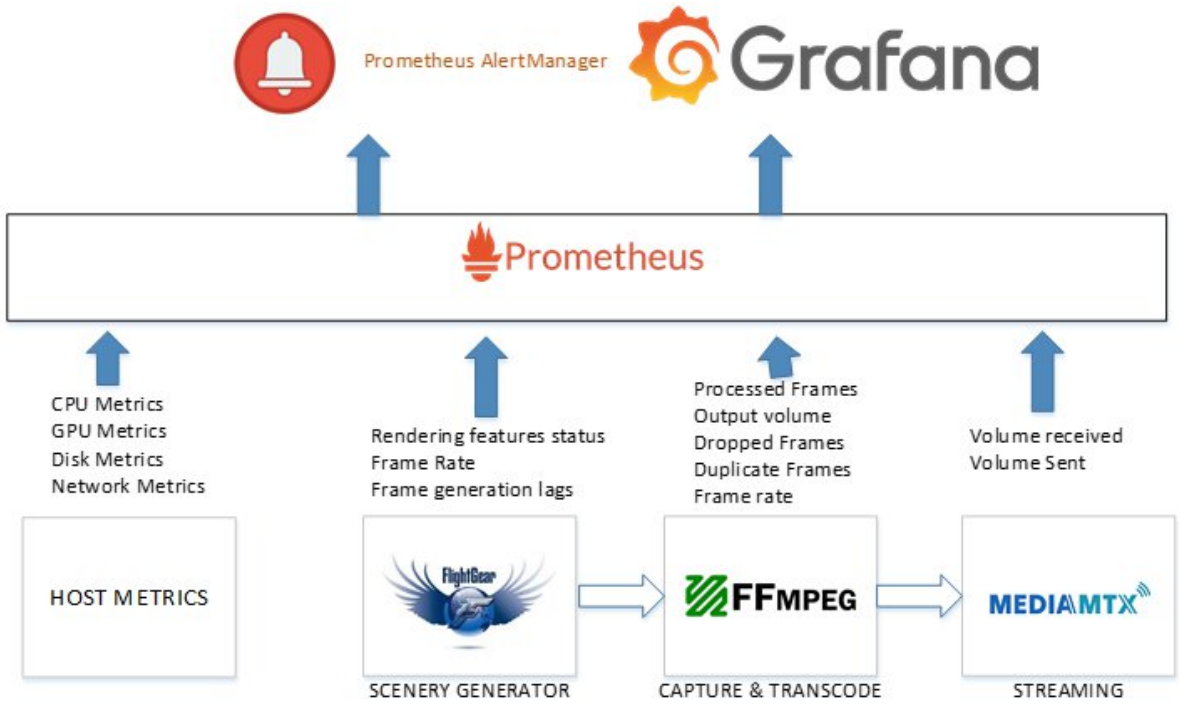


Figure 55: Custom exporters deployed for cloud pod monitoring.

The integration of custom exporters enables monitoring while also giving us crucial insights into behaviour using graphical dashboards with Grafana and the ability to configure custom alerts in response to key indicator changes as depicted below in Figure 56.

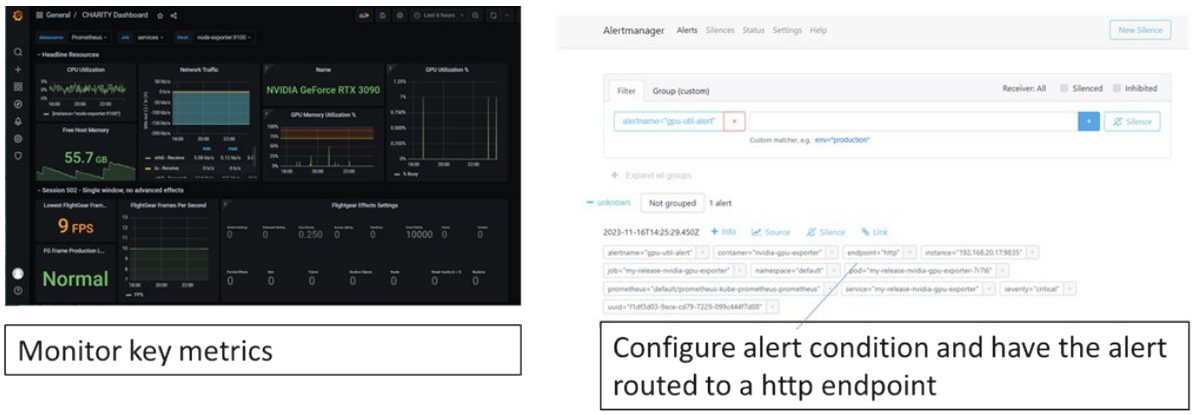


Figure 56: Leveraging the CHARITY monitoring technology stack to monitor, analyse and react.

This brings us to a more complete MAPE-K loop where we can show Prometheus and alert manager playing their roles as shown below in Figure 57.

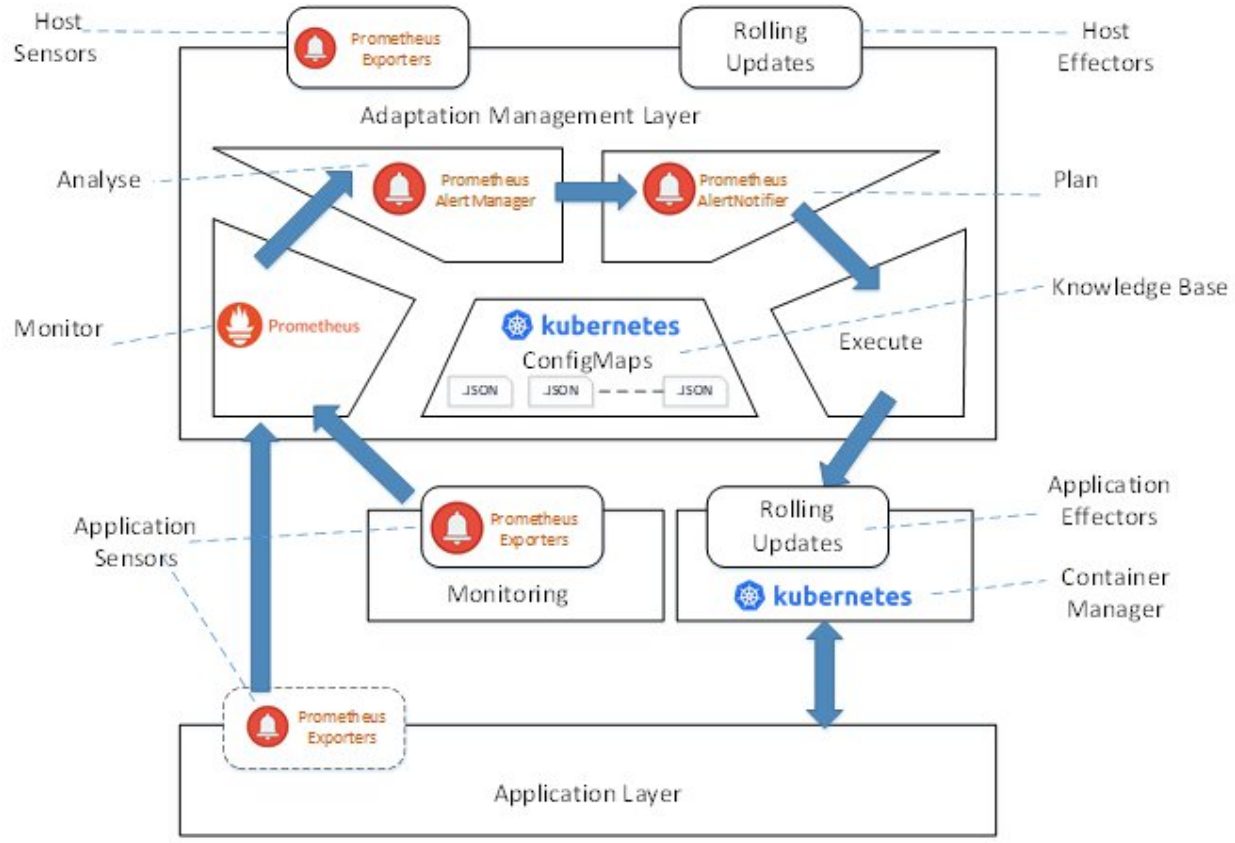


Figure 57: Adapted MAPE-K loop showing roles fulfilled by Kubernetes & Prometheus.

As can be observed above in Figure 57, we show Host and Application sensors both being delivered with Prometheus Exporters. In the case of Host Sensors, we employ application-independent Prometheus exporters whose role is focused exclusively on monitoring host resource metrics (such as CPU, GPU, disk, network). Application Sensors are application dependent. They may be embedded into application code itself (to monitor the number of active users for example) or as standalone tools that monitor application behaviour through application APIs or log files and report onwards to Prometheus.

4.5.4 Adaptation Execution

In Figure 58 below we present a high-level view of how the adaptation process currently executes.

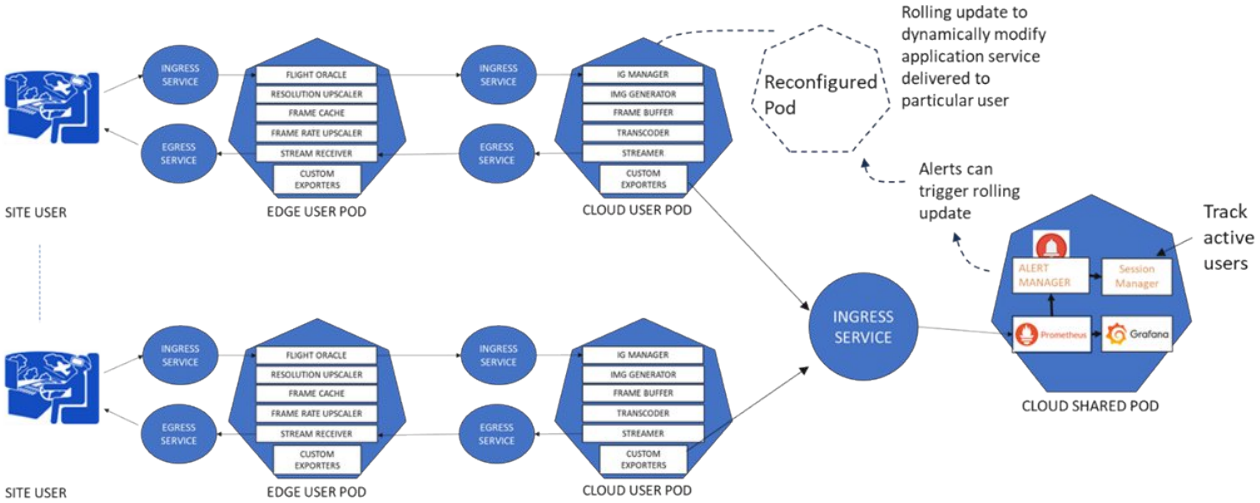


Figure 58: Dynamic software adaptation driven by monitoring.

Kubernetes ingress and egress services are instrumental in maintaining a seamless connectivity experience from the perspective of service clients. Initial experiments revealed the need for Kubernetes readiness probes to delay the switch from active pod to replacement pod. Such probes ensure that the replacement pod is fully bootstrapped, initialized, and ready to take over. Without this step we experienced jarring breaks in service as the handover was happening before the new services were ready to take over. In Figure 59 below, we present a rolling update in action where we move a user from a fully featured, resource intensive experience to a reduced feature experience.

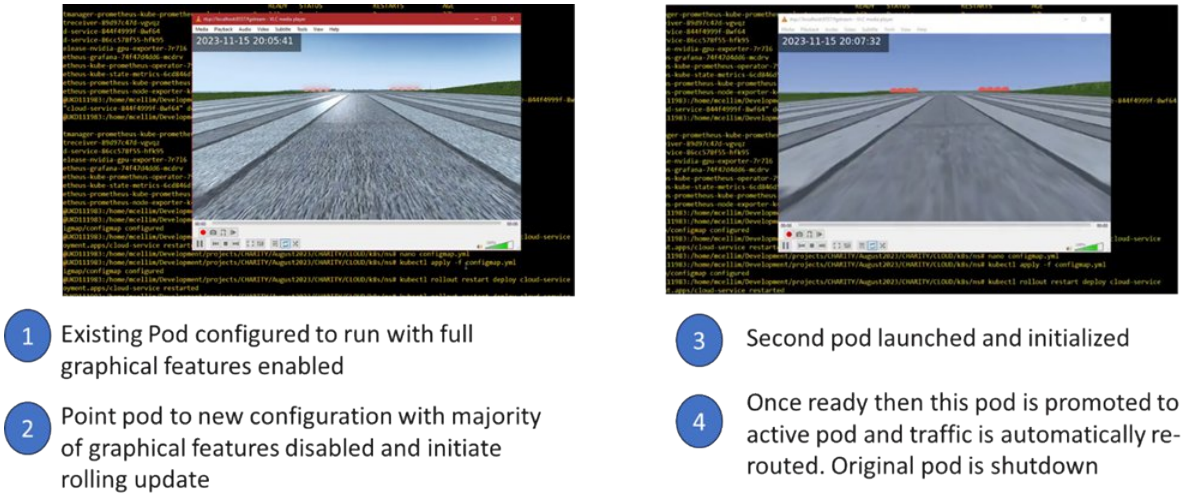


Figure 59: Dynamic Software Adaptation using rolling updates for the Collins use case.

5 Enabling XR technologies

5.1 Enabling Advanced Computing Mechanisms: The Virtual Machine and GPU Challenges

This section describes the incorporation and the role of advanced computing mechanisms, namely Virtual Machines (VMs) and Graphics Processing Units (GPUs), in the XR deployment and orchestration process. The benefits of such mechanisms are many-fold. *Virtual Machines* emerged in the past as a first step towards achieving hardware abstraction. Indeed, encapsulating the entire software stack in a VM facilitates consistent and reproducible environments, including for Extended Reality (XR) applications. Nevertheless, throughout various fields, Cloud-Native and microservice-oriented approaches are gaining prominence, offering additional layers of flexibility and scalability. Cloud-native emphasizes deploying applications as smaller and more manageable micro-services (i.e., the containers) and uses orchestration tools like Kubernetes for their lifecycle orchestration. This modular approach facilitates the development and deployment of XR applications, allowing for a more efficient and dynamic workflow. Indeed, the portability brought by Cloud Native architectures is particularly relevant in highly dynamic and distributed XR scenarios, as CHARITY considers. That being said, as in other domains, the usage of Virtual Machines is still a today's reality. Whether dealing with intricate legacy code or unsupported third-party libraries, their use remains essential. Hence, their usage creates the challenge of seamlessly orchestrating both (i.e., VMs and containers). Namely, how to ensure their coexistence, communication, and compatibility with the various tooling. In the opposite direction, *GPUs* are progressively used for data-intensive processing tasks such as Machine Learning or physics rendering engines. Therefore, their support should also be considered taking into account a comprehensive XR orchestration process. For instance, an XR orchestration solution should be able to recognize the GPU requirements of specific components and strategically plan the deployment process accordingly. The orchestration solution should also be capable of deploying infrastructure environments appropriately, including additional infrastructure settings. Similarly, adaptations in monitoring and decision workflows are necessary for incorporating both VMs and GPUs requirement and to ensure an optimal component deployment. In CHARITY, the need for VMs and GPU support can be seen in use cases such as the Holographic assistant or the VR Medical Training.

From the CHARITY orchestration perspective, VMs and GPUs introduce four conceptual changes. First, TOSCA definitions derived from AMF should be able to characterize such additional requirements. For instance, the need of a GPU-enabled node for running a given component. Then, Low-Level Orchestrator should be able to understand and translate them into its internal CRD definition. Moreover, the cluster bootstrapping process, should also consider the correct installation of additional components, both for VMs and GPUs. Furthermore, specific VMs and GPU metrics should also be exposed for the remaining monitoring components of CHARITY which will be leveraged by the AI-based algorithms. Figure 60 depicts the overall interaction between different components.

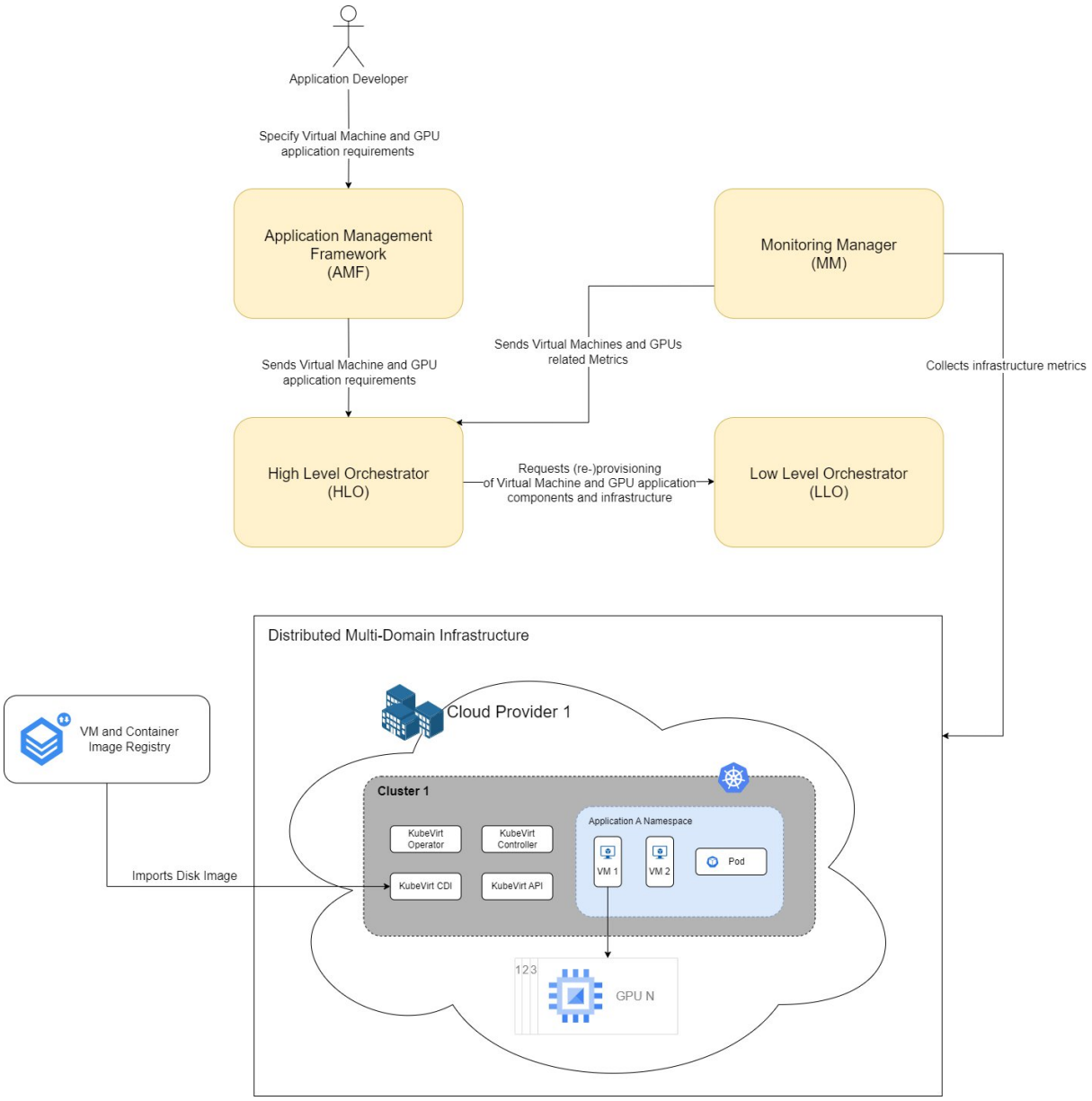


Figure 60: VM and GPU orchestration support architecture.

The rest of this section is split into two key parts, one devoted to VMs and another to GPUs. For each, we analyse the state of the art, compare the existing solutions and detail the experimental work for evaluating their current support in Kubernetes.

5.1.1 Virtual Machine Support in Kubernetes Environments

Effectively managing a hybrid infrastructure involving container-based applications alongside VM remains an open challenge; Mavridis et al. [54] discuss an approach to overcome this problem and ensure both can co-exist side-by-side within the same infrastructure and tooling. In broad terms, such an approach consists of having them being orchestrated by the same platform and different types of workloads (VM or container-based) scheduled, hosted and managed in a unified way.

Lee, J. et al. [55] integrated the Kubevirt platform[24] as part of their Management and Orchestration (MANO) proposal, where they translate Virtual Network Functions (VNF) into both containers or VMs

[24] https://kubevirt.io .

as required. Other authors also suggest the usage of Virtlet[25] for bringing VM support to Cloud Native environments and orchestration platforms [56] . Mavridis et al. [57] compare different container routines for addressing multi-tenant workloads, including using Kubevirt for running Unikernels and Virtual Machines on top of Kubernetes clusters. Despite being more resource-intensive, they achieved comparable performances and low deployment times when utilizing Kubevirt and VMs compared to other container runtimes. Their analysis touches on the performance of various solutions, a consideration that holds relevance for the demanding requirements of XR applications.

Indeed, Kubevirt and Virtlet are both open-source technologies for supporting the coexistence of VMs and containers on top of the same computing infrastructure. Whereas both serve the same purpose, they differ in their implementation[26]. Kubevirt works as an extension that uses Kubernetes Custom Resources (CR) to define Virtual Machines. Whereas Virtlet is a Container Runtime Interface (CRI) implementation to run/interpret Virtual Machines in the same way as other Kubernetes resources (e.g., pods). When comparing both technologies, Virtlet provides a simpler and lighter approach, whereas Kubevirt provides more flexibility regarding the virtual machines' configuration and storage. Moreover, Virtlet, with its own CRI, builds on existing higher-level Kubernetes objects such as StatefulSets or Deployments, which makes it possible to think and map virtual machines to existing Kubernetes resource abstractions. Furthermore, Virtlet supports Kubernetes networking and multiple CNI implementations (e.g., Calico, Weave, and Flannel). On the other hand, Kubervirt was designed as a hypervisor-agnostic solution which builds on top of existing hypervisors such as Kernel-based Virtual Machine (KVM), leveraging its own Custom Resources and controllers, which can enable additional customization/configuration options. For instance, Kubevirt offers an option of using PVCs as disks via the Containerized Data Importer (CDI)[27], which is implemented by Kubernetes itself. Overall, both intend to provide a Kubernetes native experience of managing Virtual Machine definitions either by using kubectl or Kubernetes API. Regardless of the debate on whether it is preferable to have a new CRI versus the integration of existing and widely-used hypervisors, as of the time of writing, the last stable release of Virtlet was in 2019. Whereas, KubeVirt had several releases in 2023.

Hence, we choose KubeVirt as the technology to evaluate within the CHARITY project due to its active state, broader adoption and support within the Kubernetes community. We plan to leverage KubeVirt's flexibility and container-native experience within Kubernetes to seamlessly integrate virtual machines with containerized workloads.

Figure 61 depicts an overview of the Kubevirt architecture and main components introduced on Kubernetes.

[25] https://github.com/Mirantis/virtlet .

[26] https://www.mirantis.com/blog/kubevirt-vs-virtlet-comparison-better .

[27] https://github.com/kubevirt/containerized-data-importer .

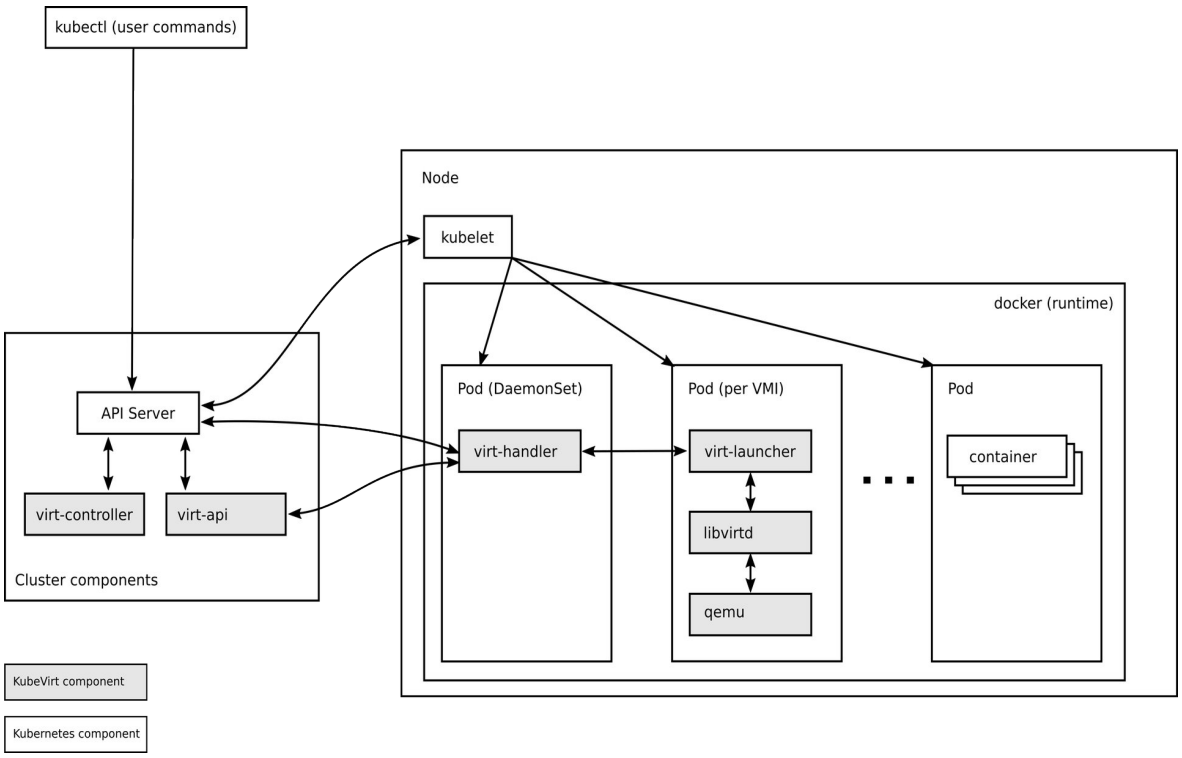


Figure 61: KubeVirt architecture[28].

In Kubevirt, each VM instance is seen as a Kubernetes CR -- a Virtual Machine Instance (VMI) in this case, as depicted in Figure 61. KubeVirt uses a dedicated controller (i.e., *virt-controller*) for detecting and reacting to changes in the VM resources and interacting with Kubernetes API. It also uses a dedicated component designated as *virt-handler* on each node to interface (via *libvirtd*) with each VM and the underlying hypervisor. Notice, that the setup of all of these components becomes a non-functional requirement, implying that the CHARITY orchestration solution should be able to install and configure them as required on the target computing clusters. Like Kubernetes pods, Kubevirt VMs can use Kubernetes Services and Ingresses to expose application endpoints allowing the connection with other applications and/or users.

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: debian
  labels:
    special: debian-vm
spec:
  runStrategy: Always
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: debian
    spec:
      domain:
        devices:
          disks:
            - name: operative-system
              disk:
                bus: virtio
            - name: cloudinitdisk
```

Figure 62: Example of Kubevirt VMI definition.

[28] https://kubevirt.io/user-guide/architecture/

Kubevirt focuses on Kubernetes integration and the means to provide the same tooling as any other Kubernetes resource while it relies on widely used hypervisors for the virtualization itself. Indeed, such an architectural approach helps to mitigate the challenges of orchestrating both types of workloads without needing different workflows or separate management tools. The external hypervisor strategy of Kubevirt also facilitates the use of operating system-dependent applications and extends compatibility to legacy applications that were previously virtualized and deemed impractical to containerize. Moreover, Kubevirt provides a live migration feature, which brings additional value to highly dynamic environments, as considered in CHARITY, where it is possible to migrate Virtual Machines without disrupting services running within them.

In the following, we detail the experiments executed to assess Kubevirt functionalities and their usage within the CHARITY project.

Virtual Machine Deployments through KubeVirt

In the first experiment, we conducted a series of tests to deploy Virtual Machines within a Kubernetes environment, utilizing KubeVirt. These initial steps were performed to enhance our understanding of KubeVirt's capabilities in managing the lifecycle of Virtual Machines. We also use these experiments to assess the connectivity between containers and KubeVirt Virtual Machines, along with the necessary requirements for their integration into the orchestration solution.

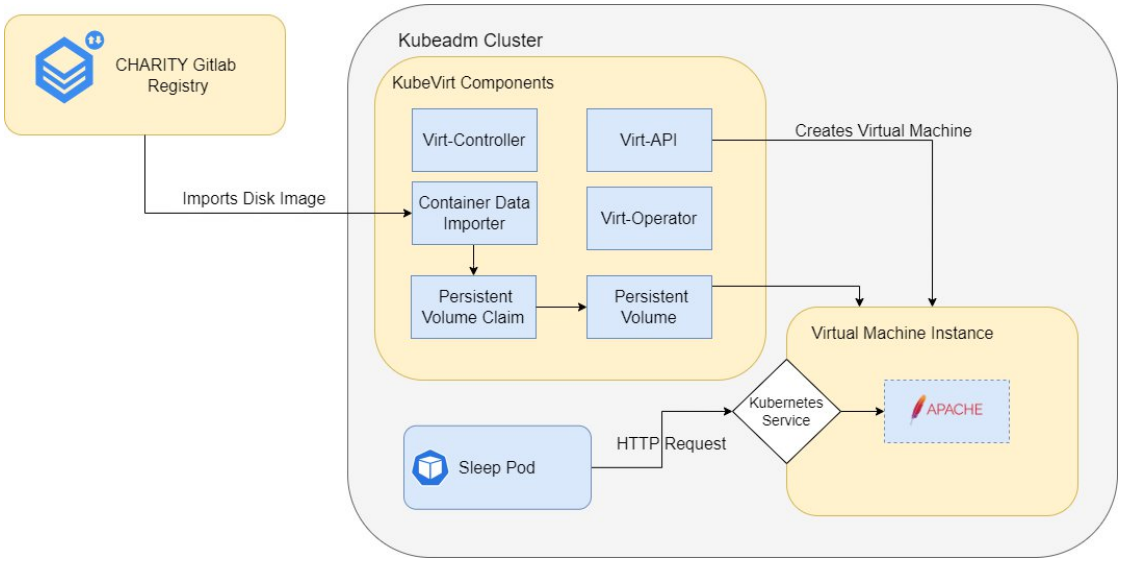


Figure 63: KubeVirt experimental scenario.

Figure 63 shows the scheme of the experimental scenario reproduced within the CloudSigma testbed. It consists of a single-node Kubernetes Cluster deployed using *kubeadm* as the bootstrap provider. The installation of Kubevirt followed the steps outlined in the official documentation[29].

Virtual Machine creation was performed using a Virtual Machine image coming from a private registry, the Containerized Data Importer (CDI) from Kubevirt and Persistent Volumes (PVs) from Kubernetes. CDI is a utility designed to enable PVs as data volumes of the Virtual Machines. For the sake of proof of concept, we used a standard Linux/DebianOS 12 image hosted in the GitLab registry of CHARITY and a sleep pod (for testing the communication). Next, we specified the VMI resource (i.e., virtual machine definition) by defining the Virtual Machine properties and image location. The VM was started through

[29] https://kubevirt.io/quickstart_cloud/

Kubectl and *Virtctl* (i.e., KubeVirt's CLI[30]) was used to check the VM instance status, confirming the success of the VM deployment.

To validate the communication between pods and VMs, an *Apache2* web server was installed within the deployed VM, featuring a simple web page. To assess the communication, first, we exposed the web page by attaching a Kubernetes service -- a similar process for container service exposure. Later, we used a lightweight sleep container running within a pod and a *curl* command to test their communication through HTTP.

Both deployment and communication testing were successful. Yet, more than testing the successful communication between the pod and the VM instance, this first experiment was pivotal in understanding how the process could be later realized and automatized from the orchestrator's standpoint. In CHARITY, XR developers, which rely on AMF and image registry to upload their application components, can use the same approach for uploading VM machine images. For communication, although additional evaluation is needed, by using Kubernetes service resources to expose their communications, we can expect similar support in Virtual Machines. Indeed, KubeVirt documentation states their support for ClusterIP, NodePort and LoadBalancer types of services.

5.1.2 GPU support in Kubernetes environments

This section delves into the challenge of how GPUs can be used within Kubernetes and later integrated into the CHARITY orchestration solution. From AI-based workloads to enabling High-performance computing (HPC) applications, GPUs in Kubernetes, in tandem with KubeVirt, extend the array of possible workloads. Whereas with VMs, we are mainly focused on enabling the otherwise not possible VM-based workloads when considering GPUs, we mainly focus on bringing performance and efficient exploitation of hardware-specific resources. Bringing GPU to Kubernetes can be defined into two key challenges: the additional changes, interfaces and components involved in the software stack; and how the scheduling and sharing occur among GPUs [58][59][60]. In this section, we expand the first one.

GPU-enabled infrastructure highly depends on the underlying vendor hardware (e.g., AMD, NVIDIA) and, thereby, their specific drivers. This way, like the underneath hypervisor components in Virtual Machines, GPU vendor-specific drivers become non-functional for such setups. Considering the existing CHARITY testbed facilities, we choose to focus on NVIDIA GPU environments.

[30] https://kubevirt.io/user-guide/operations/virtctl_client_tool/

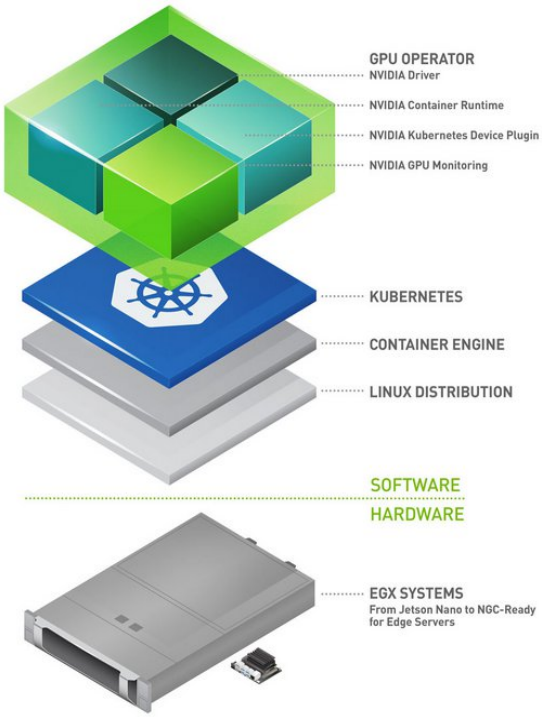


Figure 64: NVIDIA GPU components for Kubernets[31].

Figure 64 shows the main components involved for NVIDIA GPUs, including the NVIDIA GPU driver, the NVIDIA container runtime, the Kubernetes device plugin, the Data Center GPU Manager (DCGM) monitoring agent, and GPU Feature Discovery. NVIDIA provides NVIDIA Container Toolkit[32], which contains a container runtime library and utilities to expose GPU hardware capabilities to allow GPU-accelerated containers. At the cluster level, the NVIDIA device plugin[33] provides the facilities for exposing, keeping track, and running GPU-enabled containers. Hence, Kubernetes CRI should be compatible with the NVIDIA Container Toolkit. Moreover, Node Feature Discovery is used for exposing GPU characteristics as a set of Kubernetes labels. This is especially relevant for scheduling algorithms, as they benefit from precise knowledge of the cluster capabilities, in this instance, those related to GPUs. This information becomes an integral part of their decision-making logic. Furthermore, NVIDIA also provides a GPU operator to facilitate the installation of NVIDIA components into the cluster. Such NVIDIA GPU Operator simplifies node configuration by autonomously managing most of the setup aspects, though it's important to note that not all NVIDIA GPUs are supported. In other words, the GPU model can be seen as a non-functional requirement according to the NVIDIA GPU Operator compatibility list[34]. Finally, there is the DCGM component to interface with Prometheus' monitoring capabilities and gather specific GPU telemetry data.

To evaluate the GPU support, *two scenarios* were devised. The first scenario focuses solely on the usage of GPU containers in a Kubernetes cluster. The second one combines the usage of Virtual Machines and GPU to enable the most complex scenario of supporting GPU-enabled Virtual Machines in Kubernetes. For the first experiment, we used a single-node Kubernetes cluster on top of a bare-metal server equipped with an NVIDIA GPU GeForce GTX 1660 SUPER. The setup of GPU-related components and monitoring followed the official documentation aforementioned.

[31] https://developer.nvidia.com/blog/nvidia-gpu-operator-simplifying-gpu-management-in-kubernetes/

[32] https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/index.html

[33] https://github.com/NVIDIA/k8s-device-plugin

[34] https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/platform-support.html

In this first experiment, we focused on evaluating the challenges of setting up containers within a GPU-enabled Kubernetes cluster. For comparison, we run a sample training Tensorflow-based application[35] on CPU-only and GPU-enabled containers and compare their performance. Figure 65 illustrates the time necessary to train each batch on CPU and GPU. Whereas, Figure 66 depicts the CPU and GPU usage of each one.

```
******** On CPU: *********

batch size 8: 7500/7500 - 135s 18ms/step - accuracy: 0.9323
batch size 16: 3750/3750 - 102s 27ms/step - accuracy: 0.9786
batch size 32: 1875/1875 - 87s 47ms/step - accuracy: 0.9877
batch size 64: 938/938 - 83s 88ms/step - accuracy: 0.9926
batch size 128: 469/469 - 83s 176ms/step - accuracy: 0.9949
batch size 256: 235/235 - 83s 355ms/step - accuracy: 0.9966
batch size 512: 118/118 - 82s 697ms/step - accuracy: 0.9973
cpu_times:  [134, 102, 87, 82, 82, 83, 82]
```

```
******** On GPU: *********
batch size 8: 7500/7500 - 24s 3ms/step - accuracy: 0.9321
batch size 16: 3750/3750 - 13s 3ms/step - accuracy: 0.9781
batch size 32: 1875/1875 - 8s 4ms/step - accuracy: 0.9883
batch size 64: 938/938 - 6s 6ms/step - accuracy: 0.9930
batch size 128: 469/469 - 5s 10ms/step - accuracy: 0.9955
batch size 256: 235/235 - 4s 17ms/step - accuracy: 0.9972
batch size 512: 118/118 - 4s 34ms/step - accuracy: 0.9978
gpu_times:  [24, 12, 7, 5, 5, 4, 4]
```

Figure 65: GPU vs CPU times from each training batch.

[35] https://www.tensorflow.org/api_docs

Figure 66: CPU and GPU usage from Tensorflow.

The faster result times on the GPU application prove the functional behaviour of the setup and the efficient exploitation of GPU-enabled containers. Moreover, Prometheus and Grafana proved to be a good choice when it comes to the ability of monitoring the performance and behaviour of resource usage, in particular the GPU-specific metrics. As discussed before, such metrics are pivotal for the orchestration solution.

For the second scenario, our objective was combine the previous experiments and have a KubeVirt VM instance with GPU. For that, we specified a KubeVirt VM with Windows 10 and GPU-passthrough. The Kubernetes cluster was installed as described in the previous experiment. For the VM definition three datavolumes were used: one for the OS installation ISO, another for the VirtIO drivers and the last one for the storage of the VM itself. For the OS installation we set up a local docker registry which we used to store a Docker image containing the Windows ISO. The second volume contained the *virtIO* drivers which are used by Kubevirt for interfacing with guest OS. illustrates the KubeVirt GPU Passthrough components. VirtIO drivers were later load as part of Windows installation (see Figure 68). Finally, the third volume was configured as a PersistentVolume (PV) to host the VM disk.

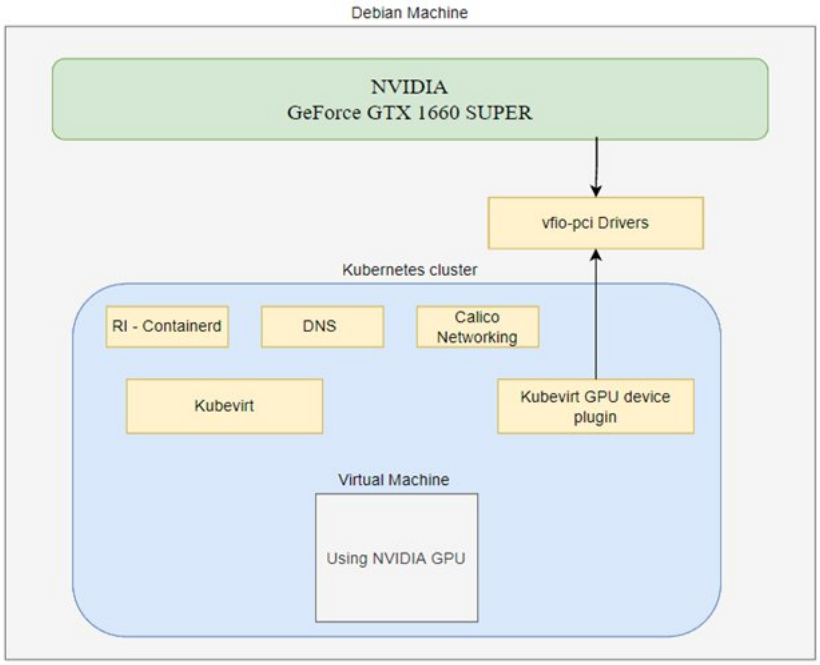


Figure 67: KubeVirt + GPU Passthrough experimentation scenario.

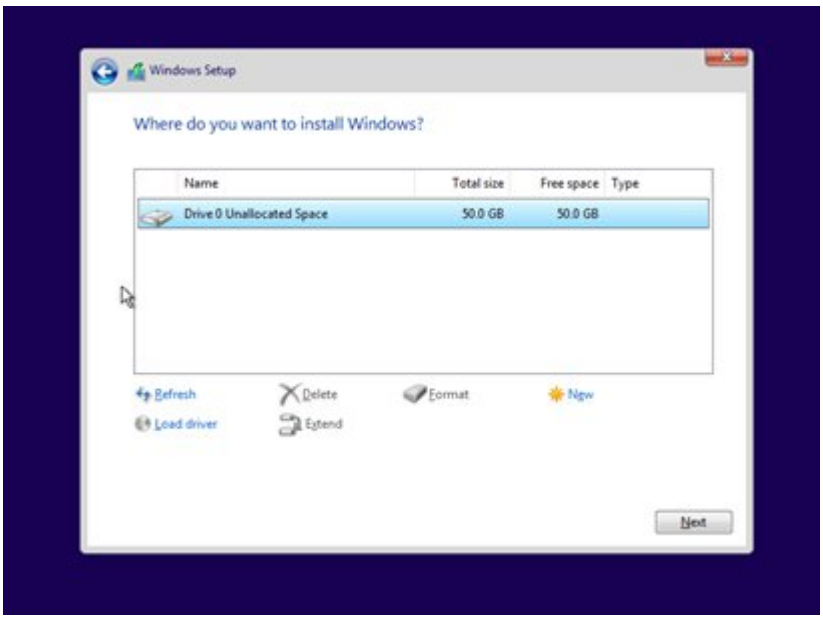
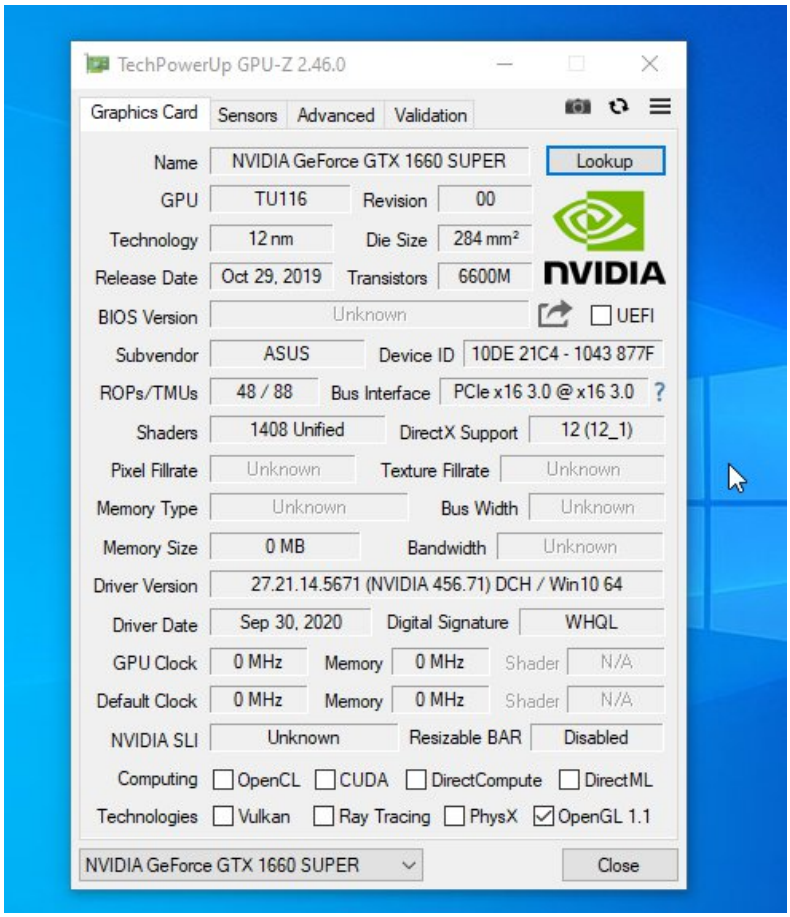


Figure 68 Windows 10 initial boot.

In Figure 69 the correct GPU detection using GPU-Z is shown. Such results proves the ability of KubeVirt support for GPUs. It is important to note that GPU-passthrough allocates the GPU hardware to a single VM. From an orchestration standpoint, this means each VM will be mapped to a single GPU. For sharing GPU resources across multiple VMs, it should be used the vGPU feature of KubeVirt instead[36].



[36] https://kubevirt.io/user-guide/virtual_machines/host-devices/

Figure 69 TechPowerUp GPU-Z detecting the GPU.

5.2 Migrating from on-premise to on-cloud

For Collins Aerospace, the vision of CHARITY to enable highly efficient network slices spanning the domains of Cloud providers, Edge infrastructure and local resources inspired a radical re-imagining of what could be achieved in terms of real-time, interactive XR streaming on the cloud. The traditional approach to flight simulators has been to deploy sufficient compute and storage resources alongside 2D fixed screens to deliver on the stringent quality demands of a certification-grade simulator. Scaling up or down is essentially constrained to vertical scaling in which we use more powerful or less powerful hardware as the deployment dictates. In Figure 70 below, we depict three sample deployment configurations

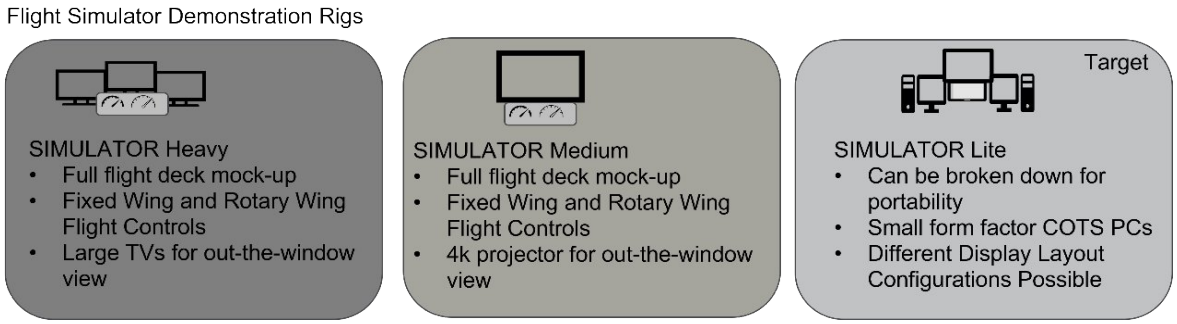


Figure 70: Some deployment models for the existing flight simulator.

As presented in Figure 71, the traditional approach is somewhat monolithic in terms of deployment flexibility. Multiple flight simulators co-located on the same site have no interaction or resource sharing and each operates independently on its own dedicated hardware.

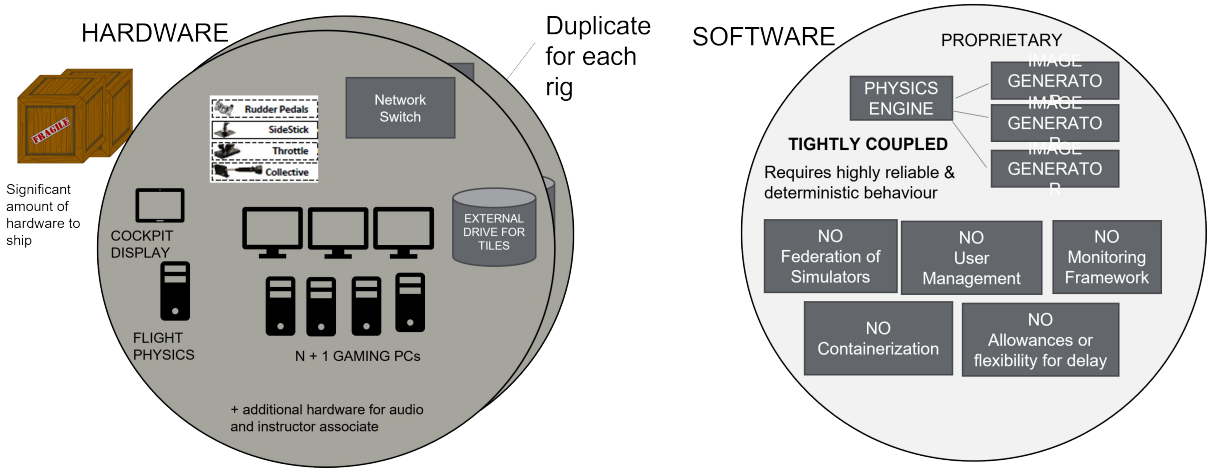


Figure 71: Existing deployment options revolve around a monolithic approach.

The current deployment model presents a variety of challenges as outlined below in Table 19 .

Table 19: Challenges presented by the traditional deployment model.

Challenges	
Each user requires their own full rig – dimension site hardware up front for max number of simultaneous users	MS Windows focused

No sharing of resources between rigs	Strict latency demands
Software updates are problematic – especially tiles database which is very large	Specialized scenery generator coupled with flight dynamics
Hardware updates are problematic	Difficult to scale
Sense of immersion with low-end rig is poor	Licensing complicates experimentation on third party edge/cloud
No centralized monitoring (across users)	

At the outset, these considerations drove our decision to rethink the flight simulator architecture, to work in a distributed manner with the ability to leverage the CHARITY platform. We envisaged clear benefits that a redesign should bring as outlined below in Table 20.

Table 20: Target benefits from redesign.

Benefits	
Greatly reduced local hardware footprint	User & session management, simulator federation
Edge and cloud resources shared between simulators	Monitoring framework integration
Tiles database and rendering engines can be updated on the cloud	Improved versatility through Microservices with Docker containers
Hardware upgrades simplified	Caching with lookahead rendering to manage delays
Improved sense of immersion	Pluggable scenery generator -> flightgear
Improved Scalability	Headless remote rendering for remote computation and local display
Pluggable upscaling	Customizable latency compensation tactics available

5.2.1 The Latency Challenge

Operating a commercial Flight Simulator requires speed and consistency. The turnaround budgets are tight. In deploying to the cloud, we take an already demanding problem that is currently addressed using dedicated local hardware and network resources and exacerbate it by distributing resources across large distances as summarized below in Figure 72.

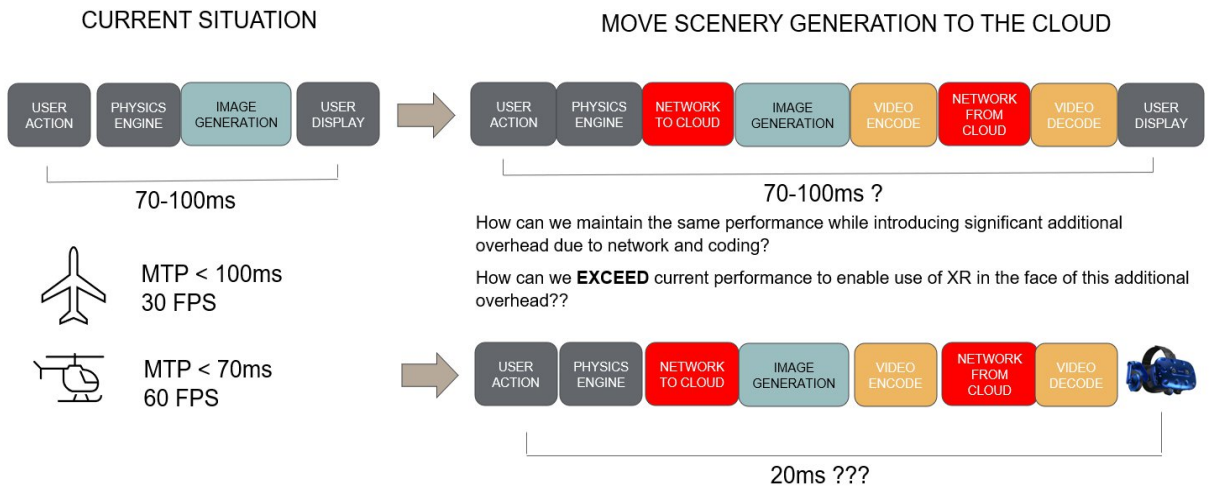


Figure 72: Motion To Photon budgets become even more demanding with XR and the cloud.

In 2018, Collins conducted internal experiments assessing the viability of cloud hosted flight simulation [23]. The findings revealed significant challenges that needed to be overcome with respect to network latency and jitter:

- Network Latency is a significant obstacle "*Transport delays vary widely based on network topology, provider, virtual private network, user-to-cloud distance, and other factors*".
- Sporadic variations in rendering times can result in stalls "*cloud-based computing model will require stringent provisioning of shared resources to provide the kind of performance and determinism guarantees users expect*".

The experiments were predicated on the display of scenery on two dimensional monitors – not XR headsets which have far more demanding latency budgets. It was clear from the early stages of the CHARITY project that we were facing significant challenges that could be alleviated but not solved entirely by the CHARITY platform alone. The physics of distance needed to be tackled.

5.2.2 Tackling XR Latency

A key observation about latency budgets in XR is that there are different types – rotational and translational as shown below in Figure 73. The charts on the right portray how latency demands are dependent on the nature of the user activity [25] and we superimposed the position that scenery generation for a flight simulator would occupy.

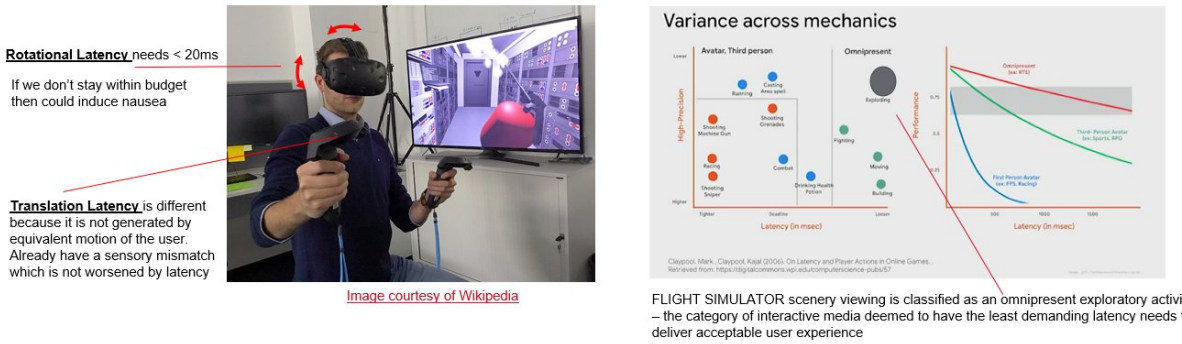


Figure 73: The latency budget available depends on the activity.

Updates caused by the user rotating their head need to be very fast (< 20ms) to prevent nausea for a significant proportion of the population. However, in [24] the authors note that translation motion delays of 100-200ms are "non-trivial to notice". For the flight simulator scenario, we have a user that sits within a virtual cabin and is able to look out the window at synthetically generated scenery. If the user turns their head then the local view inside the cabin needs to update quickly. The outside view only changes with the movement of the simulated aircraft itself (which alters course slowly in response to user actions). We propose to leverage this dichotomy to move the generation of synthetic scenery seen through the cabin windows to the cloud while keeping the rendering of the cabin itself local.

5.2.2.1 Prediction to extend the latency budget

If we detach the world outside a simulated aircraft cabin from the world inside then an additional opportunity presents itself to further extend our latency budget. As pointed out previously, the out-the-window view updates in accordance with movement of the aircraft. Aircraft possess nothing like the rapid freedom of movement of a human pilot. Its position within the seconds ahead should be predictable with a high degree of accuracy. This presents the opportunity to render what we need ahead of time on the cloud and cache it locally to enable what Google have referred to as Negative Latency [27] – a variation of which they employed in the Google Stadia platform.

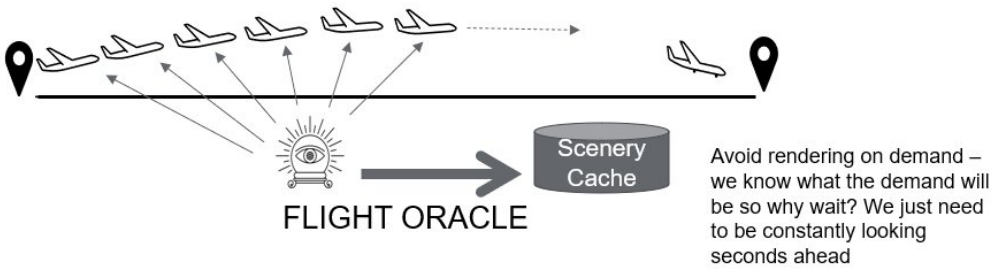


Figure 74: Movement of an aircraft can be predicted to enable pre-rendering of scenes.

By caching at the edge, our goal is to detach the cloud from the stringent motion-to-photon loop to reduce the latency and jitter that would otherwise be experienced with cloud rendering in the real-time chain.

5.2.2.1.1 Experiments

We employed an LTSM (Long Short-Term Memory) approach for trajectory prediction.

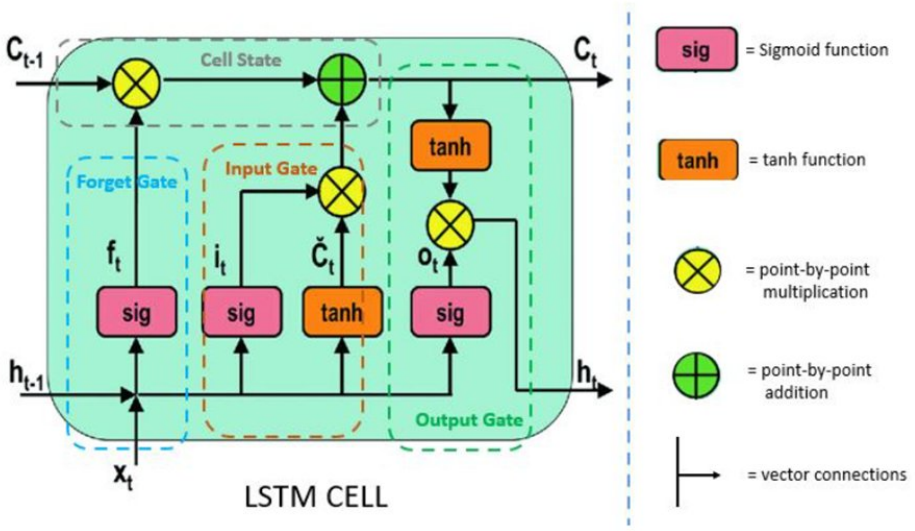


Figure 75: Long Term Short-Term Memory model of operation.

LSTM processes sequential data by controlling the flow of information through a combination of gates (input, forget, and output gates) and a cell state:

Input Gate:

- Significance: The input gate controls how much of the current input should be incorporated into the cell state. It helps in determining what information from the current input is important and should be remembered.
- Operation: The input gate computes a sigmoid activation, which acts as a filter to gate the input and decide what information is relevant for the current time step. It determines how much to update the cell state with new information.

Forget Gate (f):

- Significance: The forget gate regulates what information from the previous cell state should be retained and what should be discarded. It helps the LSTM in forgetting unnecessary or outdated information.

- Operation: The forget gate computes a sigmoid activation, which decides what portion of the previous cell state should be retained. It controls what information should be carried forward from the past and what should be discarded.

Output Gate (o):

- Significance: The output gate controls how much of the current cell state should be exposed as the hidden state for the current time step. It influences what the LSTM cell will output as a prediction or what information will be passed to the next LSTM cell or the output layer.
- Operation: The output gate computes a sigmoid activation and a tanh activation. The sigmoid activation determines how much of the cell state should be exposed, and the tanh activation scales the cell state to produce the new hidden state. The new hidden state captures the relevant information from the cell state for the current time step.

We trained a model using a small number of recorded flight trajectories and tested with an unseen trajectory. The position of an aircraft is captured by a set of values for latitude, longitude, heading, altitude, pitch and roll. Of these figures, we would expect a fast moving commercial aircraft to experience most change on the geographical coordinates – latitude and longitude – and this has been borne out with our predictions which demonstrate prediction errors on these vectors. Our results can be viewed below in Figure 76.

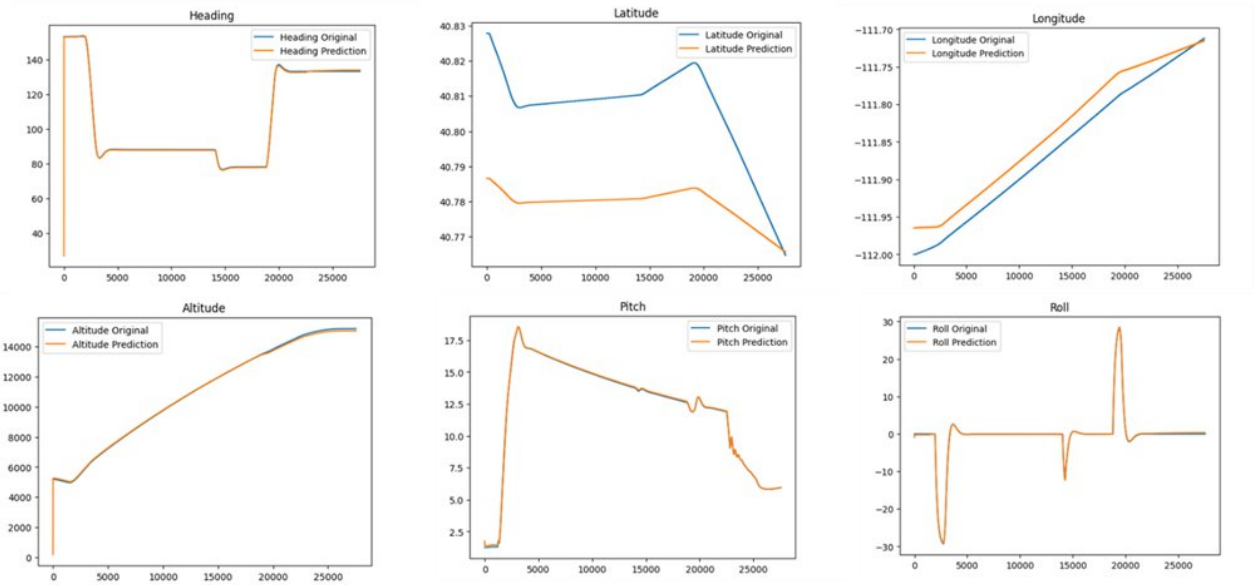


Figure 76: Predicted trajectory versus observed trajectory.

Our goal within the scope of CHARITY was to demonstrate the feasibility of the concept and we feel this has been accomplished. We believe future refinement that considers additional sensor data such as aircraft velocity and windspeed offer significant room for improvements.

5.2.2.2 The Frame Rate Challenge

As witnessed by the steadily increasing refresh rates of XR headwear, high frame rates are seen as an essential component of an acceptable XR user experience. Regardless of what is deemed to be an acceptable rate of frames per second – 30, 60, 90, 120 – we assume that the originator of the media stream must generate that rate. If we want to attain 90FPS with flight scenery, then must we render 90FPS in the cloud and ship back to the nearest cache? As with modern televisions, frame interpolation has become standard functionality in XR headsets. The manufacturers of such headset want to avoid inconsistent or below-par frame rates emanating from media sources to result in compromised experience for the user who may attribute blame to the headset itself. XR headsets need a consistent frame rate. If they do not get it, then they use predictions cached locally on the headset to backfill any

missing frames. We observe functionality termed Asynchronous Timewarp and Spacewarp [26] in the Oculus headsets and Motion Smoothing in SteamVR headsets.

Instead of the XR experience imposing more stringent quality demands than conventional 2D monitors, we propose to explore using the stabilization technology built into XR headsets to our advantage. It gives us the option of generating a lower frame rate on the cloud when resourcing pressures preclude us from either rendering the required frame rate due to computational resource stresses or from delivering the required frame rate to the edge due to bandwidth stresses.

5.2.2.2.1 Experiments

We first set out to investigate doing frame rate upscaling on the Edge. This offers a means independent of headset choice to ease experimentation while additionally enabling us to investigate the latest developments in frame interpolation that may not have made their way into commercial headsets.

Frame interpolation algorithms estimate the content of intermediate frames that would fill the gaps between existing frames. These intermediate frames are generated by interpolating pixel values between the original frames. The key is to create new frames that smooth out the motion between the existing frames. To create the intermediate frames, pixel values are blended or interpolated between two consecutive frames based on their motion.

During our investigation we experimented with several approaches:

- **Bicubic Interpolation:** Produces smooth results. It considers a 4x4 neighbourhood of pixels around the non-integer coordinate and uses cubic polynomials to estimate the interpolated value.
- **Lucas-Kanade Optical Flow:** estimates the motion or displacement of image features between two consecutive frames in a video sequence. Lucas-Kanade optical flow estimates the motion vectors (displacements) of image features between two frames. These motion vectors can be used to understand how objects or points in the scene move from one frame to the next.
  Once the motion vectors are obtained, they can be used to guide the generation of intermediate frames. Given two consecutive frames and the estimated motion vectors, we can interpolate pixel values between these frames to create intermediate frames. The interpolation process involves warping and blending pixel values based on the estimated motion.
- **RIFE (Real-Time Intermediate Flow Estimation):** This is a deep learning image interpolation technique which derives intermediate frames using Convolution Neural Networks (CNNs). RIFE estimates bidirectional optical flow fields between the input frames. By estimating optical flow in both directions, RIFE can generate more accurate and visually pleasing intermediate frames. Furthermore, RIFE aims to ensure temporal consistency between the interpolated frames and the original frames, resulting in smooth motion and reduced artifacts. As the name suggests, RIFE is designed for real-time applications, making it suitable for video frame interpolation in scenarios like video playback and video editing.

RIFE was the approach we adopted. It operates in real-time (less than 0.5 seconds to upscale 10 frames to 40 frames) across a range of resolutions and consumes acceptable GPU resources in testing so far. We are still in the process of quantifying the precise performance under different conditions.

The Pixel Resolution Challenge

As consumer XR headsets evolve to target 4K or 8K resolutions, there is a growing imperative on the part of media stream producers to render higher and higher resolution imagery. This has significant impact for bandwidth as 4k resolution requires an order of magnitude more bandwidth than High Definition (approx. 15Mbps versus 1.5Mbps). As with frame rate, we assume that the originator of the media stream must generate the required resolution. Modern TVs and games consoles need to deliver a high-resolution viewing experience even when the source of frames is of low resolution. To accomplish this, they employ upscaling algorithms to ‘fill out’ the missing pixels. We propose to integrate a resolution upscaling component into our streaming pipeline to cater for scenarios in which

we cannot render high resolution imagery on the cloud for reasons of resource availability (compute or network bandwidth). Lower resolution frames will be received at the Edge and upscaled as appropriate.

5.2.2.2.2 Investigation & Experiments

Traditional resolution upscaling entails copying and repeating pixels from a lower resolution image to produce a higher resolution version. Filtering is applied to smooth the image and round out unwanted jagged edges that may become visible due to the stretching. The result is an image that could fit on a larger display but can often appear muted or blurry.
AI upscaling takes a different approach. Given a low-resolution image, an AI model predicts a high-resolution image that would downscale to look like the original, low-resolution image. To predict the upscaled images with high accuracy, a neural network model must be trained on large numbers of images. The deployed AI model can then take low-resolution video and produce impressive sharpness and enhanced details beyond the capabilities of traditional upscaling – the edges look sharper; hair looks more authentic and detail in general is crisper.
When evaluating upscaling approaches in CHARITY, we considered both traditional and AI approaches. Our assessment of suitability was primarily based on speed and quality[37]. To objectively assess the quality of one approach over another we used VMAF which we will now briefly introduce before continuing to the various upscaling techniques we investigated.

VMAF (Video Multimethod Assessment Fusion).

VMAF is a perceptual video quality assessment algorithm developed by Netflix. It is designed to estimate the quality of videos as perceived by human viewers. It has been widely adopted in the video streaming and broadcasting industry as a standard for measuring video quality, especially for contents distributed over the internet.
VMAF uses a machine learning approach to train its models. It leverages a dataset of videos with known quality scores, which have been rated by human subjects through subjective testing. The models are trained to learn the relationship between the extracted features and the human-rated quality scores. The essential pipeline is shown below in Figure 77.

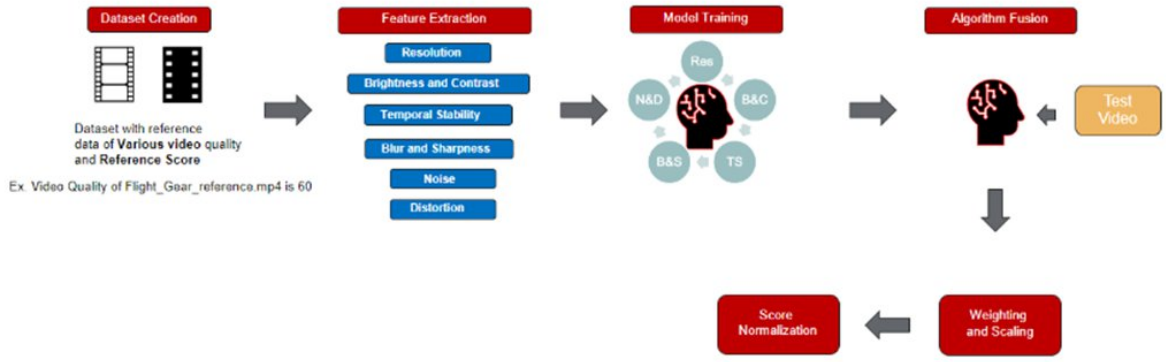


Figure 77: The VMAF Pipeline.

VMAF starts by extracting a variety of features from the reference video and the upscaled video. These features can include spatial and temporal information, luminance, contrast, and other visual characteristics. After extracting features and applying the perceptual quality models, VMAF combines the individual quality scores from these models using a fusion algorithm. This fusion step considers the strengths and weaknesses of each model and produces a final quality score that is more robust and reliable. The final VMAF score is a single value that represents the perceived quality of the upscaled video compared to the reference video.

[37] Secondary factors such as GPU memory usage and ease of integration were also considered.

Approaches Evaluated.

Following a review of the state of the art, we narrowed our considerations to the approaches below. This was based on the availability of open source code that could be readily tested and adapted to our needs along with compelling published results in terms of speed and quality.

Bicubic Interpolation

This is a mathematical technique used to estimate the colour values of new pixels in an upscaled image. It considers a larger neighbourhood of pixels surrounding the target pixel than bilinear interpolation and has higher accuracy as a result. For each pixel in the target (upscaled) image, it considers a 4x4 grid of neighbouring pixels from the original image. It uses a weighted averaging technique to estimate the colour value of the target pixel. These weights are derived from cubic functions and are used to blend the colours of the neighbouring pixels.

Execution time is very fast at 0.007 seconds per frame[38] which comfortably delivers real time upscaling. The difficulty with this approach is that it essentially does not do any reasoning or add any new data to the image so this technique produced quite blurred images.

EDSR (Enhanced Deep Super Resolution)

EDSR [42] is a deep neural network architecture that uses convolutional neural networks (CNNs) to perform image super-resolution. It has been designed to produce high-quality upscaled images with a focus on accuracy and detail preservation. It takes a low-resolution image as input, which is typically a down sampled version of a higher-resolution image. This low-resolution image is passed through the network to generate an upscaled image.

EDSR typically consists of a deep stack of convolutional layers. These layers are responsible for learning the complex features and representations from the input image. The core of the EDSR architecture is the residual blocks. Residual learning is a key component of EDSR, which involves skipping connections (shortcuts) that allow gradients to flow directly through the network. This helps in training very deep networks efficiently. Each residual block extracts and enhances features from the input image. These features capture important information about the image, including edges, textures, and other visual elements.

ESPCN (Efficient Sub-Pixel Convolutional Neural Network)

ESPCN [43] is a compact convolutional neural network architecture designed for real-time image upscaling and super-resolution. It focuses on efficiently increasing the resolution of images while minimizing computational requirements. The key innovation in ESPCN is the use of sub-pixel convolution layers to upscale low-resolution feature maps into high-resolution images.

The network starts with a series of convolutional layers that extract features from the low-resolution input image. These layers learn to capture essential information about edges, textures, and other image features. The distinctive feature of ESPCN is its use of sub-pixel convolution layers. These layers are responsible for the upscaling process. Instead of using traditional upscaling techniques, sub-pixel convolution layers learn to transform low-resolution feature maps directly into high-resolution images.

The sub-pixel convolution layers rearrange the feature maps spatially, effectively increasing the resolution. Each feature map is divided into non-overlapping sub-pixels and placed next to each other to form the high-resolution image.

[38] Using an Intel i9 processor.

FSRCNN (Fast Super-Resolution Convolutional Neural Network)

FSRCNN [[41]] was introduced as an efficient and faster alternative to traditional methods for single-image super-resolution. It focuses on providing high-quality results while minimizing computational costs, and positions itself as suitable for real-time applications such as video upscaling and image enhancement on resource-constrained devices. It takes a low-resolution image as its input which has typically been down-sampled from a higher-resolution image.

The network starts with a series of convolutional layers that extract relevant features from the low-resolution input image. These convolutional layers help capture important information about edges, textures, and other image features. It typically incorporates a set of convolutional layers that reduce the dimensionality of feature maps to simplify processing, followed by another set of layers that increase the dimensionality. These layers help reshape the features for the super-resolution process.

A key innovation in FSRCNN Is its use of learnable filters and convolutional layers to increase the resolution of feature maps. These filters are designed to upscale the image and are applied in a learnable manner. This is different from traditional methods like interpolation or deconvolution.

LAPSRN (Laplacian Pyramid Super-Resolution Network)

Like FSRCNN, LAPSRN [44] is also based on deep learning but employs a different architecture and approach. The difficulty with this approach is that it requires training on domain imagery. While it exhibits impressive performance, its quality was poor due to the lack of training we carried out. We were keen to seek an approach that was more reusable and avoid this type of customization that would only deliver good results for our own application.

SRGAN (Super-Resolution Generative Adversarial Network)

SRGAN [45] is a deep learning architecture that is used for image super-resolution. It is known for its ability to generate highly detailed and realistic high-resolution images from low-resolution inputs. SRGAN is based on the principles of generative adversarial networks (GANs) and is specifically designed for the task of single-image super-resolution.

Generator (G): The generator network in SRGAN is responsible for taking a low-resolution image as input and producing a high-resolution image as output. It achieves this through a series of convolutional layers, activation functions, and other operations. The generator aims to learn the mapping from low to high resolution.

Discriminator (D): The discriminator network in SRGAN is used to evaluate the realism of the generated high-resolution images. It attempts to distinguish between real high-resolution images and generated high-resolution images. The discriminator is also a neural network that operates on images.

Generative Adversarial Network (GAN): SRGAN uses the GAN framework, which consists of a generator and a discriminator. The generator tries to generate images that are indistinguishable from real high-resolution images, while the discriminator tries to get better at distinguishing between real and generated images. The two networks are trained in an adversarial manner, where the generator aims to fool the discriminator into accepting its generated images as real.

ESRGAN (Enhanced Super-Resolution Generative Adversarial Network)

ESRGAN [46] is an extension and improvement upon SRGAN. It is more complex than SRGAN and builds upon the SRGAN architecture by introducing additional enhancements and refinements. It is typically trained on more diverse and larger datasets, which can lead to improved generalization and the ability to handle a wider range of image types. ESRGAN generally uses deeper networks compared to SRGAN, allowing it to capture more complex image features and textures.

ESRGAN is generally considered more advanced and capable of generating even more realistic and detailed high-resolution images from low-resolution inputs.

Experiments.

Below in Table 21 we see the results we gathered while evaluating different approaches. In cases where the performance or quality was too poor, we discontinued and advanced onto the next alternative.

Table 21: Performance of upscaling techniques investigated

Hardware	Method	Execution Time (per frame)	Input Image	Upscale Resolution	FPS	VAMF Score	CPU Data Transfer
NVIDIA GeForce RTX 3090	FSRCNN	0.01 sec	640*480	2560*1920		22	
	EDSR	2.27 sec	640*480	2560*1920		12	
	LapSRN	0.007 sec	640*480	2560*1920		20	
	ESPCN	0.93 sec	640*480	2560*1920		24	
	SRGAN	0.33 sec	640*480	2560*1920	3		
	ESRGAN	0.05 sec	320*240	1280*960	20	70	200,704 bytes
		0.09 sec	480*360	1920*1440	10	85	488,621 bytes
		0.15 sec	640*480	2560*1920	6	89	757,760 bytes
NVIDIA RTX A4000	ESRGAN	0.11 sec	320*240	1280*960	10	70	200,704 bytes
		0.23 sec	480*360	1920*1440	4	85	488,621 bytes
		0.31 sec	640*480	2560*1920	3	89	757,760 bytes

The hidden cost of GPU to Host transfer

The ESRGAN approach was the most suitable we found in terms of speed, quality and reusability. However, we discovered an unexpected bottleneck when integrating the approach into our overall pipeline. We were getting blistering speed per frame on the GPU (in the region of 0.004 seconds per 640x480px frame) but there was an enormous overhead in the transfer of data from GPU to CPU to save the upscaled frame. We tried a wide range of tactics to reduce this cost. A significant challenge is the GPU memory consumption of the ESRGAN approach. We were observing consumption exceeding 20GB which made the approach untenable. With tuning, we found we could pin the memory consumption to approximately 9GB which was still very high but workable. Upon experimenting with submitting batches of frames for upscaling however, we quickly exhausted available memory so had to abandon this approach. We tried using queues and multiprocessing on the CPU and again ran out of memory. We tried using GPU arrays (cupy[39]) but this did not prove fruitful. We tried compression on the GPU before transferring to the CPU but the three-dimensional tensor output from the ESRGAN approach was not amenable to this. The only approach was to reduce the amount of data we needed to transfer from the GPU and this entailed reducing the resolution we could achieve with upscaling. Currently, the approach is only feasible for upscaling from input images of 320x240 to 1280x960px.

Ongoing work and future prospects

In seeking to deploy an application independent approach to upscaling, we limited our range of options somewhat. If we opted for an application-specific approach which was trained specifically on our

[39] https://cupy.dev.

application imagery, then we would expect better results (particularly from the LapSRN algorithm[40] and the FRSCNN approach).

One avenue we seek to explore is a two-phase upscaling approach. We would initially upscale using ESRGAN on the GPU and then upscale again using bicubic on the CPU. This would release pressure on the GPU and enable us to multithread the bicubic upscaling on the CPU. The initial upscaling effort would add detail and sharpness that would be absent from a pure bicubic approach.

We remain cognizant of the fact that our approach of upscaling imagery is limited by our inability to intrude on the source rendering pipeline in the open-source image generator we employed. We have no access to depth buffers for example that would otherwise have allowed us to experiment with approaches such as NVIDIA DLSS[41]. From the outset, in the spirit of the CHARITY project, we sought to push through with an approach that would be reusable for any third-party graphical application and shied away from application-specific solutions. While hardware advances continue and will undoubtedly deliver improved performance, we may have reached the limits of what is currently feasible with the time and resources available.

5.2.3 Towards Cloud Native

We began our journey with a monolithic platform that was not amenable to distributed deployment and execution. We proceeded to redesign the platform and move towards a cloud native architecture. As can be seen below in Figure 78, we decomposed the platform into self-contained microservices.

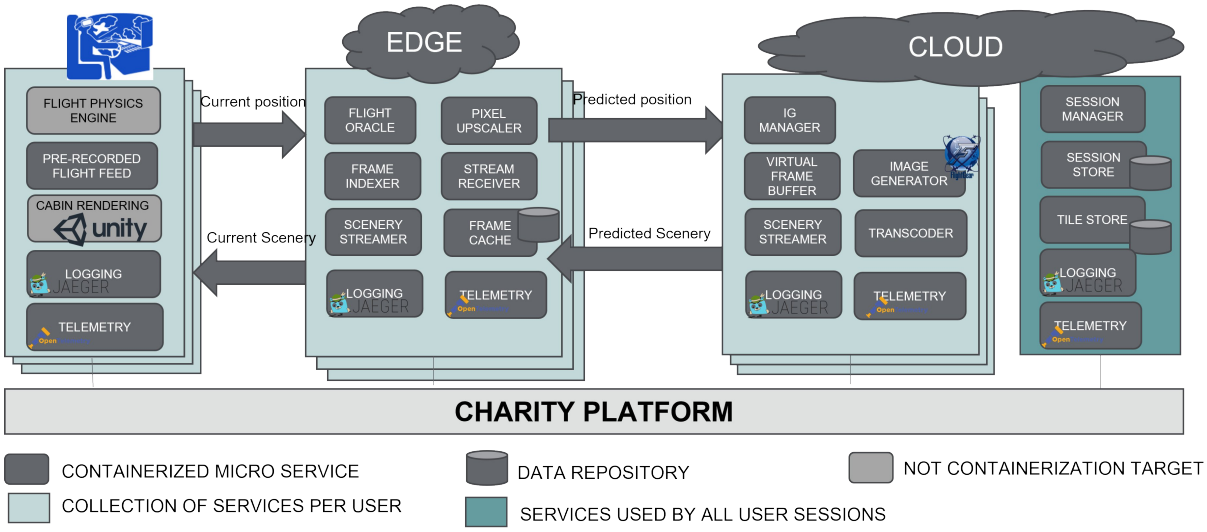


Figure 78: Flight Simulator redesigned as cloud native.

The new architecture better reflects modern application design and gives us the opportunity to leverage core features of the CHARITY platform that would have been difficult and far more restricted with the original design such as the CHARITY service mesh for application adaptation, monitoring and alerting, dynamic deployment and orchestration. Crucially, it brings options and mechanisms to explore distributed deployment across the edge and cloud.

5.3 Dissection of the Unity3D Physics engine

ORAMA's commercial gamified multi-user VR medical training platform (UC2-1) is built using the

[40] This approach produces two-dimensional output that can be compressed and produce far superior speeds as the GPU to host transfer issue is greatly reduced. Without imagery-specific training however, the quality is poor.

[41] https://www.nvidia.com/en-gb/geforce/technologies/dlss/.

MAGES SDK on top of the Unity3D game engine. Exploiting Unity3D's network layer, the MAGES SDK handles and synchronises in-game interactions, deformable object transformations and physics simulation by broadcasting transformation values over the network. Under the hood, as part of the MAGES SDK, the custom Geometric Algebra interpolation engine is utilised for efficient network transmission and local interpolation of in-between positions/rotations for each end-device (HMD). The architectural design of ORAMA's training applications involves a single application component, installed and run on untethered HMDs, that employ local processes for storage, rendering, and physics deformations.

An experimental architecture, based on MAGES SDK, allows the transition to an Edge-Cloud application, upscaling to collaborative cloud VR training applications specially formulated for untethered HMDs. The goal of this R&D version of ORAMA's training application is to optimize the status of the cooperative mode in terms of lower latency, higher performance on average network conditions, and, ultimately, higher number of CCUs. This new approach, realized through computation offloading of the entire ORAMA's application in edge-cloud resources, requires interactions and data exchanges between the different modules placed on device, and services on Edge-Cloud.

ORAMA is currently designing and developing the required technologies and solutions to support its advanced media applications by exploiting methods and techniques for the dissection of the Unity physics simulation engine as a separate VM microservice that will run on the Edge-Cloud. Methods and techniques regarding multi-threaded rendering and physics in Unity are also being investigated.

5.3.1 Dissection of Physics Simulation Engine

Currently, a typical Unity3D game engine pipeline involves simultaneous execution of CPU physics-related calculations along with GPU calculations related to the rendering of the scene.

In this section, we provide an overview of how a dissection of the physics and the scene-rendering pipeline can be achieved. Although a distributed application architecture usually decreases running times, an unoptimized dissection may lead to increased latency, since there are numerous inter-calls between the physics engine and the renderer. In the case of a desktop-VR local network system setup, the dissection is feasible and almost straightforward. However, in the case of a mobile-VR edge-cloud setup the physics engine dissection is rather challenging. ORAMA investigated methods and techniques to support the dissection and allow the physics simulation engine to be run as a separate edge-cloud, as a containerized microservice.

5.3.2 Methodology – Notation

The dissected Unity3D pipeline involves two, bidirectionally communicating, components:

- The Graphics Client (Graphics rendering), and
- The Physics Server (Physics simulations & Game Logic).

The Graphics Client includes the entire Unity3D pipeline, along with its own, local, physics engine, which remains mostly inactive, only used for the initial connection to the Physics Server. Main goal of the dissected Unity3D pipeline is to allow any Game Object on the scene to be fully simulated by the dissected Physics Server and not by the Graphics Client's local physics engine.
For the reader's convenience, we define below some terms used throughout the dissection overview.

- **Graphics Object:** A Game Object component, with no physics-related scripts and data residing within the Host.
- **Physics Object:** A Game Object component, responsible for storing all physics parameters. It has attached a Rigid Body script, a Collider script, or a combination of the two.
- **Remote Game Object:** A Fully Dissected Game Object, that exists on both the Graphics Client and Physics Server. This is not a tangible component or Object, simply a term that encompasses both Graphics and Physics Objects.

5.3.3 Methodology - Overview

5.3.3.1 Communication of the two Components

The main two components, i.e., the Graphics Client and the Physics Server, communicate using Photon Unity Networking, a solution for providing Multiplayer support in simulations and Games made with Unity. All Remote Game Objects are known at build time for both the Physics and Graphics Server, making it possible to synchronize across Host and Physics Server via Photon.

5.3.3.2 Splitting a Game Object into a Graphics and a Physics Object

All Graphics Objects have no Physics Related Components attached to them (Colliders, Rigidbodies) and the Physics Server has no Rendering camera (thus, no Graphics processing takes place).

During gameplay, the transformations of the Graphics Client's Graphics Object are synchronized with the respective Physics Object present in the Physics Server. The Game Object's transformations can either be controlled entirely by the Physics Server, or by the Graphics Client. In the latter case, the Physics simulation continues, and the controlled Physics Object interacts with the rest of the Physics Objects as expected.

5.3.4 Implementation

5.3.4.1 Initial Setup

With the successful initiation of the Physics Server, the Physics Client creates a session and waits for users to join. After the Host's successful initialization, the user chooses which session they would like to join, and use the User Interface to join.

5.3.4.2 Game Object creation after Initialization

The Game Logic resides in both the Physics Server and Graphics Client, so any Game Object that exists in all Graphics Clients also exists in the Physics Server.

5.3.4.3 Simulation and Gameplay

Depending on the developer's choice, the transformations of a Game Object can be either controlled by the Graphics Client or the Physics Server. In both cases, the Physics simulation is always running. When a Graphics Object is translated by the Host, the corresponding Physics Object is also translated using Physics calculations and not direct transformation changes so that the simulation is accurate and ensuring that no undesirable object clipping occurs. All transformations and synchronization is handled by Photon Networking, which sends only the necessary data to ensure minimal network usage with optimal QoE.

5.3.5 Lab Testing

We conducted Lab Tests using simulated users to assess network usage and Quality of Experience on a Local network set up.

The experimentation was conducted as follows:

- 1 Physics Server running in the Unity Editor
- 12 Graphics Builds running at the same time.
    - 10 Bot Builds (User Input is simulated via software)

- 2 Player Builds (Real Players)

- Photon Relay Server located in the Photon Cloud, accessed via Internet.

Table 22: Average network usage metrics.

Per Client	**Upload**	**Download**
Idling Player	1.9 KB/s	13.4 KB/s
1 Player moving Around	3 KB/s	14 KB/s
Physics Server	**Upload**	**Download**
Idling Player	400 B/s	13.3 KB/s
1 Player moving Around	450 B/s	15.2 KB/s

The very low network usage observed is due to the usage of a Dual Quaternion Interpolator used in the MAGES SDK Photon Implementation. This allows the objects' transformation to be updated less frequently over the network with the same QoE as if they were updated every frame.

5.3.6 QoE Subjective remarks

During gameplay, whenever the network quality falls below a threshold, there were two issues that were especially noticeable. First, when directly interacting with objects in VR, the network latency causes the objects to feel "squishy", since the user's hand that pushes the object would initially penetrate inside the object and after some milliseconds the object would react to the push and move away. Secondly, when there is a high amount of packet loss, some objects tend to "flicker" between two positions. This issue is not very common as it is not experienced every time network conditions deteriorate. The first issue, however, is rather common, but not distracting from the gameplay in a severe way.

5.3.7 Conclusions - Future Work

The work in this section has shown that the dissection of the Unity3D pipeline is feasible, yet dependent on the network characteristics between the Host and the Physics server. The conducted tests helped the derivation of the network latency and packet loss thresholds, below which we can achieve a pleasant QoE to the VR medical training application. These thresholds should not be exceeded by the provided testbed network.

Although docker containers outperform VMs in the case of space and processing overhead, they are rather immature in graphics acceleration processes. In this case, the use of VMs is far more advantageous since they have highly optimized graphics drivers and kvm passthrough support. Additionally, docker containers have limited graphics drivers support, since only experimental versions (for all vendors) for Linux are currently available. As such, we opt for VMs when graphic acceleration is required, while dockers are used when only CPU resources are utilised as it it the case of the Physics Engine in Lspart2.

.

5.4 Investigating Multi-threaded rendering in the Unity3D game engine

Multi-threading exploits a CPU's capability of processing many threads concurrently across many cores. A multi-threading program always starts in one main thread, which subsequently creates new threads that run in parallel. Upon completion, these threads usually synchronise their results with the main thread.

The generation of more concurrent threads than the available CPU cores, leads to a concurrent sharing of CPU resources among the threads, which causes frequent, resource-intensive, context switching. As such, the multi-threading approach always suits cases with a few long-life tasks. Game Engine pipelines mostly deal with many short-life unrelated tasks that execute at once. Multi-threading in such systems often results with a large number of short-life threads that challenge the CPU's and operating system's processing capacity, due to frequent creation and destruction of threads for short-lived tasks. The employment of a pool of threads, often mitigates this issue, increasing performance and avoiding latency in execution. However, even this solution does not always prevent a large number of concurrent active threads.

Multi-threaded programming faces high risks for race conditions which often produce significant challenges. A *race condition* occurs when the output of one task depends on the timing of another process outside of its control. This issue may be a source of crashes, deadlocks, incorrect output, and generally non-deterministic behaviour that produce non accurate rendering or simulations. As the cause of these problems depends on timing, the recreation of the issue could happen on rare occasions, making debugging a cumbersome process. Debugging tools, such as breakpoints and logging, often change the timing of individual threads, causing the problem to falsely disappear.

In the frame of taking advantage of the edge-cloud resources parallel processing, methods and techniques for parallel/multi-threading Rendering and Physics in Unity3D was explored. Unity3D supports multi-threaded math calculations and, in this regard, we will seek to exploit parallelization techniques for various sub-tasks, such as the skinning algorithms. Furthermore, Unity3D supports a limited form of multi-threaded rendering by utilising specific graphics API implementations or through the utilisation of Graphics Jobs System.

5.4.1 Single-threaded Rendering

Unity3D mainly features a single client occupying the main thread with the execution of the high-level rendering commands. The client also owns the real graphics device GfxDevice and performs the actual rendering through the underlying graphics API (GCMD) on the main thread.

5.4.2 Unity3D Multi-threading Built-in System

Multithreaded rendering in Unity, provided its graphics API permits it, is implemented as a single client, single worker thread. This works by taking advantage of the abstract GfxDevice interface in Unity3D. The different graphics API implementations, such as Vulkan, Metal and GLES, inherit from the GfxDevice.

When this system is enabled, rendering calculations are performed on a separate thread, called the *RenderThread*, while the rest of calculations are performed on the main game thread, namely the *MainThread*.

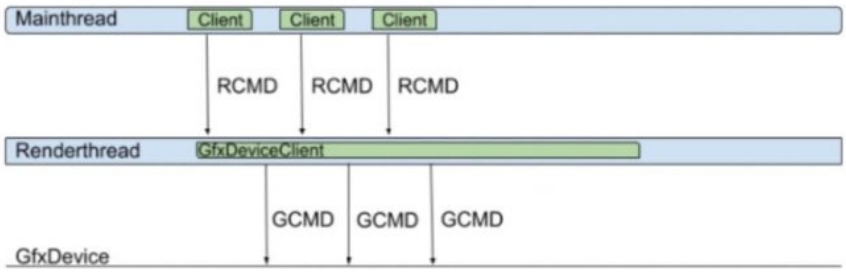


Figure 79: Unity3D multi-threading Built-in System.

5.4.3 Graphics Jobs System

The Unity3D *Jobs* system is not the traditional kind of multi-threading system as it manages multi-threaded code by creating jobs instead of threads. In that frame, a game is split into small units of work where each is responsible for one specific task. These units of work are called *jobs*. The Graphics Job system manages a group of worker threads across multiple cores. It usually has one worker thread per logical CPU core, to avoid context switching. Some cores may also be reserved for the operating system or other dedicated applications. As the job system enqueues the generated jobs in the job queue, the Worker threads take items from the job queue and execute them.

A job may receive parameters and operate on data in a similar way to a method call. As such they can be self-contained, or they can depend on other jobs to complete before they can run. Once scheduled, it cannot be interrupted. In complex systems, such as those required for game development, it is unlikely that a job is self-contained. All jobs are usually dependent on other jobs as they prepare data for them. The Graphics Job system supports dependencies across jobs, as it is responsible for managing them, ensuring job execution in the appropriate order. The Unity3D C# Job System is able to detect all race conditions, protecting the programmer from potential bugs.

Writing multithreaded code can provide high-performance benefits, such as significant gains in frame rate. Using the Burst compiler with C# jobs gives you improved quality, which also results in substantial reduction of battery consumption on mobile devices.

Graphics Job System integrates Unity's native job system. As such, User-written code and Unity3D engine code share the same Worker threads, avoiding the creation of more threads than CPU cores, which would cause contention for CPU resources.

Using the Job system, multiple native command generation threads take advantage of the graphics APIs that support recording graphics commands (GCMD) in a native format on multiple threads. It is implemented as multiple clients, no worker thread. This removes the performance impact of writing and reading commands in a custom format before submitting them to the API.

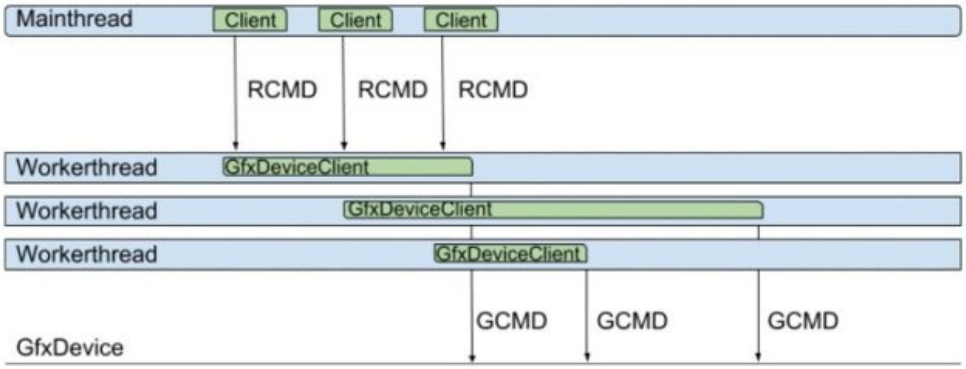


Figure 80: Graphics Jobs System

Note: Currently, Graphics Jobs do not have a RenderThread to schedule jobs, causing a small amount of overhead on the main thread for scheduling.

5.4.4 Vulkan Graphics API

By enabling Graphics Jobs and the use of the Vulkan graphics API for Windows on Unity3D, we tested the potential increase of performance of Unity3D for our VR offloaded solution. In most cases, the positive performance impact was minimal:

Table 23: Potential increase of performance of Unity3D using Vulkan graphics API for Windows

	Direct3D-11	Vulkan
Average Frame rate	45.47 fps	46.16 fps

As an additional remark, Vulkan on Unity3D proved to be more unstable than Direct3D11; in some cases, performance dropped significantly to ~30 fps when Vulcan was enabled.

5.4.5 Conclusions

In our research we noticed that, regarding multi-threaded rendering for 3D applications, one rendering thread is used, while many other work threads can parallelize other jobs such as physics, logic, AI, etc. To the best of our knowledge, there is no other multi-threaded rendering solution or any other alternative solution within Unity.

5.5 Adaptive rendering algorithms for low latency immersive applications

Virtual Reality (VR) applications have gained importance and interest over the last few years in various fields, like in manufacturing, training, entertainment, and so on. Moreover, modern wireless lightweight powerful Head Mounted Display (HMD), reach a high level of maturity and provides a more immersive experience. Despite these, a high level in quality of experience is still challenging when using standalone HMDs. as well as modern cloud/edge rendering pipelines considering the requirements for ultra-low latency (<20 ms) and high-bandwidth for a comfortable, satisfying, and convincing immersive experience [30].

Several solutions have been developed by the VR research community to achieve this goal. A lot of effort has been spent to reduce the computational burden related to the rendering. In fact, rendering for immersive devices requires at least the rendering from two different viewpoints, or more in some cases to generate a 360-degree panoramic image/video to stream accordingly to the position and orientation of the gaze of the user. The computation of the rendering can be alleviated in different ways. Some approaches exploit the fact that the best visual acuity is around the fovea, and exploit eye tracking to optimize the rendering, obtaining the so-called foveated rendering. Many other solutions exploit how the Human Visual Systems (HVS) works to reduce the quality of the rendering ensuring the same visual perceptual quality. For example, the work in [31], modifies the standard primitive rasterization considering some perceptual effects, allowing a more efficient rasterization pipeline for HMDs. Some other approaches take into account that distant objects do not require to be rendered with different disparities to be perceived correctly. For example, the work of [32], assumes that disparities are reduced for distant objects, and it uses a mix of stereoscopic and standard rendering to generate the images to be displayed. The experiments conducted demonstrate that this simple solution can give a satisfying experience in many cases. Other approaches work by super-sampling the temporal line, so they create/interpolate new frames in-between other ones to reduce the total number of images to generate. The state-of-the-art of this type is ExtraNet [33], a deep learning network capable to double the speed of the frame generation by extrapolating the new frame for the previous ones. The new frame is generated by minimizing the visual artefacts that typically happen in view-dependent parts of the images (e.g. specular reflection).

Recently, with the main goal of obtaining a high fidelity VR experience for standalone mobile devices, solutions that take advantage of computing the rendering at the edge are explored [34]. In this case,

the total end-to-end latency is dependent on the time to transmit sensor data from HMD to the edge computing node, the time for physics computations, the rendering/encoding time of the views on the edge node, the transmission time of the rendered images/video from the edge computing node to HMD, and the time to decode and display the view on the HMD. The encoding and decoding phases are optional and depend on the specific application. In this setting, different strategies can be used to optimize the rendering, caching, and streaming of the different views.

FlashBack [35], is a VR system which pre-renders all possible views on a 3D grid of suitable size and delivers frames according to the position and orientation of the viewers. Obviously, this is not optimal from a caching point of view. The work of [36], adopts a parallel rendering and streaming mechanism, that reuses rendering parts, that remain the same during the interaction, allowing a reduced streaming latency. Long-Short Term Memory (LSTM) ([37], [38]) model and Recurrent Neural Networks (RNN) ([39], [40]) are used to predict the head/body movements, that allow the optimization of the view generation, reducing the overall computational and improving performance.

In CHARITY, we investigated the integration of the above methods to some Use Cases, to obtain an adaptive rendering solution, to support high-quality low-latency VR applications. In that respect, we studied the above methods in relation with the CHARITY use case applications that utilize a remote computation pipeline for VR: UC2-1 VR Medical Training Simulator (ORAMA) and UC3-2 Manned-Unmanned Operations Trainer VR Simulation (CAI), and concluded that UC3-2 architecture is more suitable to integrate frame extrapolation/interpolation and super-resolution based methods. These algorithms have been tested extensively in the context of the UC3-2, as reported in Section 5.2.

5.6 Point Cloud Encoding / Decoding

5.6.1 UC1-3 Holographic Assistant

The CHARITY UC1-3 Holographic Assistant (Figure 81) adopts the physical principles "diffraction and interference of light" to enable real 3D holography, based on sophisticated custom optical components and algorithms. This lays the foundation for showing a butler-like avatar in 3D space on a holographic 3D display with true depth and true eye focus - for your eyes it is like natural viewing. The butler shall react to natural language and assists by providing information gathered from the cloud or the internet. Beside the 3D holographic presentation, this use case enables a lot of challenging services and new technology to be developed and implemented in the CHARITY cloud.

The use case is focusing on a cloud-based application rendering a virtual holographic 3D assistant including additional information and transferring / streaming the content to a local client system in a format compatible to interference-based holography. On the client system, the content is computed into a real-time 3D hologram and is presented on a holographic 3D display from SeeReal Technologies (SRT). By using eye-tracking, the observer always sees the correct perspective of the holographic assistant 3D scene. The hologram enables natural viewing for correct eye focusing and convergence to experience true depth and natural viewing. So, the well-known accommodation convergence conflict known from classic 3D stereo does not apply here.

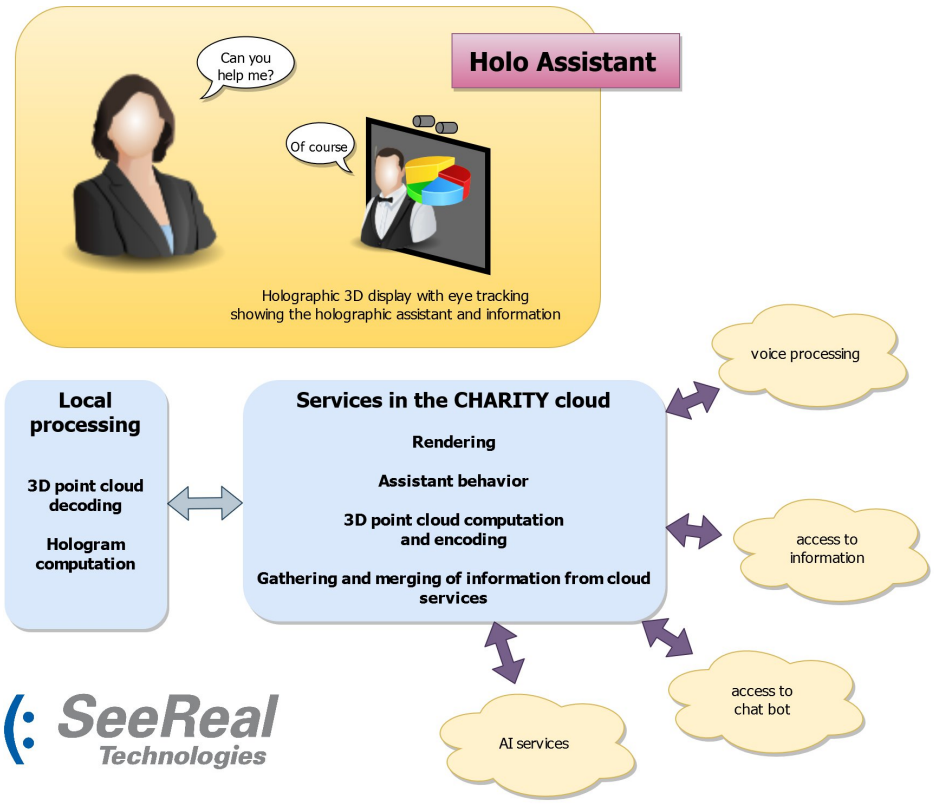


Figure 81: The Holo Assistant User Case.

The preferred data format for the management of the visual data is a Point Cloud (PC) based format which provides the following advantages:

- Multiple views can be encoded acting as a cache on client side - if many views are available, there is no missing data when data coming from the cloud is delayed
- Multiple points on one "line of sight" allow for looking around an object within one and the same data set, but also enable transparency effects
- Compared to typical point cloud, here is targeted to include only certain views or a certain view range, thus the point cloud scene must not be viewed correctly from all sides

The use case thus requires the following modules to be developed:

- Point cloud (PC) generation module (which is dependent on the rendering engine)
- PC compression
- Data transfer of compressed PC data
- PC decompression

We underline that the R&D activity in CHARITY regards the aspects just mentioned, and no other aspects involved in the fruition of the holographic content, such as the interaction modalities between the user and the avatar or the design of the user interface.

5.6.2 Point cloud encoder/decoder (PC E/D) – first design considerations

The overall process that we have to take into account for the development of the PC E/D is the following:

PC generation: We need to generate a point cloud from a generic 3D scene created with a game engine like Unity 3D. This point cloud contains all the 3D scene points to be seen from different views - at least two for the two eyes of the observer looking at the holographic 3D display. The generation could be based for instance on rendering multiple views of the Unity 3D scene, but the point cloud can be generated also in other ways. An advantage of this

method is that it relies on a very generic approach, easy to be applied to any 3D content and any 3D engine.

PC compression: the 3D point cloud needs to be compressed. Algorithms and heuristics like detecting changes from frame to frame can be applied in order to reduce the amount of data to transfer. Network quality adaption is also done here, reacting to indicators and control mechanisms from the CHARITY monitoring. For example, the resolution of the 3D point cloud could be adapted dynamically and/or the number of encoded views could be reduced.

PC transfer and decompression: the data is transferred over the network. Some feedback about network quality is provided by the receiving client to the cloud. The receiving client decompresses the received point cloud data and applies it to the existing data model - i.e. applies scene point changes for the case that only changes in the 3D point cloud have been transmitted. In the last step, depending from the actual observer's eye location at the holographic 3D display, the views needed to generate the hologram are extracted from the local 3D point cloud and the hologram is computed and presented to the observer.

To start, we need to define a data format suitable for data compression / decompression algorithm. One good option is to use a volumetric format, i.e. a voxel, and store in each cell of the voxel a 3D points plus additional information such as:

- Location in space → defined by position in the grid
- Color + optional alpha + material tag to define transparency behaviour
- material tag could be something like: fog/smoke, clear glass, distorting glass, coloured glass
- *Viewability* - definition from where the point or a certain list of points can be seen → certain eye boxes in space are needed to be defined
- If no eye boxes are defined, we assume this is not a reduced PC and could not be seen from all sides, in this case no viewability attributes are provided.

For the overall PC we need:

- Eye boxes / ranges for which this PC is valid → in 3D space we define the PC cuboid's location and size + multiple eye boxes
- Resolution in X/Y/Z → number of voxels / definition of the 3D grid
- Information about globally contained attributes → alpha and / or material tags, viewability information.

Figure 82 explains what is meant with eye boxes and 3D point viewability. Certain 3D points would be seen only from certain eye boxes while most points are visible from all eye boxes.

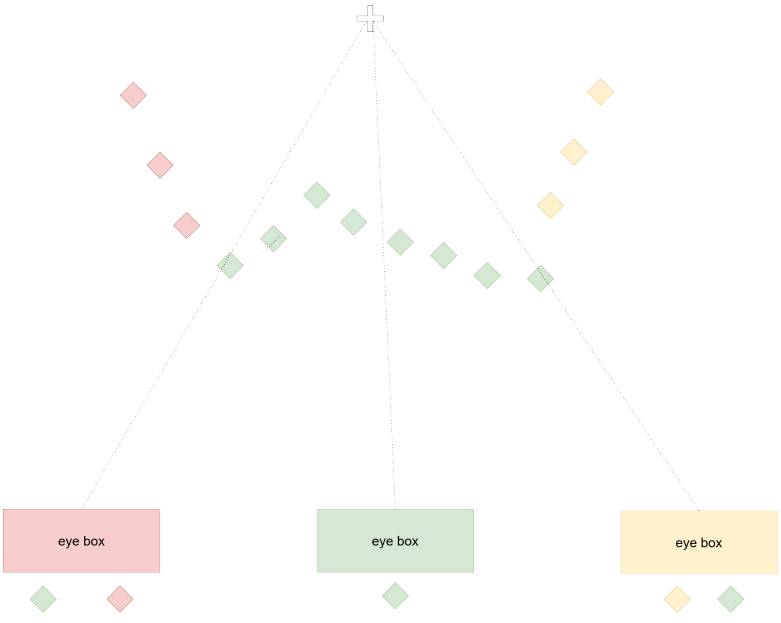


Figure 82: Relationship between the eye boxes and visibility of the 3D points.

Regarding the existing standards for point clouds, we analysed the recently published MPEG Point Cloud Compression (MPEG PCC) standard [21].

From the viewpoint of official standardization, good progress was made by the MPEG Point Cloud Compression project (MPEG PCC). It was initiated in about 2014. A call for proposals in 2017 resulted in a first draft of the standard at the end of 2018. Until today the standard is under development and there is an actively maintained reference implementation. Basically, the standard proposes two types of 3D point cloud compression - video based (V-PCC ISO/IEC 23090-5) and geometry based (G-PCC ISO/IEC 23090-9).

Figure 83: Example data sets used for comparing V-PCC (image taken from paper in Ref MPCC-1).

The V-PCC variant uses classic image-based processing (color + depth + occupancy maps). By applying common image-based compression methods (HEVC in the reference implementation), quite good compression rates can be achieved. The method is based on projection of the 3D source scene or point cloud on multiple 2D maps from different perspectives. These projections or patches are then mapped into the frame - the "atlas" - to be encoded / decoded by means of video compression. Here multiple maps are generated, attribute maps (can be RGB color but also something else), depth maps (representing the distance from the according perspective) and an occupancy map (representing valid pixels). Within a (lossless encoded) meta data channel, information about how to reconstruct these patches back into the 3D point cloud are provided within the multiplexed data stream. Within the process of generating the patches and atlas, some improvements on the data are done, i.e. detection and removal of duplicate 3D points or improvement of quality on the regions between patches (seams). As a result, very good compression rates are achieved. The MPEG PCC research group defined some reference data sets (see Figure 83), where the rates and quality of different algorithm versions and parameter variants could be measured and compared. For example, a scene with 100k points @30fps corresponds to 360Mbit/s uncompressed data rate. With V-PCC a compression to about 1 MBit/s can be achieved using version TMC2v8.0 while achieving good quality.

The G-PCC variant is based on compressing the 3D points directly one by one. Here the 3D points structure (point locations) is encoded lossless by using an octree approach (divide a cube into 8 cubes iteratively until we are at point level – noting down if there is something inside the cube or not – represented with 8 bit per cube). For encoding point attributes (i.e. RGB color), three compression methods have been developed. These methods basically make use of similarity / redundancy between colors down the octree graph. The algorithm also allows for different level of details - usable e.g. to adapt for variations in available data rate or to adapt for current detail requirement in rendering process. Currently the algorithm does not use temporal compression approaches, that would enable lower data rates in situations where the 3D scene does not change much from frame to frame – as

compared to MPEG video compression where this approach is employed and is extremely effective. However, some work in this direction may be done for the next version of the standard.

Preliminary analysis

For G-PCC some of the above data sets have been compared. For example, in a scene with 100k points at 10 fps, corresponding to 110 MBit/s uncompressed data rate, a compressed rate down to about 24 MBit/s could be achieved with good quality.

Further tests are required, but from this preliminary analysis, we can conclude that the V-PCC encoding time is too much high for our target requirements, while G-PCC approach would be better. Anyway, G-PCC has no support for taking into account visibility of the 3D points.

Hence, the key factors for the effective development are:

- to exploit the visibility information to reduce the amount of data required by the edge device; the viewpoint information can be used also to make the generation of views more efficient
- to find/to develop compression scheme alternative to G-PCC and V-PCC standards to achieve the target requirements.

5.6.3 PC generation module

Before a point cloud can be compressed, it needs to be generated. Typically, one creates a point cloud from a static 3D scene/3D model, which then can be watched from different angles at different level of details. In this case the point cloud is often directly generated from triangles or 3D-mesh.

In the context of the UC1-3 Holographic Assistant a different approach was chosen. The main goal is to convert the visual output of any 3D-application with any content including animations, complex materials and lighting into a video based, streamable 3D point cloud. The advantage is that such a point cloud enables to generate the required views from certain directions locally at the end user device, while the actual 3D-content is managed and rendered somewhere else, e.g. in the cloud. This has following advantages: first the certain views required by the output device, e.g. a holographic 3D display, are generated with very low delay independent from actual network performance. Secondly, the end user device could be something like a thin client, thus it needs only to output the required views and does not need to render high fidelity 3D-content. This is comparable to actual 2D based game streaming services commercially available.

Thus, in our particular case, the point cloud is generated from GPU renderings of the 3D scene in Unity 3D from different viewpoints (one RGB and depth image per view, see Figure 84 and Figure 85) and then merged into a single or multiple point clouds. Compared to typical point cloud data sets where the data provides information from all watching directions, full details are in this case visible only from certain angular ranges (see Figure 86). These limited valid viewing ranges or zones are generated from the different provided views mentioned above. This concept has the advantage to dramatically reducing overall amount of required 3D points in the point cloud to enable more efficient compression and frame by frame-based transfer of point cloud-based video. Frame by frame-based point cloud data will also enable the opportunity to make use of differences between point cloud frames, so for quite static 3D scenes with limited changes from frame to frame, a lower number of changing 3D points is to be expected so this can be used for efficient compression and transfer of video-based point cloud data.

Figure 84: Example of three slightly different views (depth + RGB data). These views can be merged together to form the point cloud.

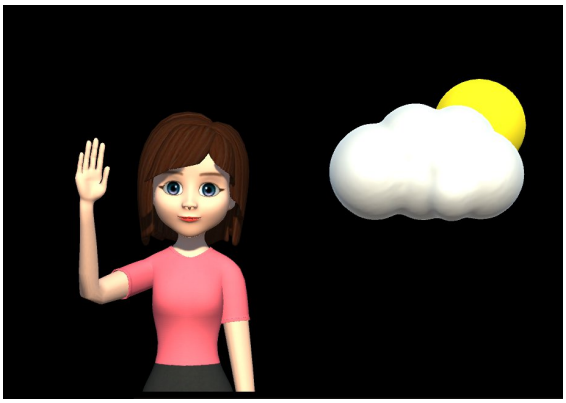


Figure 85: Rendered final result after reconstructing into an image.

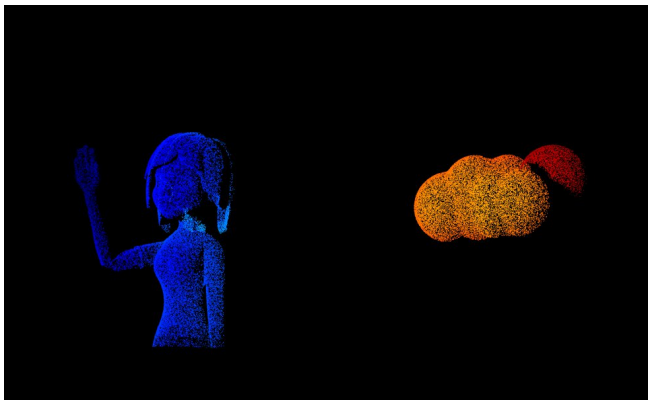


Figure 86: Example of a merged point cloud visualized from a different, invalid, perspective.

This approach has been preferred over the voxel-based one of the initial design, because is general and simple, and avoid to manage octrees that is more difficult to parallelize due to lack of memory coherence. Obviously, this approach becomes costly if a large number of views are required, but a few number of views from slightly different viewpoints, we used 8 views in our experiments, are typically sufficient for a good quality of experience.

5.6.4 PC E/D component

The point cloud resulting from merging depth maps from slightly different views, can be more efficiently represented as a 2D depth map with colour information and additional points whenever a jump in depth occurs: (see Figure 87).

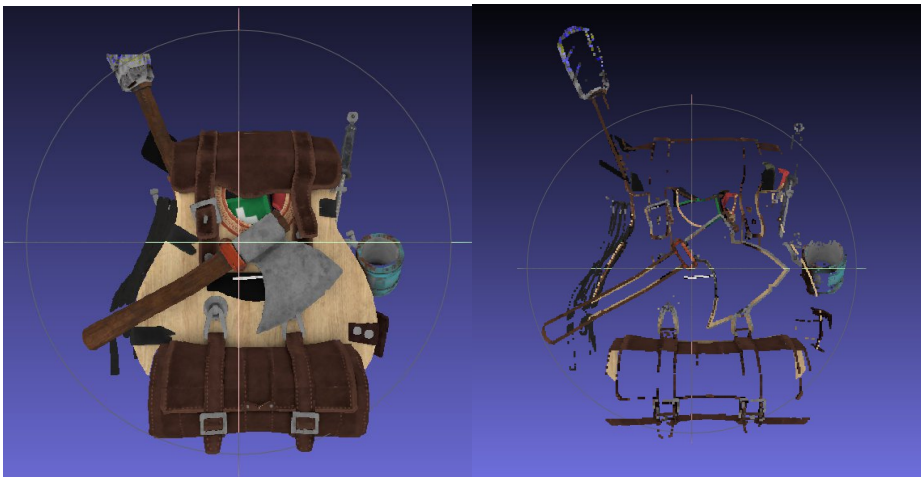


Figure 87: (Left) Depth+RGB, central view (Right) Hidden points revealed through the others views.

For these reasons, a more suitable data structure is a 2D grid where each element is the start of a linked list (usually containing zero or one element). A similar structure is a hash map with linked lists to resolve conflicts, but in our case an 'identity' hash already guarantees very few collisions, optimal memory coherence and O(1) access time.

To extract this merged mesh from the several depth maps+RGB generated from different viewpoints, we can start from one of the views (V0) depth map, then for each pixel of the following views (V1) we un-project in world space and re-project in V0 space, compare the depth to determine if the point is already present and if not add it to the merged point cloud (see Figure 88).

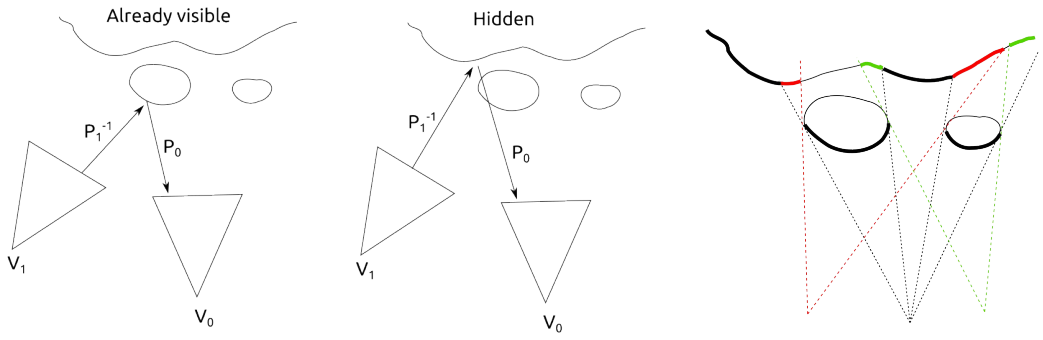


Figure 88: Depth map can be used to find the hidden points using projection between different views. In green and red the points revealed by this operation.

The most resource-intensive operation is the matrix multiplication required to convert between different views' coordinate systems. We tested this strategy, and we can process 25 views 800x600 in CPU in 30ms, including rendering and transfer of the views depth maps from GPU to CPU for a simple dataset. Another advantage of this data structure is its potential for parallelization by dividing the image into horizontal or vertical strips, depending on the displacement direction. In this prototype, we adopted the CPU parallelization strategy due to its ease of deployment.

In order to increase the resolution, we can move the bulk of these computations in GPU: instead of saving depth map and RGB, we project each point in a common final voxel space saving x y and z as additional attributes. While data size increases, we save matrix multiplication per pixel in CPU.

The linked list could be relatively easily implemented on the GPU, recycling order-independent transparency algorithms, resulting in improved computational speed and reduced data transfer from GPU memory to the CPU: we use the first depth-map as a texture and the following renderings can directly compare each pixel with the corresponding (projected and in the first view space) pixel in the texture and write the 'hidden' pixels only if the depth does not match.

Finally, the few 'hidden' pixels can be directly written to an array or compacted in a second pass, to minimize the amount of data transferred back to the CPU.

The final data to be compressed consists of a depth+rgb map, where depth is quantized accordingly to the precision needed by the hologram projector, and a small array of xyz+rgb points.

The RGB+depth data is encoded as two images, with the RGB image converted to YUV and the depth data color-coded. These two image sequences are video encoded using the H.264 video codec, muxed into an RTP stream. This approach allows for different parameters to be used for depth and colors, as their effects on visual fidelity are distinct. We chose lossless encoding for the depth stream, as even small errors result in noticeable artifacts, especially around depth discontinuities.

The additional hidden points and eventual parameters (resolution changes, for example) are entropy compressed and transferred using the SEI message mechanism of the H.264 codec. We opted for LZ4 library as the best compromise between compression ratio and speed. For experiments conducted on the possible geometry entropy encoders, see the next Section (Section 5.6.5).

A result of encoding, transmitting, and decoding 8 views is shown in Figure 89.

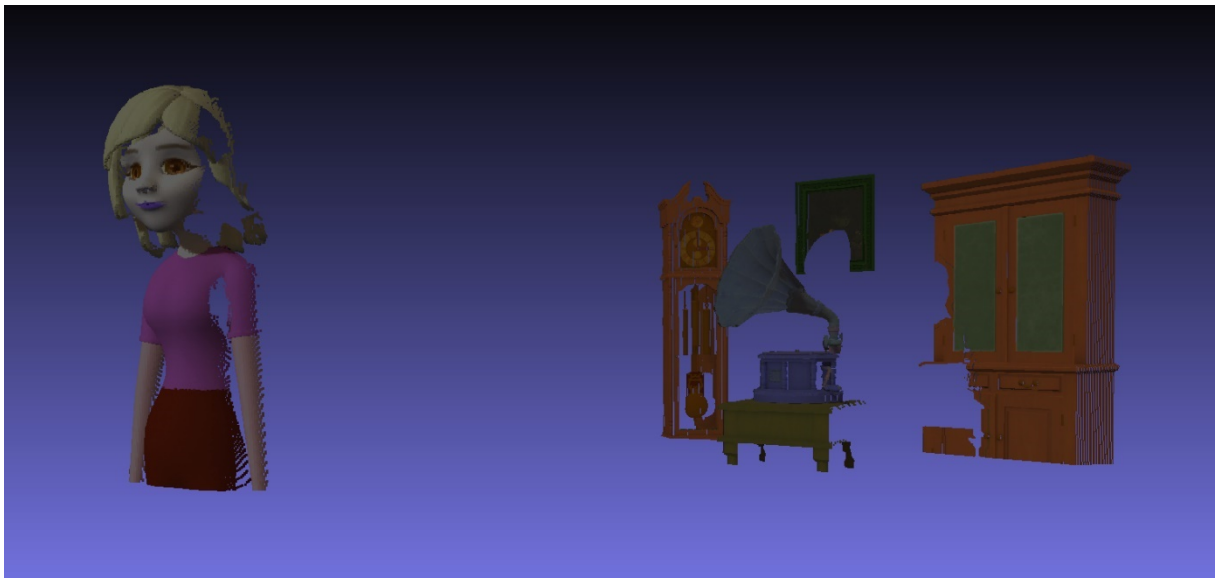


Figure 89: Example of a point cloud decoded from 8 views with small offset.

Performances

On a test dataset of about 50 frames, selecting lossless video compression we manage 20fps on a common PC running both client and server using 4 cores. The most time consuming part is the point cloud extraction and in particular the per pixel matrix multiplication required to compare the different views, amounting to about 50% of the execution time. Further optimization in term of parallelization (for example socket transmission is not currently parallelized with point cloud extraction and video compression) or more importantly GPU matrix multiplication and point cloud extraction would easily bring the fps above 30 or allow for additional views.

The compression ratio is dataset dependent: H264 performs an excellent job at exploiting temporal coherence. The hidden points transmitted separately (not video encoded) depends on the depth complexity of the scene.

Limitations

Compression artifacts in the depth channel are extremely apparent around the edges of 3D shapes, so it is not advisable to lower the CRF quality setting parameter. This problem could be alleviated using post processing filtering on the decoded video stream.

5.6.5 Geometry encoding - evaluations

Data compression trades CPU computation and latency for reduced network bandwidth usage: the effectiveness of a compression algorithm depends both on compression ratio which determines the bandwidth reduction and on compression and decompression speed. However, long computational time might negate the bandwidth advantage.

Compression and decompression speed of an algorithm have always played a crucial role in determining its success, where good compression performances are especially difficult to obtain. Historically, in Computer Graphics, geometry compression algorithm competition has been focused mainly if not almost exclusively on compression ratio, and consequently widely used compression algorithm has become available only very recently when good performances combined with fast decompression have become possible, (Draco [15], Corto [18], Potree [19]) especially on the Web where the limited performance of JavaScript prevented a solution for a long time, while at the same time, bandwidth limitations made the problem more pressing.

We performed an evaluation of the performances of the available open-source libraries on a sample point cloud containing 20K points with colour information, weighting 570KB in raw binary format. All tests were performed using the same attribute and position quantization and a single thread processing. Results are reported in Table 24.

Table 24: Evaluation geometry compression algorithm

Algorithm	Compression time in seconds	Compressed size
Quantization	< 0.001s	140KB
gzip -1	0.004s	100KB
gzip: -7	0.018s	90KB
Corto	0.005s	71KB
Dracol	0.030s	71KB (missing colours!)
Tmc13	0.138s	53K

All geometry compression algorithms perform some form of quantization on the vertex position and attributes. Due to the limited size of the dataset, drastic quantization can be performed on the positions (from 32 bits to 11 bits per coordinate) at a negligible cost in quality. Larger datasets can be easily cut into blocks so the numbers from this experiment remain significant.

As a comparison we tested a zip library (actually zlib), a general-purpose compression algorithm. The low compression ratio is mainly due to the fact that it cannot exploit the geometric coherence of the point cloud. Due to the relatively low compression ratio, there is a small difference in compression ratio when changing the dictionary length of the algorithm. On the other hand, large dictionaries become a large penalty in decompression time (4 times here) mostly due to the fact that the dictionary will not fit in the L2 cache generating many cache misses. Other entropy compression algorithms (LZ4 for example) have been tested, with much faster compression timings but worse compression ratio.

Corto [18] adopts a very simple Morton-code based geometry compression with a difference encoder for the attributes (colours in this case).

Tunstall [20] (which is basically a reverse Huffman) is used as an entropy coder due to its extreme speed in decompression while still being fast enough in compression and having compression ratio similar to Huffman. Corto is able to encode five million vertices per second, while decoding at around 25M vertices per second. Adopting Huffman instead would probably reverse the speeds. Other entropy coders could be used and offer different trade-off between speed and compression ratio.

Draco [15] adopts a similar approach based on differences combined with arithmetic entropy coding. Surprisingly the compression ratio is worse while colour information has not been encoded (command line software does not support it). Unsurprisingly, due to more sophisticate entropy coding, the

compression timing is 5 times worse. Draco offers much better trade-off for meshes.

Tmc13 [21] offers the best compression ratio (1:10), at the cost of a long processing time 0.14s, 142K triangles per second. This software offers a very large set of parameters to be tuned, coupled with a lack of a decent documentation or guidance. We tested a (very) large number of configurations with mixed results. We are confident that marginally better results can be obtained, the picture is not going to change substantially.

For each compression algorithm, speed and compression ratio defines a bandwidth above which it makes no sense to compress as it would take more time to compress/decompress the data than to send raw, quantized (11 per coordinate 8 per colour channel, for small datasets, in total 58 bits) data.

Tmc13 becomes useful when the network bandwidth is smaller than (58/8)*142K/s ~ 1MB/s, while Corto keeps being competitive up to 58*5M/8 = 36M/s. For bandwidth lower than ~1.3MB/s higher compression ratio of Tmc13 allows to better make use the limited bandwidth.

Since it is relatively easy to perform point data compression in parallel, adding computational power allows Tmc13 to remain competitive with higher available bandwidth.

The compression algorithm could be very easily swapped for a different one at any time in the streaming depending on bandwidth or CPU limitations, and the most promising algorithms to adopt for geometric compression, according to these preliminary investigations are Corto, Tmc13. Also the simple quantization and LZ4 are competitive.

5.6.6 Conclusions

The current prototype, called “Cloudstream”, is implemented as a C++ library prototype. This library consists of a server component that converts a set of RGB+depth data into a streamable dataset made available through a socket, and a client component that connects to the socket and provides a point cloud (RGB + XYZ) for rendering. It provides functionality for monitoring performance in terms of framerate, compression ratio, and timing for various tasks. The library also allows for throttling computational effort and data transfer at the expense of resolution and quality. The library is available on CHARITY GitLab[42]. The dependencies are OpenCV, libav, and libz4, and it has been tested both on Linux and Windows. This library has been integrated and tested in the UC Holographic Assistant.

SRT developed and tested also an alternative, similar approach. The main difference consists in the processing and transmission of the hidden points. Instead of the linked list, only the first conflicts of the same pixels are saved to another texture which is then encoded in a second video stream. The performances of both algorithms are similar (around 5 fps) due to the bottleneck of the CPU matrix transform of the library.

The CPU version of the library is general, easily integrable in different rendering environments, and further optimization can be implemented, as previously discussed, to increase the final number of fps. The GPU version needs to be specific for the rendering engine used, and this is not ready at the moment of writing. Anyway, we can estimate, according to tests conducted, that the usage of GPU brings the final performance around 30 fps for a video of resolution 1280 x 752.

5.7 Virtual Experiences Builder Platform

Cyango Cloud Studio is the name of the software platforms related to the UC2-2 VR Tour Creator Application in CHARITY. Cyango Cloud Studio is an easy-to-use and resourceful tool that will help Explain, Show, Teach and Sell directly inside 360° Videos.

The 360° Video Editing Software provides access to technology to anyone who wants to create marketable immersive digital experiences at an affordable price. Cyango Cloud Studio can be used to

[42] https://gitlab.charity-project.eu/ponchio/cloudstream.

showcase a story that help enhancing the business, product or brand of a company through a dynamic and unique experience.

One of the main goal inside CHARITY is to deal with the micro services of the platform to provide a better performance and better user experience. One of the biggest achievements is the total migration of all the services of the platform to CHARITY, making it 100% cloud native. There are many problems addressed in CHARITY, mainly related to the livestreaming, editing in real time of media content, and transcoding / converting files uploaded by the users to create multiple adaptive levels.

Besides this we have been focusing and developing many features in the back-office while keeping harmony with the user interface and user experience according to the feedback we gathered via user focus groups, meetings, calls and demos at events showing our software.

5.7.1 Milestones

We defined many milestones on our roadmap. Below we describe shortly the most relevant ones:

5.7.1.1 Refactoring of the 3D Web engine framework

We did a total refactoring of the 3D Web engine framework. We used Aframe[43] in the past, but we decided to migrate to a more modern and compatible framework with React.js, which is called React-Three-Fiber[44]. This change of framework required an extensive code re-factoring, as its logic was different from Aframe, although both used Three.js engine.

This assured better scalability, allowing a more streamlined way of coding, and it is compatible with our team's knowledge.

5.7.1.2 Friendly to use

We also completely re-designed and improved the UI/UX of the platform. All the cloud features we had implemented needed to be in sync with the UI/UX, for example the video conversion and assets management needed to have many UI/UX features to actually work. This also allows the software to be actually user-friendly therefore marketable. The figures below (Figure 90 and Figure 91) show some screenshots of the whole platform after the re-design.

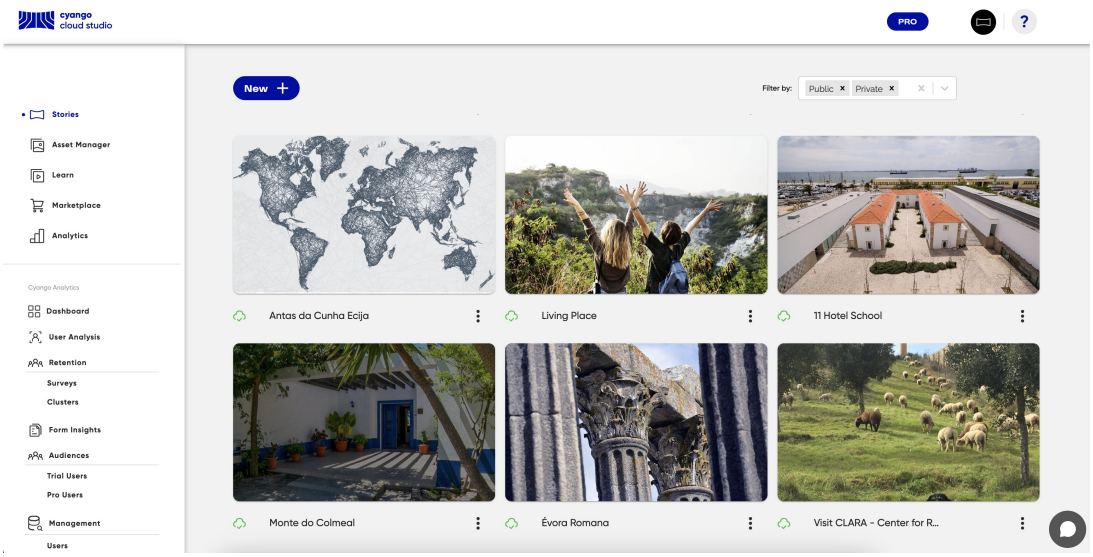


Figure 90: New design of Cloud Studio (screenshot 1).

[43] https://aframe.io

[44] https://github.com/pmndrs/react-three-fiber

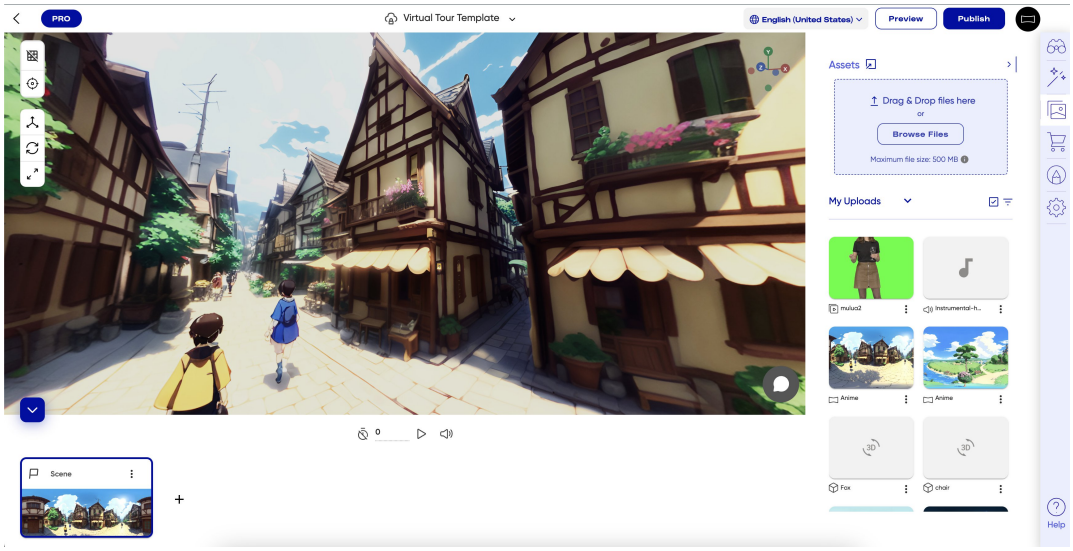


Figure 91: New design of Cloud Studio (screenshot 2).

5.7.1.3 Video Editor

We also made important research about how to achieve an important feature the users requested, which is the online video editor, that allows the user to edit the video and audio.

We explored a possible video editor tool solution based on the FFMPEG+WASM[45], a pure web assembly port of FFMPEG which could allow editing video, audio and stream inside the browser. Unfortunately, we found some critical technology obstacles so that this solution would not be feasible on the client side. Hence, we have decided to create a solution that works on the edge-cloud. We call this component the *cyango-worker*, which is mainly focused on heavy tasks like video and audio conversion, image conversion, and 3D models conversion. To be able to do heavy tasks asynchronously without blocking user's work on the frontend, we had to implement a Kafka cluster to act as a message broker between cyango-backend component and cyango-workers.

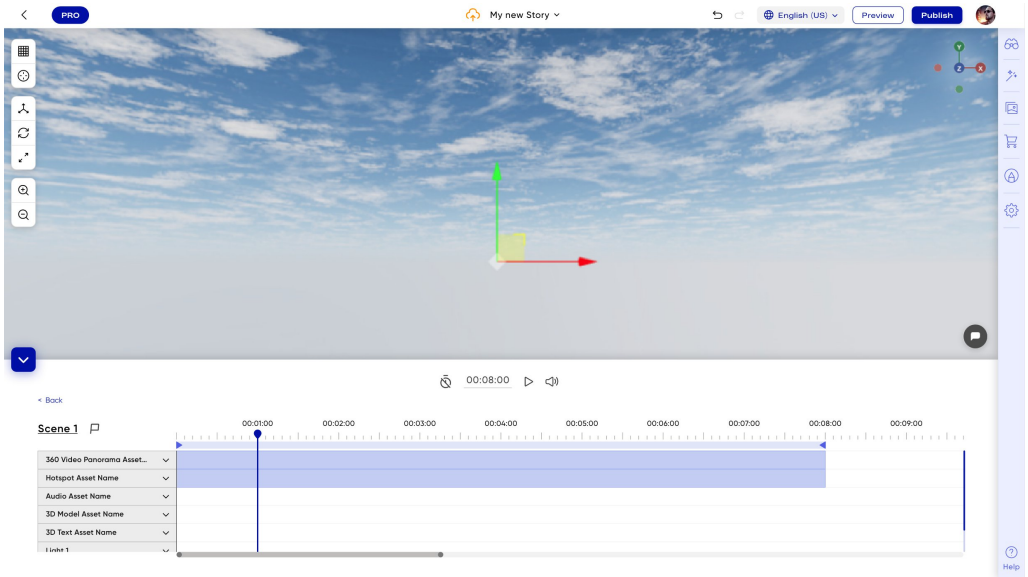


Figure 92: Video timeline editor.

We also designed a screen of how this video editor tool would be like in Cyango Cloud Studio, shown

[45] https://github.com/ffmpegwasm/ffmpeg.wasm

in Figure 92.

5.7.1.4 360 VR Livestreaming

The implemented 360 VR livestreaming feature is still under optimization.

Regarding the performance, a series of livestreaming tests using a high quality 360 camera and a server of the company has been conducted. These tests allow to understand what factors are preventing a good user experience.

The setup used in such tests were: 360 camera streaming in Lisbon, Portugal and the consumer user located in Évora, Portugal. The 360 camera was streaming to a service inside a docker container hosted in our Synology NAS 918+[46] . This container is built on:

- Nginx 1.17.5 (compiled from source)
- Nginx-rtmp-module 1.2.1 (compiled from source)
- FFmpeg 4.2.1 (compiled from source)

and allows to stream to a RTMP url using a server public IP address, and then the front-end app consumes the url called https://live.cyango.com . This url points to the docker container in a server of the DOTES. This docker container receives a video stream from the 360° camera via RTMP and then uses ffmpeg to convert the video in real-time to the HLS format so we can consume it on the front-end.

The network parameters of each endpoint are the following:

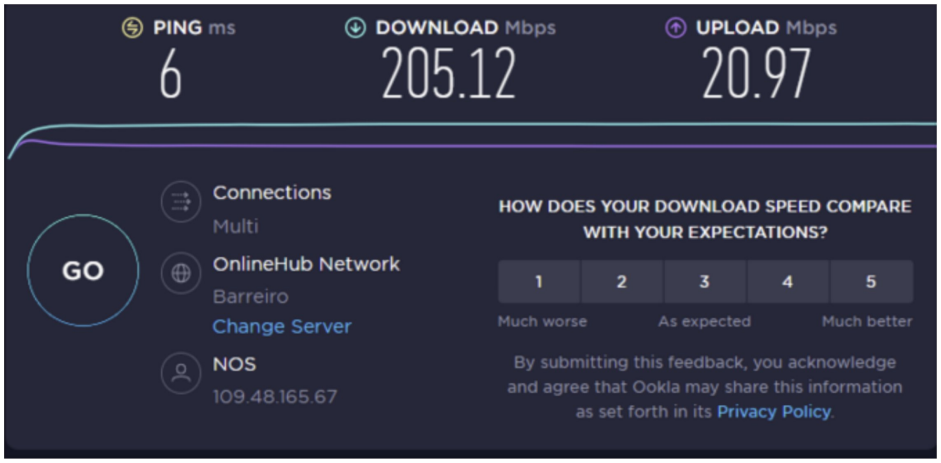


Figure 93: Camera end network settings.

[46] https://www.storagereview.com/review/synology-diskstation-ds918-review

Figure 94: End user network settings.

We did some tests with different parameters as detailed below. These tests were conducted to understand the performance of the algorithm and protocols used in Cyango Cloud Studio.

Test 1

In the first test the camera was streaming video at 4k 3840x2160 with a bit rate of 10MB/s. In this first test we experienced an high number of video stops during playing, approximately 10 times per 20 seconds of streaming. An accurate perceptual measure of this problem is under evaluation but the streaming quality has shown to be clearly insufficient.

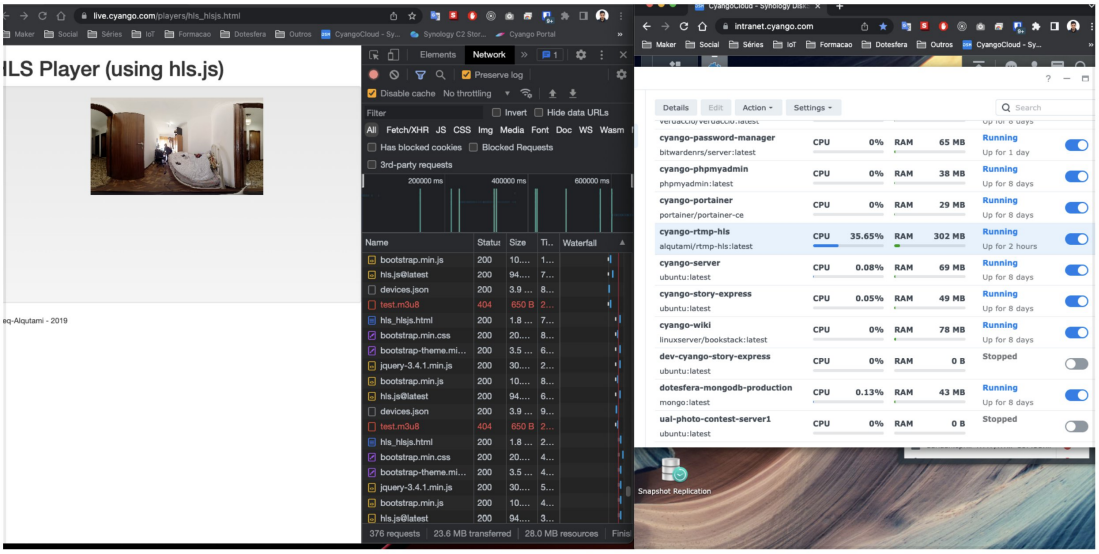


Figure 95: Screenshot of the livestream test

Test 2

In this test we lowered the camera settings to 1440P 2560x1440 with a bit rate of 10MB/s, and still experienced video stops similar to test 1.

Test 3

We lowered the camera settings to 1080P 1920X1080 with a bit rate of 10MB/s, and still experiencing the same as test 1 and test 2.

Test 4

In this test we used the camera settings as 960P 1920x960 with a bit rate of 5MB/s. And in this test the video plays without stops, but we noticed about a 3 minutes delay. We could confirm this delay, because we had a phone call between the two DOTES collaborators confirming the delay.

Test 5

In this test we lowered the camera settings to 720P 1440x720 and a bit rate of 5MB/s. In this case the video plays without stops and with a delay of about 45 seconds, using the same process as test 4.

From these preliminary tests, we conclude that the server we used is the major factor of the delay, because it does not have good hardware resources to quickly transcode the video coming from the stream to HLS. In the next, we exploit resources made available by CHARITY partners to make additional tests. Also, some tweaks could be done on the algorithm approach. In the next tests iteration, we will research about Low latency HLS[47] to assess the latency reduction using this protocol.

[47] https://developer.apple.com/documentation/http_live_streaming/enabling_low-latency_http_live_streaming_hls

As the development was progressing, we implemented a new micro-service on the edge-cloud of CHARITY which we call *cyango-media-server*. This component is based on Oven-Media technology[48] and allows to stream from a 360 camera via RTMP url, that in the edge is then converted to HLS protocol which is readable on the frontend.

5.7.1.5 Workers

One of the complex features we implemented was the component called cyango-worker. This component lives on the cloud of CHARITY and is responsible for heavy tasks like the video editor, video and audio transcoding, and image conversion.

This component is always actively listening to kafka messages from the Kafka message broker implemented in CHARITY. One of the cases why we need a message broker like Kafka is because the user can upload many videos, and the cyango-workers receive the orders asynchronously to start transcoding or converting the videos, and when finished the cyango-worker starts doing the next task waiting in the kafka message bus. The cyango-worker containers can also be scaled in terms of replicas according to the demand. If there are many users sending many videos in a short period of time, the system must be capable of adapting the resources. This is also related to the work done in a paper we co-authored within CHARITY scope of work about Intelligent Multi-Domain Edge Orchestration [61].

5.7.1.6 WebXR compatibility

We also achieved major steps on the compatibility with WebXR. Traditional DOM elements on the browser are not compatible with. We had to replicate many basic UI/UX elements we had on mobile and desktop to WebXR. For example, creating buttons, images, image carousels, popups, hotspots, video players, audio player, are just a few of the components we had to do to make our application work on WebXR. This allows us to test the 360 livestreaming for example and measure its performance. This transition allows our platform to become ready for the spatial computing era. Spatial interactions like hand tracking and poses, grab 3D meshes, anchors, and teleport, and interact with 3D UI canvas. With the help of an open-source repository[49] and engaging and contributing to this repository community we had success in implementing cross-platform UI elements that work in WebXR and traditional DOM.

Figure 96: WebXR environment on Cyango.

[48] https://airensoft.gitbook.io/ovenmediaengine .

[49] https://coconut-xr.com .

5.7.1.7 Analytics and Monitoring

We also implemented the analytics and monitoring system for our platform, most precisely implemented on cyango-backend, cyango-story and cyango-cloud-editor components.

This system allows to measure how much time the users spent using the platform, measure the performance of the app on the client side, and other important analytics that allow a better understanding of the performance and help us on deciding what is the best direction of the app.

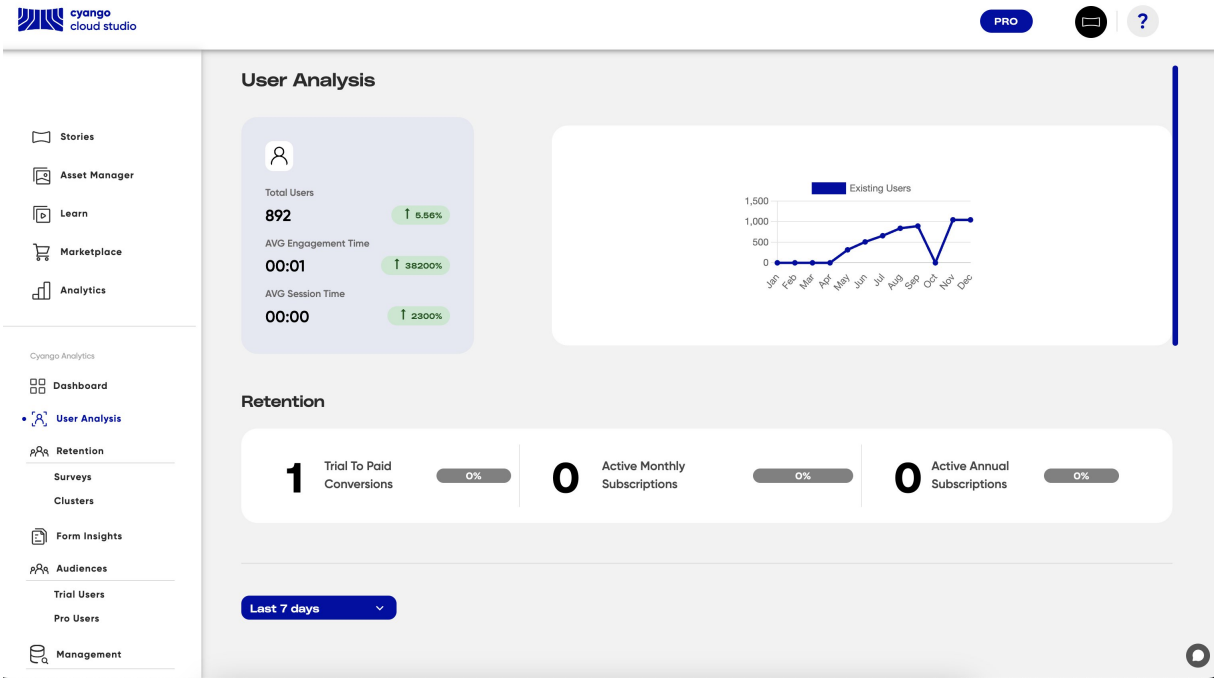


Figure 97: Dashboard of analytics inside the platform.

We also implemented the monitoring part of the 360 livestream that sends metrics about the connection performance and latency params to a Prometheus component that lives on CHARITY cloud.

Until now we did an official test of around 2 hours livestreaming from a 360 camera with 2k resolution video and a 20 mbps of bitrate, but we haven't had a good quality of experience. We need to improve and research more on how to achieve a good experience.As depicted in Figure 98 and Figure 99 below, the measurements about the total round trip time and available incoming bitrate. These measurements can vary depending of the video resolution and bitrate that is being streamed. In this case the video had a resolution of 5k and an approximate bitrate of 2 mbps.

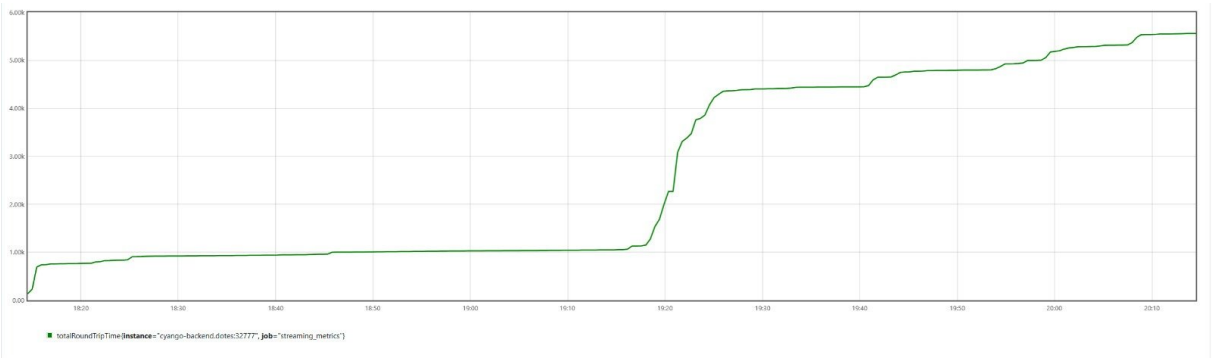


Figure 98: Total round trip time of a 360 livestream test done within 2 hours.

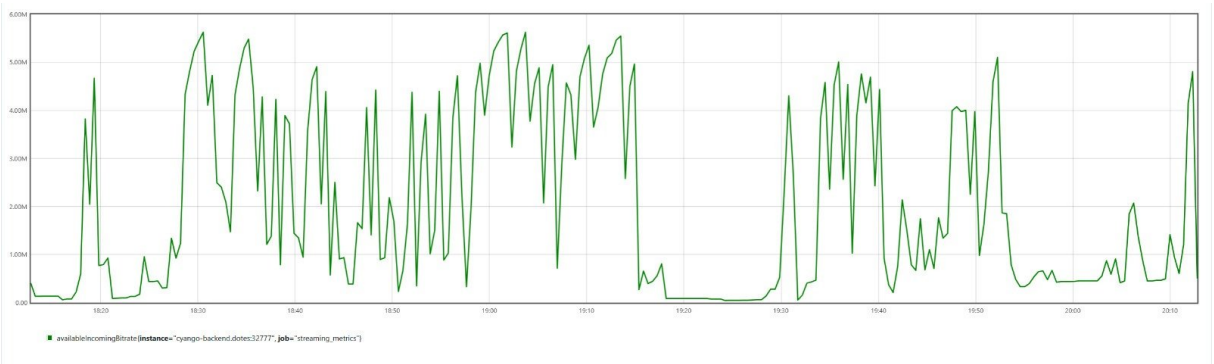


Figure 99: Available incoming bitrate of a 360 livestream test done within 2 hours.

5.8 Immersive services and cross-video streaming experiments

Immersive services, such as VR, AR, and XR applications, have stringent latency and high bandwidth requirements, often limited to standalone and costly installations [50]. Despite this, they are expected to dominate next-generation networks due to their popularity and anticipated traffic volumes, making XR applications mainstream by 2030. For this purpose, Standards Development Organizations (SDOs) are actively addressing enabling technologies for VR, AR, and XR applications to ensure interoperability. IEEE P2048, ETSI, 3GPP, MPEG, W3C, and IETF are among the organizations developing standards and frameworks for various aspects of immersive experiences, such as streaming, data representation, and communication protocols. Effectively, in the realm of Holographic-Type Communication, HTC is recognized for its role in the Network 2030 initiative of ITU-T, focusing on multi-sense networks and haptic communication services. MPEG is contributing to immersive data representation through its MPEG-A series, introducing formats like OMAF for 360 videos and standards for volumetric video under the MPEG-I project. For Augmented Reality, ETSI has defined the Augmented Reality Framework, offering a functional reference architecture for AR components, systems, and services. The architecture includes layers for hardware, software, and data, specifying component placement and potential offloading to the cloud or edge.

Regarding immersive services' Quality of Service (QoS) requirements, Table 23 summarizes latency, bandwidth, and reliability needs for each use case. Figure 100 illustrates these requirements from a cloud perspective, mapping interactivity against bandwidth for various immersive services. It shall be highlighted that the use cases shown in Table 23 and Figure 100 strongly relate to the CHARITY use cases. Effectively, CHARITY's "real-time holographic applications" fall under the HTC telepresence use case. CHARITY's "immersive virtual training" falls under both remote services and social tourism use cases, whereas CHARITY's "mixed reality interactive applications" fall under both the remote services and cloud gaming use cases.

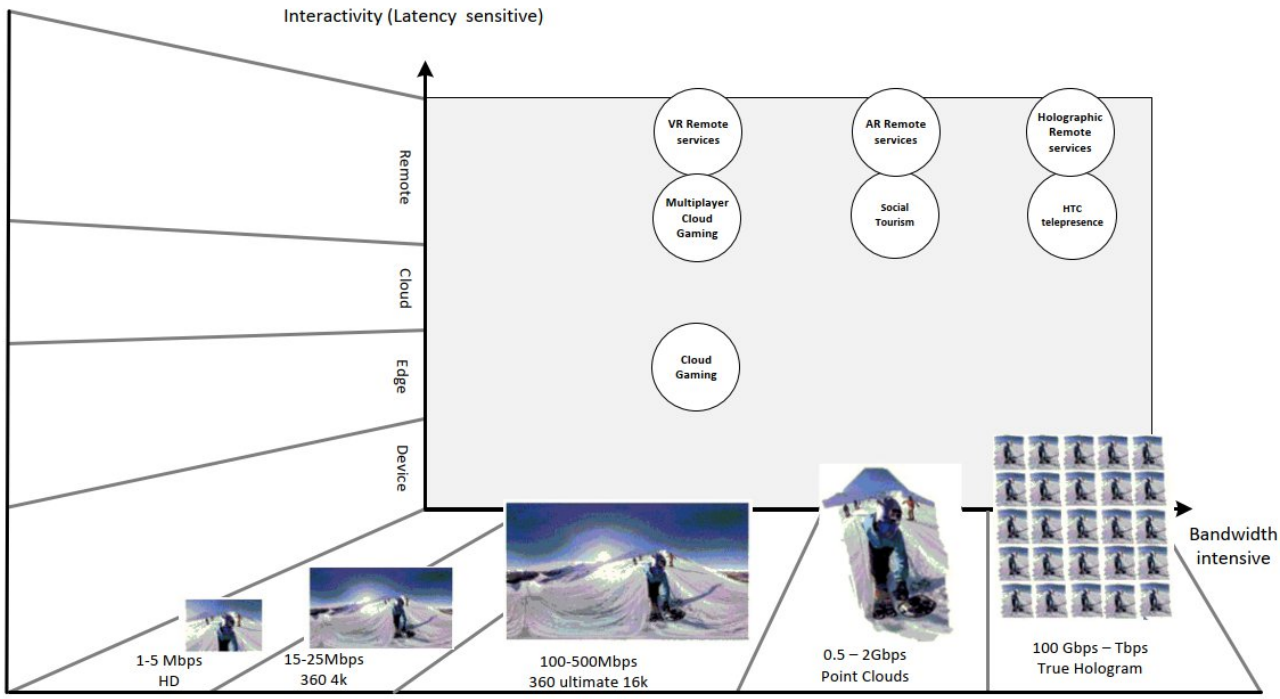


Figure 100: XR use cases and their requirements [50].

Table 23: Requirements of XR services [50].

Use Case	**QoS Requirements**		
	Latency	**Data Rate**	**Reliability**
HTC telepresence	320 ms	2Gbps to 2 Tbps	99,999%
Remote services	less than 5 ms	15 Mbps to 2 Tbps	99,999%
Social tourism	320 ms	15 Mbps to 2 Gbps	99,999%
Cloud gaming	30 to 150 ms	15 Mbps to 500 Mbps	99,999%

We conducted experiments to assess the impact of various metrics on Glass-to-Glass (G2G) latency in immersive services, using the example of a remote control system with VR headsets managing devices like robots. In the conducted experiments, the GTG latency is determined by activating a LED adjacent to the 360 camera at time t1. A light sensor connected to the HMD captures the light at time t2. Both time points are logged on a Single Board Computer (SBC). The GTG latency is calculated as the difference between these two time points, namely (t2 - t1).

End-users can smoothly control industrial equipment through hand motions and HMD controllers, facilitated by 360-degree video from remote devices. Both video streaming and device control demand low latency and high reliability. Analyzing the streaming aspect, we focus on the communication between VR HMDs and remote devices, utilizing real-time protocols like RTP or WebRTC. While RTP is designed for broadcast scenarios, our VR-based remote control scenario demands direct communication for minimal end-to-end latency, distinguishing it from traditional use cases like live concerts [51].

In Figure 101 the GTG latency performance between Oculus Quest and the 360-degree camera is depicted. Two cameras were employed in the experiments: i) Vuze XR and ii) Insta 360 Pro, both streaming at 30 FPS using the RTMP protocol. The results, as shown in Figure 101, indicate that Insta 360 Pro outperforms Vuze XR by approximately 700ms. This difference can be attributed to two factors. First, for live streaming, Vuze XR requires a connection to a smartphone via WiFi Direct,

introducing additional delays. Second, Insta 360 Pro benefits from its built-in RTMP server, significantly reducing GTG latency. In Figure 102: Performance evaluation of RTMP in case of Insta 360 Pro [51]. the RTMP protocol performance of Insta 360 Pro is plotted against GTG latency as a function of Display Refresh Rate (DRR). The trend shows a decreasing GTG with increasing DRR, but with diminishing returns. For example, increasing DRR from 144 to 244 results in only a 3ms GTG reduction, alongside increased energy consumption. Current HMD displays typically max out at 120Hz refresh rate. To assess the impact of Streaming Frame Rate (SFR) on GTG latency, Figure 103: Performance evaluation of RTMP for varying DRR values [51]. Figure 103 and Figure 104 present evaluations. Figure 103 illustrates that, for various streaming rates, increasing DRR from 50 to 244Hz leads to a modest 40ms GTG reduction. Figure 104 demonstrates a more significant GTG reduction with increased SFR, such as a 100ms decrease when going from 20 FPS to 30 FPS. However, diminishing returns occur with higher SFR values due to increased bandwidth and computational demands, causing bottlenecks in both network and computation resources at rates like 120 FPS or 240 FPS.

Figure 105 provides a detailed GTG latency analysis, emphasizing the influence of both camera refresh rate and Streaming Frame Rate (SFR) on immersive data processing. A 30 FPS camera refresh rate contributes at least 66ms to the processing time at the sender, excluding additional processing delays like stitching and encoding. Edge processing introduces a minimum of 33ms. Delayed packet arrival affects decoding times at both the edge cloud and end device. While increasing SFR significantly reduces GTG latency, it demands intensive computation to handle the higher throughput. However, in holography, escalating SFR may compromise the streaming experience due to the need for substantial network resources, potentially causing packet loss and delayed arrivals, ultimately impacting GTG latency.

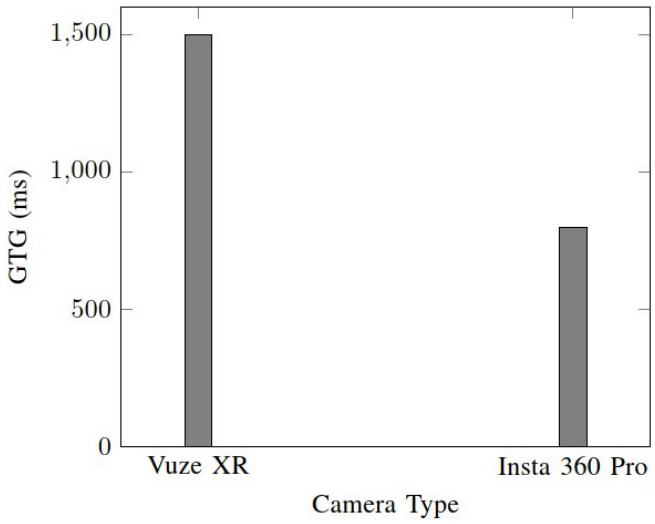


Figure 101: GTG latency for two types of 360 cameras [51].

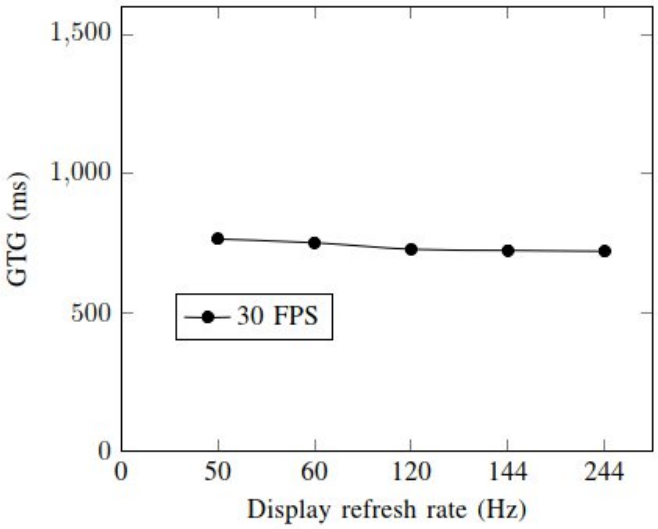


Figure 102: Performance evaluation of RTMP in case of Insta 360 Pro [51].

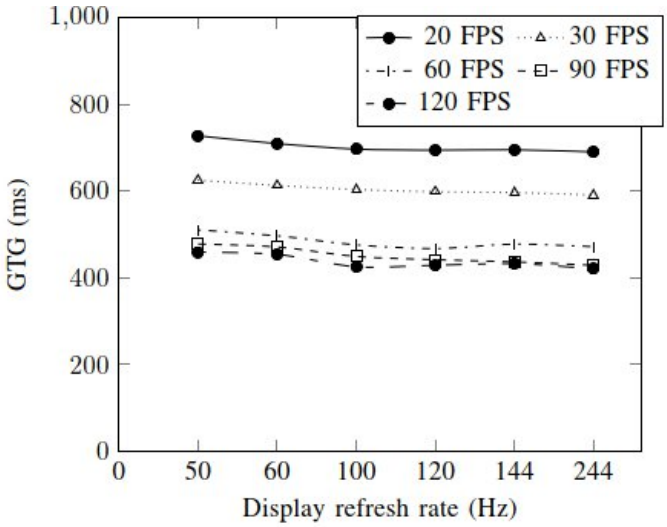


Figure 103: Performance evaluation of RTMP for varying DRR values [51].

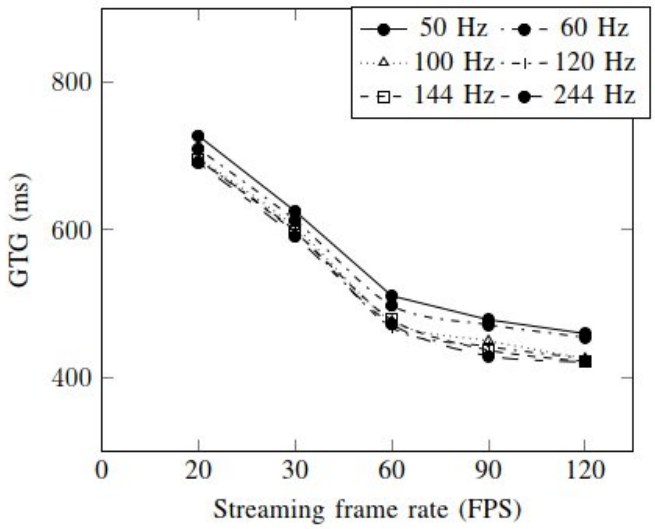


Figure 104: Performance evaluation of RTMP for varying FPS values [51].

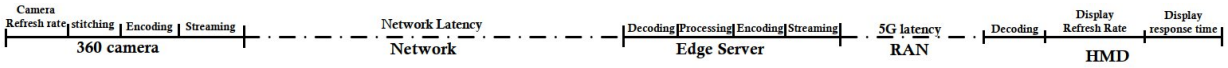


Figure 105: Detailed analysis of the GTG. The size of each box does not reflect the processing time of the respective task [51].

In another experiment focusing on remote driving assistance for autonomous vehicles, we explored the efficiency of using a 360° stream augmented with vehicle-related information, delivered via an HMD wearable device by a remote human operator (RHO) [52]. The setup involved a Labpano Pilot One 360° camera, an Intel® Xeon® server, and a Latitude 7490 receiver device. We assessed various streaming protocols and modes, comparing G2G latency for 4K 360° live streams. Results indicated sub-second G2G latency, crucial for the RHO to react promptly. Three streaming scenarios (RTMP-PUSH, RTMP-PULL, RTSP-PULL) were evaluated at different bitrates (6Mbps, 15Mbps, 30Mbps). Higher bitrates resulted in larger stream sizes and improved quality but increased latency. The evaluation focused on G2G latency at different stages: camera screen, network transmission to RHO's display, and source to RHO. The DASH technique was considered for bitrate adaptation, but its latency was relatively high. A stream selection approach based on parameters like location and video quality was proposed for better system performance.

The VR experience, built using A-Frame, allowed RHOs to view the stream through VR via desktops or HMDs. Relevant vehicle information was displayed, enabling the RHO to send commands for highway engagement [52]. Results showcased latency comparisons at different bitrates for each streaming scenario, emphasizing the impact on network latency. Notably, RTSP had higher latency than RTMP, and higher bitrates increased G2G latency due to network delays. Effectively, Figure 106 illustrates the average G2G latencies and standard deviations across various scenarios at encoding rates of 6Mbps, 15Mbps, and 30Mbps. At 6Mbps (Figure 106 (a)), the acquisition, encoding, and display times at the camera's display are consistent (ranging from 144ms to 164ms) across all scenarios. Notably, the RTSP protocol exhibits significantly higher network latency compared to RTMP push and pull, with a minor difference between the two RTMP modes. For 15Mbps (Figure 106 (b)), acquisition and encoding latencies are comparable to those at 6Mbps. However, network latency is markedly higher for RTSP and RTMP pull modes, while RTMP push mode maintains similar network latency at both 6Mbps and 15Mbps, resulting in the optimal G2G latency to the RHO's display. At 30Mbps (Figure 106 (c)), there is a noticeable increase in G2G latency for all streaming modes, primarily attributed to network delays despite the camera's hardware capability to handle different bitrates simultaneously.

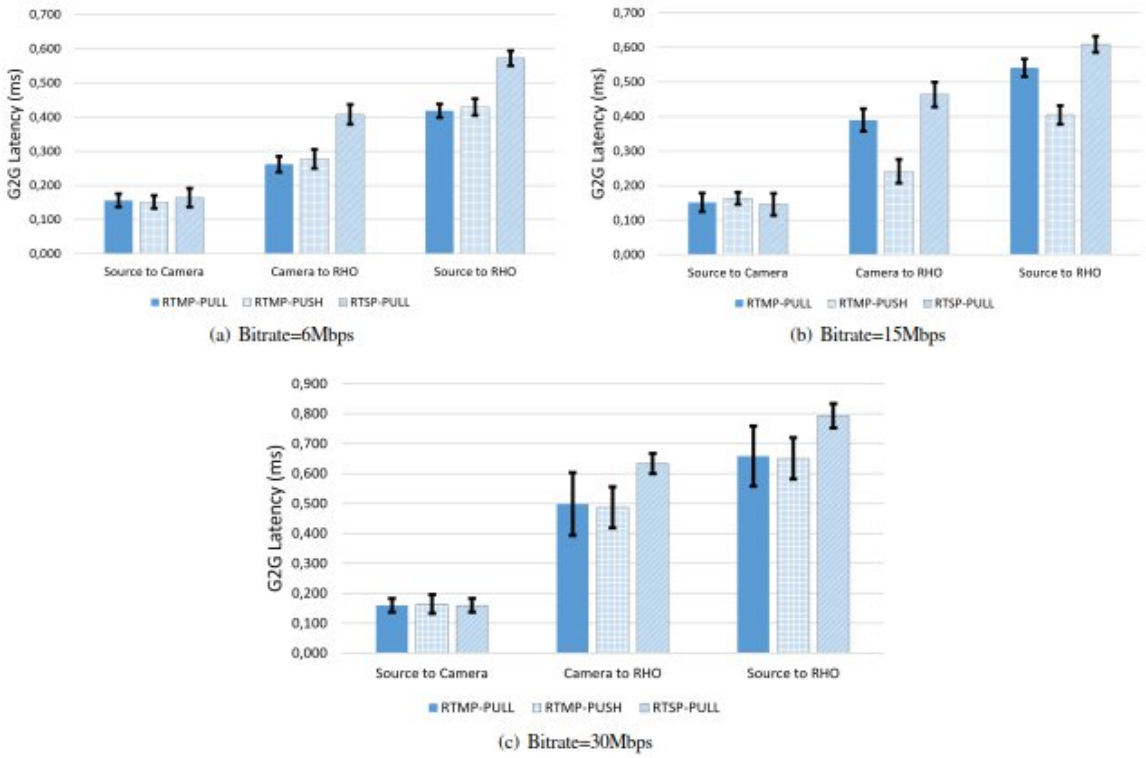


Figure 106: GTG latency of different protocols when streaming at different bitrates [52].

In a third experiment, we explore the realm of VR-based remote control for Unmanned Aerial Vehicles (UAVs), aiming to deliver a fully immersive user experience. Two distinct architectures take center stage: the global architecture and the detailed architecture, depicted in Figure 107 and Figure 108 respectively. Each component is meticulously showcased, elucidating its interactions with other system elements. The global system architecture, depicted in Figure 107, provides a comprehensive overview of the system's components. In contrast, the detailed architecture in Figure 108 delves into measured delays, offering a complete testbed that chronicles the interactions of components in a chronological order. Envisioned as a real-life implementation, the architecture incorporates micro-services based components at the edge server. Containerization is employed for components to ensure both portability and scalability. The overarching goal is to empower end-users with the capability to control remote UAVs using a 360-degree stream and sensed data from the UAV's location. At the remote location, the UAV is equipped with a 360-degree camera that live streams to the user's Head-Mounted Display (HMD). Upon receiving the camera stream, users can manipulate the remote UAV using body movements and HMD controllers [53]. The primary objective of this experiment is to conduct a detailed analysis of different latencies under varied conditions, encompassing i) mobile networks such as LTE, 5G, and WiFi, ii) user reactions, and iii) video qualities. In this study, beyond G2G latency, other critical latencies are considered. Human reaction latency characterizes the delay a user experiences in perceiving a visual event and reacting to it. Command transmission latency represents the time it takes for a user command to reach and be executed by the UAV. Figure 109 visually illustrates the various analyzed delays, providing a comprehensive view of the experiment's findings.

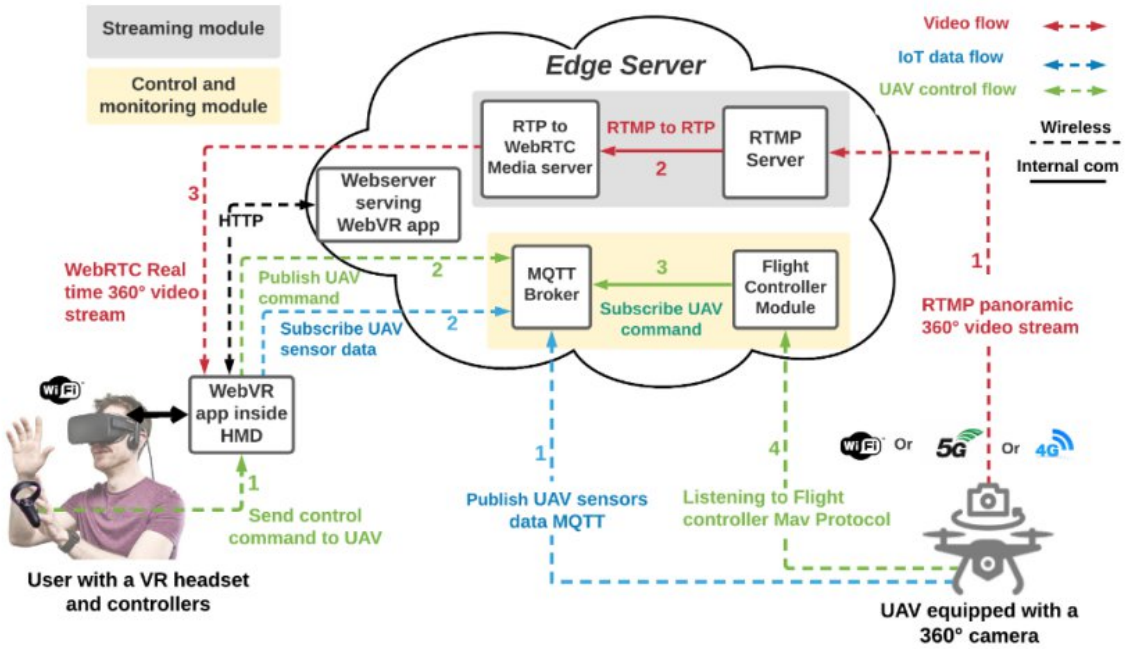


Figure 107: The high-level architecture of the system envisioned for VR-based remote control of UAVs [53].

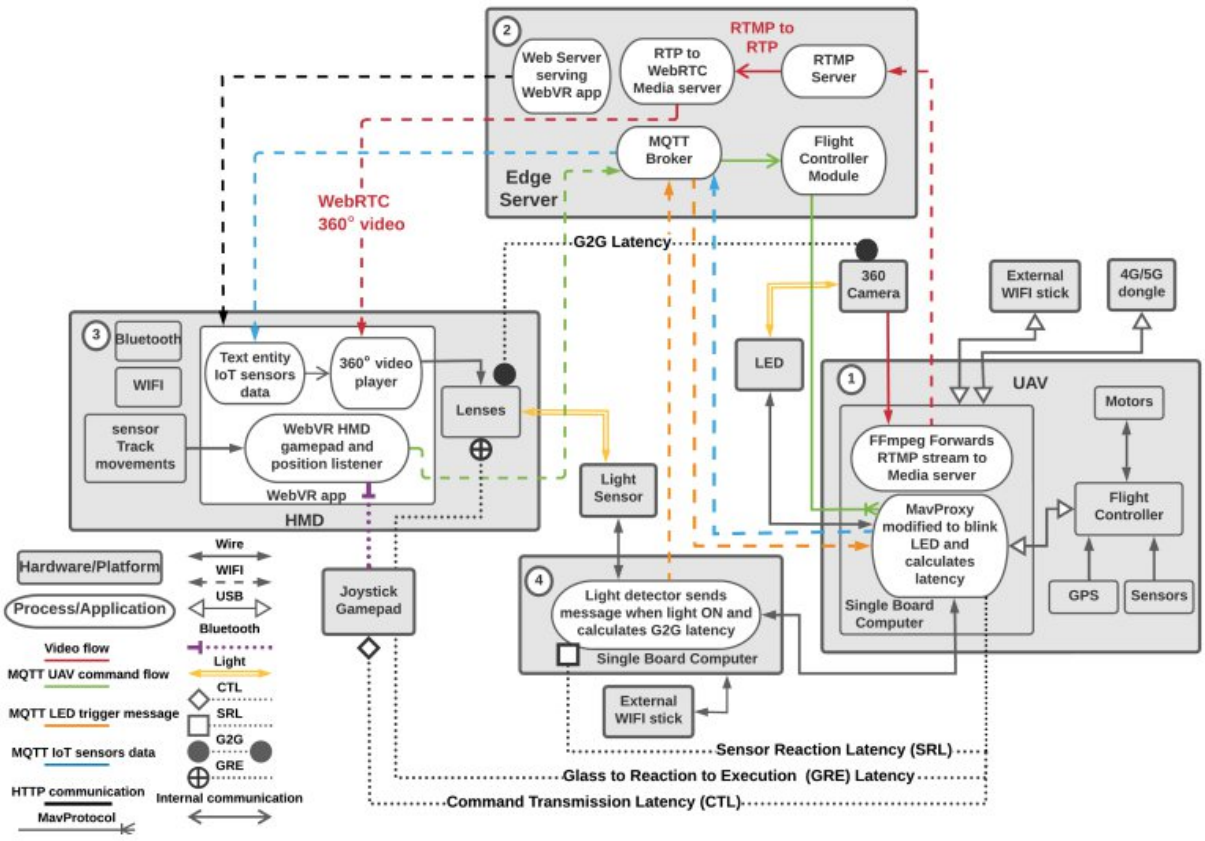


Figure 108: The hardware and software components of the system envisioned for VR-based remote control of UAVs and the measurement of the different considered delays [53].

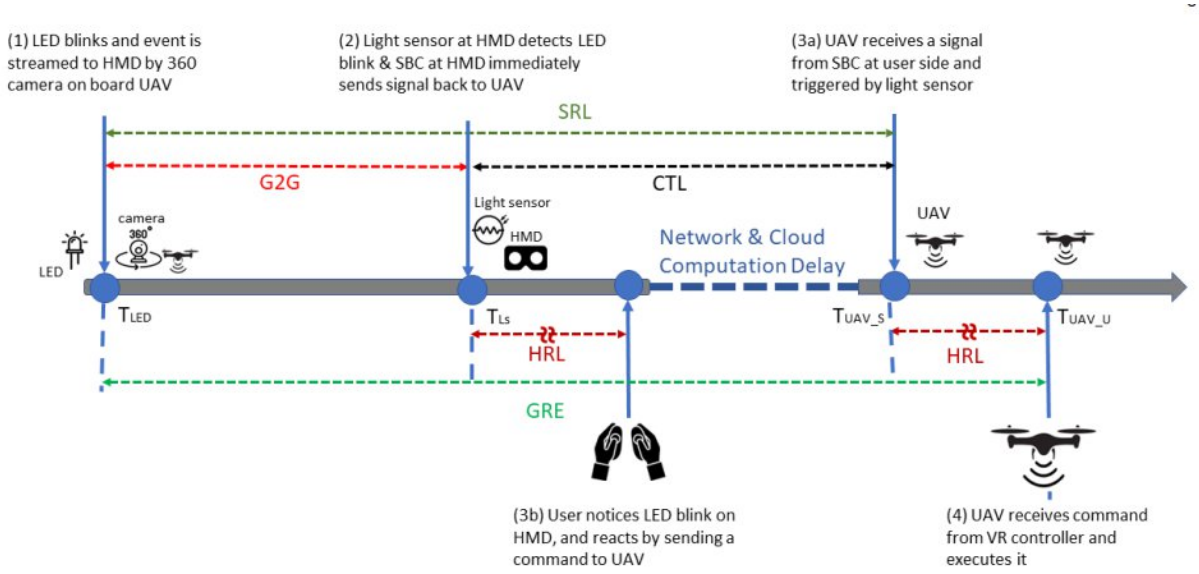


Figure 109: The different delays analyzed to evaluate the system envisioned for VR-based remote control of UAVs [53].

Hereunder presents an analysis of the results obtained through experiments on the system. Hereby, the HMD used WiFi for edge services (to remotely control the UAV), and the UAV connected to the edge server through WiFi, LTE, or 5G. The 360◦ camera streamed at 30 FPS using the H264 codec. Figure 110 depicts the GRE (Glass to Reaction to Execution) latency for varying streaming bit rates, access networks (WiFi, LTE, 5G), and two video qualities (HD or 4K). Two scenarios were considered: with and without human reaction latency. Figure 110 (a) displayed G2G latency, revealing an increase with higher Constant Bit Rate (CBR) encoding, particularly pronounced in 4G, especially for 4K videos. 5G demonstrated better results than 4G, nearly matching G2G latency with a dedicated WiFi connection. Figure 110 (b) and (c) presented GRE latency and Sensor Reaction Latency (SRL), respectively. Both increased with higher streaming bitrates, mirroring G2G latency trends. The average GRE was 900ms, representing end-to-end round-trip latency. Figure 110 (d) and (e) illustrated human reaction latency and command transmission latency. Human reaction latency converged to around 220ms, independent of network delays. Command transmission latency for 5G was similar to WiFi (103ms vs. 88ms), while 4G exhibited a higher average delay (138ms) due to network latency and bandwidth limitations compared to WiFi and 5G. Each graph's data points were averages from 40 experiment iterations by different individuals to minimize the impact of individual human reaction latency.

Furthermore, we utilized View-Port Peak Signal to Noise Ratio (VP-PSNR) and Video Multimethod Assessment Fusion (VMAF), a Full Reference metric by Netflix, for video quality evaluation (as shown in Figure 111). VMAF, employing machine learning, predicts subjective video quality on a 0 to 100 scale. VP-PSNR gauges encoding-induced distortion in video transmission. For this purpose, we followed the following steps:

- Recorded a 360° Equirectangular Projection (ERP) video at 4K (3840x1920) and 30FPS using a 360° camera.
- Generated a reference-view video using FFmpeg360 at a fixed orientation (pitch 0°, yaw 0°, roll 90°).
- Streamed the reference video over the internet, recorded it at the receiving HMD (0°, 0°, 90°), creating the user-view video.
- Compared visual quality using PSNR and VMAF filters on the reference-view and user-view videos.
- Repeated steps 3 and 4 for various streaming rates, including HD reference videos.

Figure 111 depicts the obtained results which can be summarized as follows:

- VP-PSNR and VMAF increased with higher streaming rates for both HD and 4K videos.
- 4K videos exhibited more distortion at low rates due to higher data volume.
- Overall, received streams had satisfactory quality, with the lowest VMAF values at 2Mbps being 40 and 50 for 4K and HD, respectively, and reaching 78 and 90 at 8Mbps.
- VP-PSNR values remained satisfactory at all considered streaming rates.

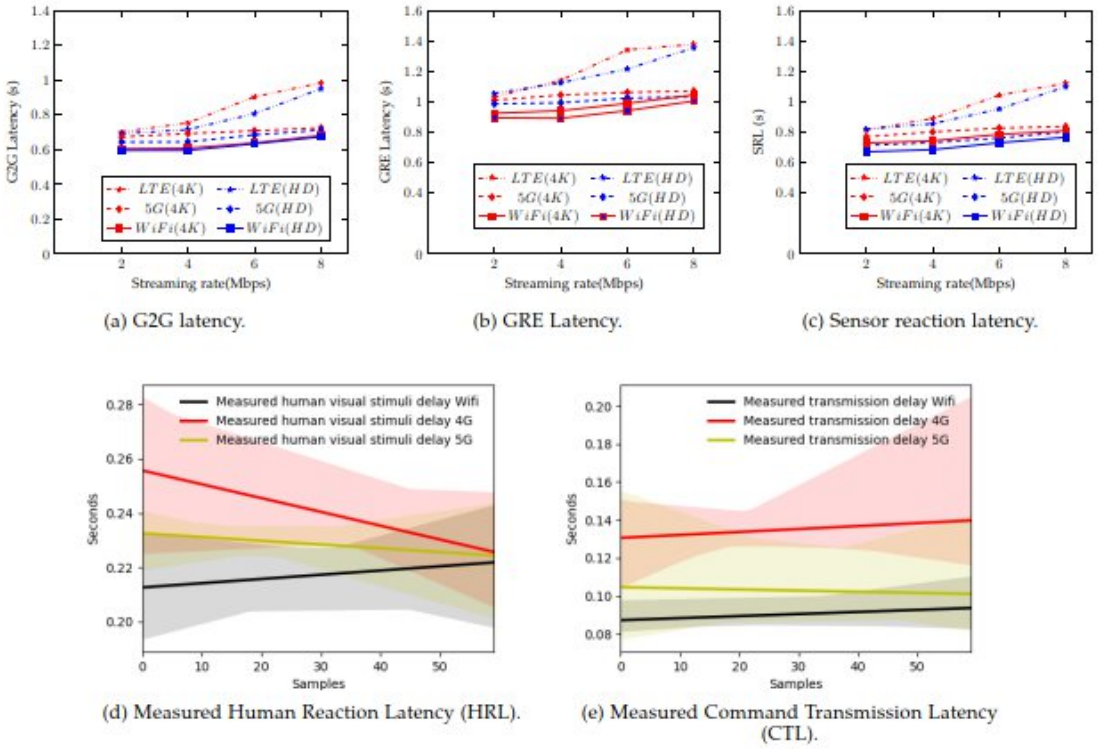


Figure 110: Measured latency [53].

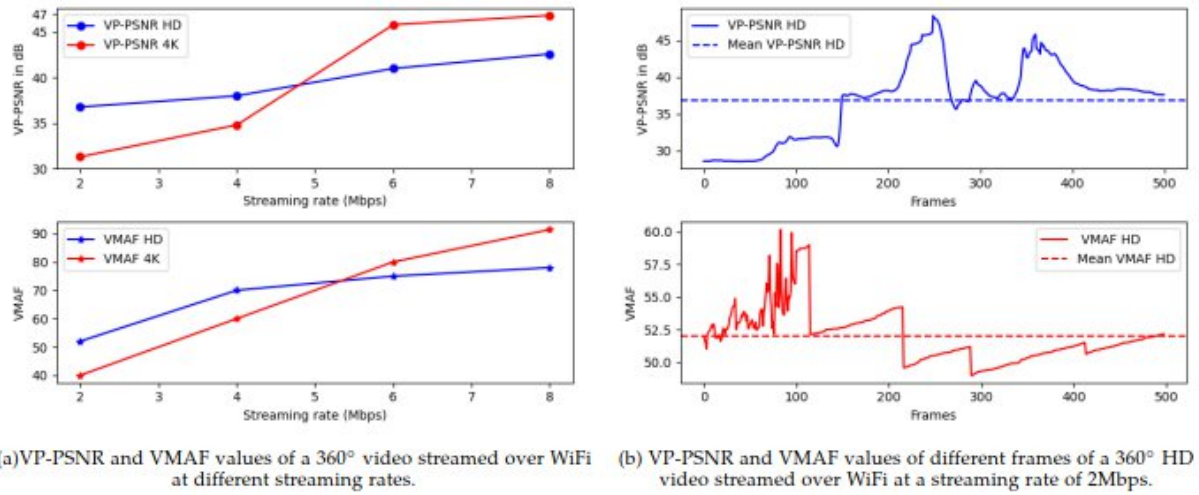


Figure 111: Video quality evaluation in terms of VP-PSNR and VMAF [53].

5.9 Mesh Merger

In UC3.1 Collaborative Gaming Application, the goal is to provide a highly immersive multiplayer augmented reality game prototype using ORBK's specialised multiplayer engine. The engine will be able to synchronise all game objects and user states across end devices, allowing for a seamless gaming experience.

The proposed solution will use a client-server architecture, where the client devices will be responsible for rendering the game and handling user input, while the server will manage the game state synchronisation and run the game system simulation.

Initially, UC3-1 Collaborative Gaming Application required *Mesh Collider Builder* service. Based on the point cloud data gathered on the mobile devices through RGB cameras, it is supposed to create a set of well-defined polygons to allow a clean and precise interaction with the environment. During the research process we found out that in many cases, 3D points reconstructed through RGB cameras cannot reach high quality to obtain such clean geometry (as shown in Figure 112).

Main issues when scanning using this method are:

- Noise/phantom data: point generated in random locations not connected to the environment features.
- No point generated at flat surfaces: flat surfaces was treated as empty space. This happens also for other featureless surfaces.
- Low precision of feature points localisation.

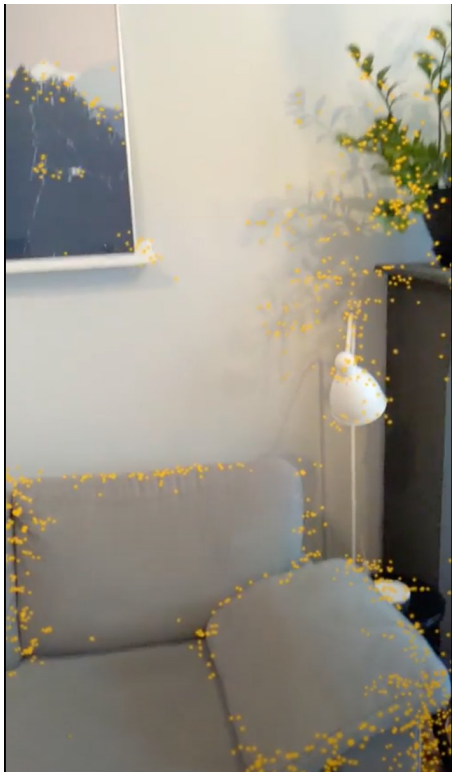


Figure 112: Environment scanning using RGB method on Android device.

To address the aforementioned challenges and to anticipate the growing prevalence of 3D sensors in future mobile devices, we are transitioning our focus to smart devices equipped with such technology. Currently, LiDAR sensors are available on select Apple devices, including iPhones Pro and iPads Pro. Consequently, we have chosen to conduct tests using LiDAR in conjunction with ARKit, a framework tailored for Apple devices.

The advantage of incorporating ARKit lies in its supplementary features, notably the Mesh Collider Builder (refer to Figure 93). Preliminary assessments of our reconstruction tests have yielded promising

results. The scanned data exhibits high quality, while the automatically generated mesh colliders by ARKit showcase commendable geometric attributes, including seamless continuity (absence of gaps) and simplicity characterised by a reduced triangle count (see Figure 113).

Figure 113: Environment scanning using LiDAR with instant mesh collider building.

Since ARKit provides proper mesh collider generation functionality, it has been decided to switch the focus on *merging* mesh colliders coming from different acquisition devices. The idea is to merge mesh colliders that are scanned and built in the same game session (and physical location) by the gamers. This functionality will significantly enrich the immersion of all participants of the game.

Each participant equipped with a smart device with LiDAR scans a fragment of the environment, ARKit builds a mesh collider from the scanned data and all mesh colliders are sent to the *Mesh Merger* service (see Figure 114) developed in the ambit of the Task 3.4. This service merges all mesh colliders into one common mesh collider.

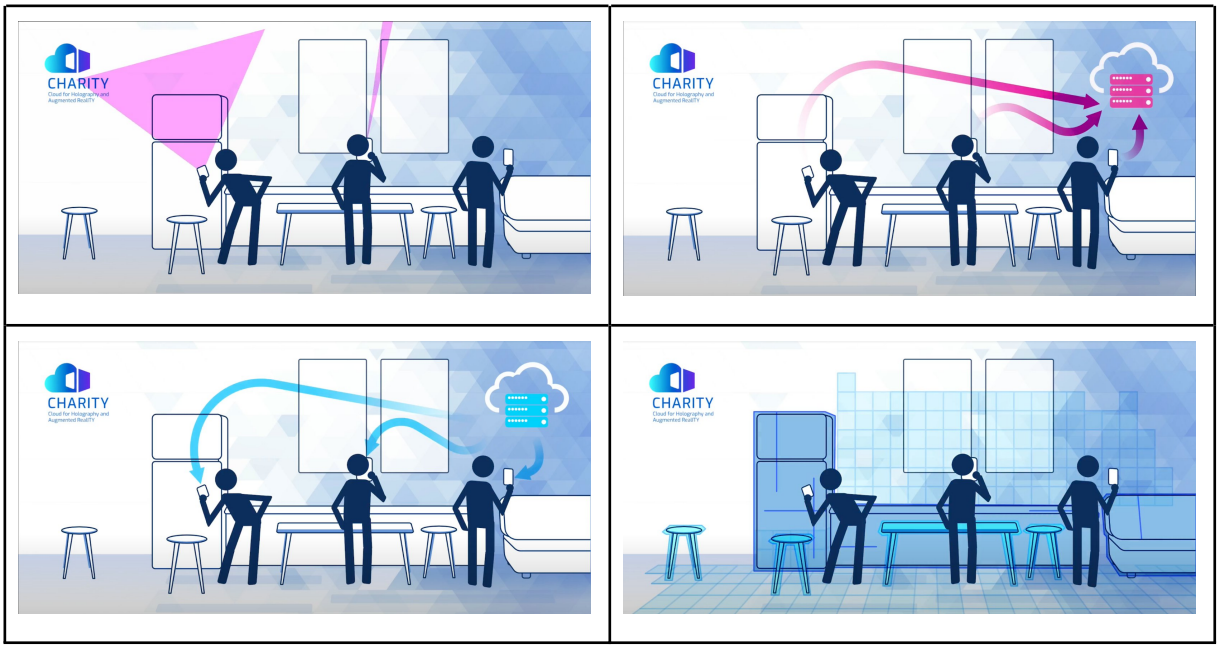


Figure 114: Merging mesh colliders.

5.9.1 Mesh Merger core

The core of the Mesh Merger relies on C++ based application that consist of two major components:

- **Mesh Alignment:** The individual mesh colliders are in their local frame of references. To build a complete environment, it is necessary to bring together those fragments into a common frame of reference (the World frame) based on a mesh alignment algorithm. For our implementation, we employ an open-source algorithm called "TEASER++"[50].This algorithm is specifically chosen due to its robustness and efficiency, making it well-suited for real-time applications. We have made adaptations and modifications to tailor it to our application. The average alignment time for a pair of scans varies between 0.65- 0.85ms. Therefore for an environment represented by three scans as the one shown in Figure 115, for example, the total time for alignment is approximately between 1.35-1.60s
- **Mesh Fusion:** Despite the robust accuracy of mesh alignment, registration artifacts can still arise due to data measurement errors. These artifacts can lead to misalignments between the individual meshes. To mitigate such errors, the Mesh Merger employs a surface conversion tool that converts the aligned meshes into a signed distance representation (SDF). The merged mesh is then extracted from this SDF representation. To ensure computational efficiency during the conversion process, the Mesh Merger utilizes the OpenVDB[51] open-source library. OpenVDB provides a framework for efficient voxel-based data structures and operations. By adapting OpenVDB and implementing it within the Mesh Merger, we can perform the conversion. with minimal computational overhead. On average, the entire mesh fusion process takes approximately 1.1-1.3 seconds, providing a quick and responsive experience. Additionally, the density of the output surface can be controlled based on the specific application requirements, further reducing the overall processing time. By leveraging OpenVDB and optimizing the implementation, the Mesh Merger achieves both accuracy and computational efficiency, enabling the creation of a merged mesh, i.e. the mesh collider of the environment, that is accurate while maintaining a fast-processing time. The result of align and merge the meshes of Figure 115 is shown in Figure 116.

Note that this prototype works with no prior information about the mesh to merge, this makes the tasks challenging. The current main limitation is a lack of robustness to align mesh with poor geometric features, i.e. planar surfaces. This problem can be alleviated by exploiting prior information, like an estimation of the relative position between the users' smartphone.

[50] https://github.com/MIT-SPARK/TEASER-plusplus

[51] https://www.openvdb.org/

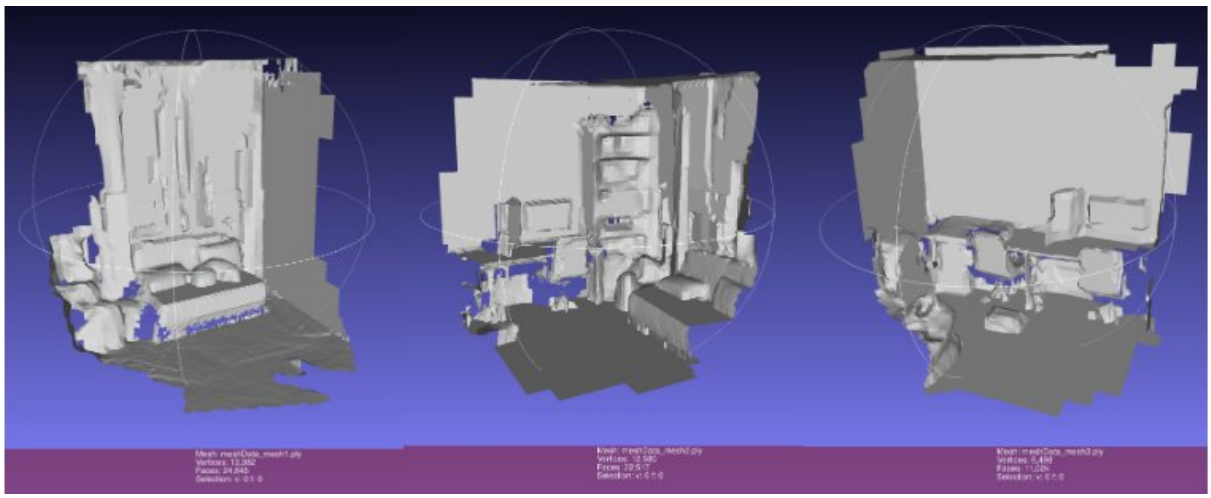


Figure 115: Individual meshes to align and fuse.

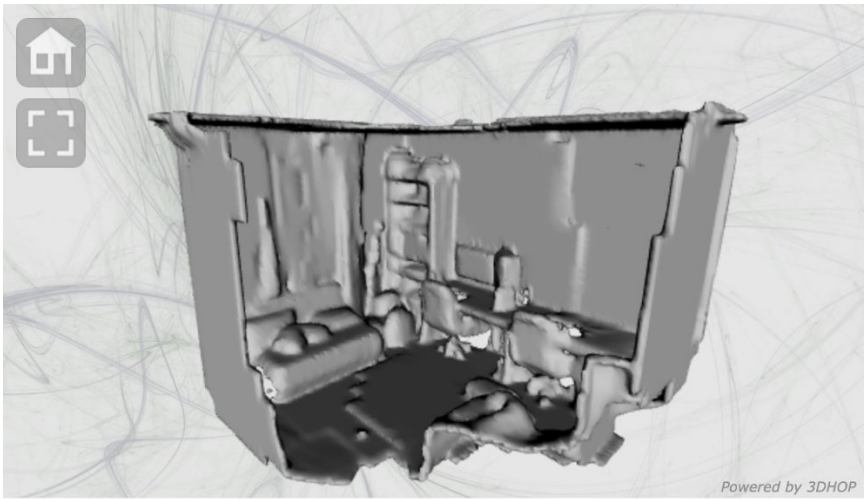


Figure 116: Final result.

5.9.2 Mesh Merger Service

The Mesh Merger component, developed by CNR, underwent extensive testing. ORBK designed a scanner application capable of transmitting fragments of scanned environmental meshes to the Mesh Merger. In the first stage of its development, the Mesh Merger was constructed as a web service (see Figure 117) to facilitate quick and straightforward verification of scanning and merging outcomes.

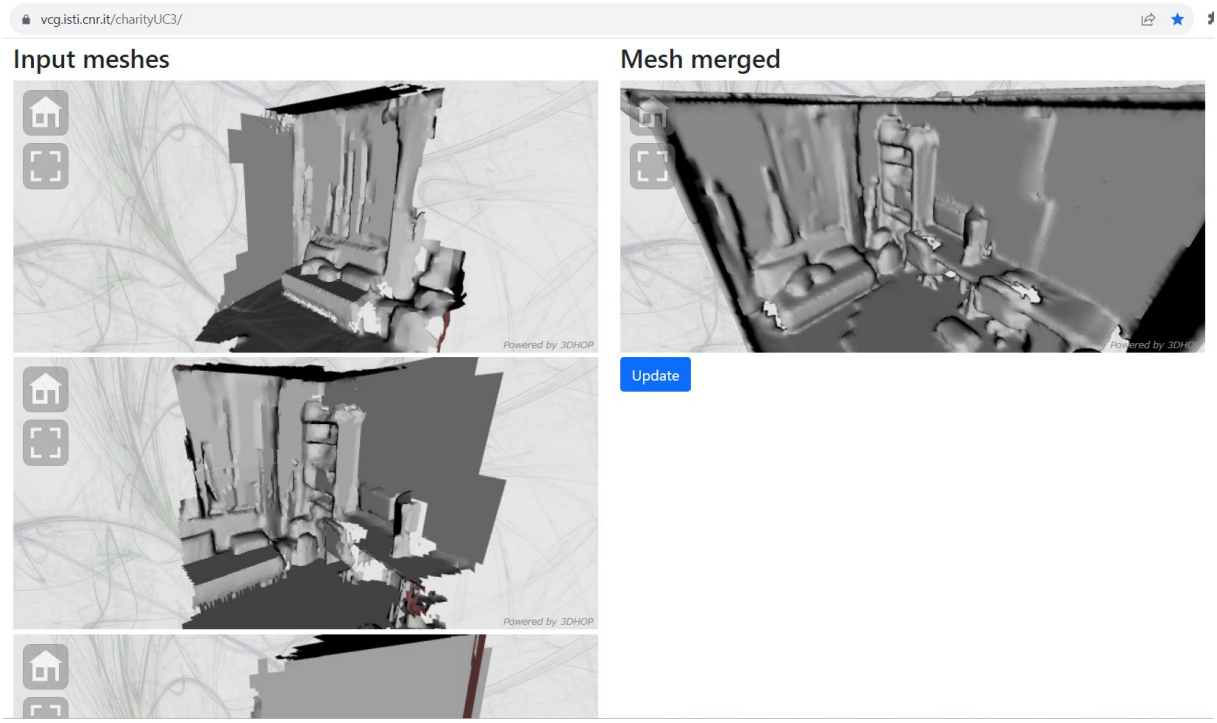


Figure 117: Mesh Merger testing website.

Now, the Mesh Merger Service is in the process of being enhanced with a REST API interface. The Game Server application will leverage this interface to transmit mesh fragments and retrieve fully merged meshes, streamlining the integration of merged meshes into the gaming environment (Figure 118).

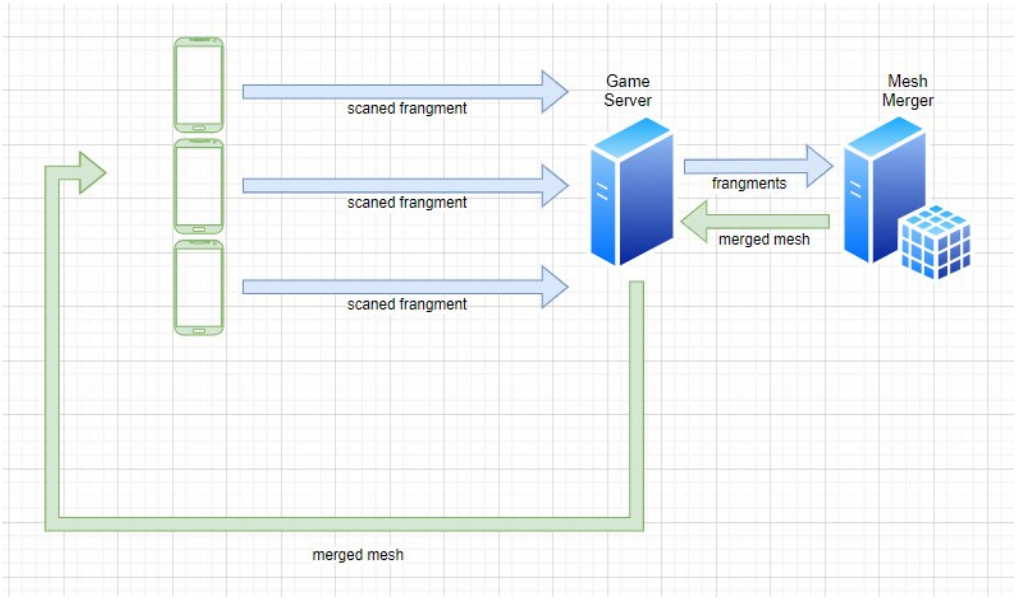


Figure 118: Game Server communicates with Mesh Merger Service via REST API.

In recent updates, a REST API paradigm has been implemented to establish communication between the Game Server application developed by ORBK and the Mesh Merger Service. The implementation is based on Node.js and follows REST-API principles of API calls. It is possible to request mesh alignment jobs and to query the status of a particular job providing its job id. The status of a job returns the URLs of the meshes being processed, the status of the algorithm, and the processing time. The actual data format for scans exchange is binary PLY which provides the advantage of having smaller size of the single colliders and so reduction in download time compared to JSON ASCII format used earlier. An

average download time for a single mesh collider using this format is approximately 0.80s. This reduces the overall latency and brings down the overall time to build the entire mesh collider.

At the moment of writing the Mesh Merger does not handle storage, so it retrieves the mesh colliders to align from another server, where the Game Server store them. To avoid this overhead in data streaming, in the following the Game Server could directly handle such storage, or it could ask to the users' devices to transmit the mesh colliders directly to the Mesh Merger, which cache them before the alignment requests. Before this, the next steps (according to the D4.2) are the finalization of the integration between the Game Server and the Mesh Merger, containerizing it and preparing the blueprint for the deployment by the Application Management Framework of the CHARITY platform.

6 Conclusions

The description of the research and development work conducted in the WP3 and the results reported in this Deliverable, demonstrate the big effort put in developing innovative solutions for XR applications, both from a scientific and a technological point of view. We do not only propose and design new technical solutions, but we also develop novel algorithms. Regarding the prototypes, even if in the past months some implementation activities are experiencing some delays, the first version of the prototypes have been implemented. Additionally, the integration work is in line with the one planned in the Deliverable D4.2. According to the just mentioned integration plan, the final version of the integrated prototypes are expected the first months of the next year.

References

[1] Baresi, Luciano, and Danilo Filgueira Mendonça. "Towards a serverless platform for edge computing." 2019 IEEE International Conference on Fog Computing (ICFC). IEEE, 2019.

[2] Gkoufas, Yiannis, and David Yu Yuan. "Dataset Lifecycle Framework and its applications in Bioinformatics." *arXiv e-prints* (2021): arXiv-2103.

[3] Koutsovasilis, Panos, et al. "A Holistic Approach to Data Access for Cloud-Native Analytics and Machine Learning." *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021.

[4] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. 2020. Exploring differences and commonalities between feature flags and configuration options. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (ICSE-SEIP '20). Association for Computing Machinery, New York, NY, USA, 233–242

[5] W. Li, Y. Lemieux, J. Gao, Z. Zhao and Y. Han, "Service Mesh: Challenges, State of the Art, and Future Research Opportunities," 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), 2019, pp. 122-1225.

[6] Svahnberg, M., Gurp, J., Bosch, J. On the Notion of Variability in Software Product Lines. Blekinge Institute of Technology Research Report No 02/01. (2001)

[7] Bashari, M., Bagheri, E., Weichang Du, W. Dynamic Software Product Line Engineering: A Reference Framework. International Journal of Software Engineering and Knowledge Engineering, Vol. 27, No. 2 (2017) 191–234

[8] Raatikainen, M., Tiihonen, J., Männistö, T. Software product lines and variability modeling: A tertiary study,J Systems and Software, Volume 149, Pages 485-510 (2019)

[9] Berger, T., Steghöfer, JP., Ziadi, T. et al. The state of adoption and the challenges of systematic variability management in industry. Empir Software Eng 25, 1755–1797 (2020).

[10] Reisner, E., Song, C., Ma, K., Foster, J., Porter, A. Using symbolic evaluation to understand behavior in configurable software systems. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (2010)

[11] Mendonca, N., Jamshidi, P., Garlan, D., Pahl, C., Developing Self-Adaptive Microservice Systems: Challenges and Directions. IEEE Software, vol. 38, no. 02, pp. 70-79, 2021.

[12] J. O. Kephart and D. M. Chess, "The vision of autonomic computing. *Computer*, vol. 36, no. 1, pp. 41-50, Jan. 2003

[13] J. Floch1 et al. Playing MUSIC - Building context-aware and self-adaptive mobile applications. *Software: Practice and Experience*. 43. 359-388. (2013)

[14] G. Alfrez, V. Pelechano, R. Mazo, C. Salinesi and D. Diaz, Dynamic adaptation of service compositions with variability models, *J. Syst. Softw*. 91 (2014) 24–47.

[15] R. Andrade, M. Ribeiro, H. Rebêlo, P. Borba, V. Gasiunas and L. Satabin, Assessing idioms for a flexible feature binding time, Comput. J. 59(1) (2015) 1–32.

[16] Google 2017. Draco: 3D data compression. https://google.github.io/draco/. Accessed: 2022-05-11.

[17] Max Limper, Stefan Wagner, Christian Stein, Yvonne Jung, and André Stork.2013. Fast Delivery of 3D Web Content: A Case Study. In Proceedings of the 18th International Conference on 3D Web Technology (San Sebastian, Spain) (Web3D '13). Association for Computing Machinery, New York, NY, USA, 11–17. https://doi.org/10.1145/2466533.2466536

[18] Federico Ponchio and Matteo Dellepiane. 2016. Multiresolution and fast decompression for optimal web-based rendering. Graphical Models 88 (2016), 1 – 11.

https://doi.org/10.1016/j.gmod.2016.09.002

[19] Markus Schütz. 2016. Potree: Rendering Large Point Clouds in Web Browsers. Ph. D. Dissertation.

[20] Tunstall, Brian Parker (September 1967). Synthesis of noiseless compression codes. Georgia Institute of Technology.

[21] H. Liu, H. Yuan, Q. Liu, J. Hou and J. Liu, "A Comprehensive Study and Comparison of Core Technologies for MPEG 3-D Point Cloud Compression," in IEEE Transactions on Broadcasting, vol. 66, no. 3, pp. 701-717, Sept. 2020, doi: 10.1109/TBC.2019.2957652.

[22] https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar, accessed 20-06-2022.

[23] Jeanette Ling, Rockwell Collins. Understanding Cloud-Based Visual System Architectures. Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC) 2018.

[24] Teemu Kämäräinen, Matti Siekkinen, Jukka Eerikäinen, and Antti Ylä-Jääski. CloudVR: Cloud Accelerated Interactive Mobile Virtual Reality. Proceedings of the 26th ACM international conference on Multimedia (MM '18). Association for Computing Machinery, New York, NY, USA, 1181–1189.

[25] Mark Claypool, Kajal Claypool. Latency and Player Actions in Online Games. Communications of the ACM, November 2006, Vol. 49 No. 11, Pages 40-45

[26] Waveren, J., The Asynchronous Time Warp for Virtual Reality on Consumer Hardware, 22nd ACM Conference on Virtual Reality Software and Technology, 2016

[27] Nicholson, N. “Exploring ‘Negative Latency’”, December 2019, https://nolannicholson.com/2019/12/16/exploring-negative-latency.html

[28] Makris A, Kontopoulos I, Psomakelis E, Xyalis SN, Theodoropoulos T, Tserpes K. Performance Analysis of Storage Systems in Edge Computing Infrastructures. Applied Sciences. 2022; 12(17):8923. https://doi.org/10.3390/app12178923

[29] Antonios Makris, Evangelos Psomakelis, Theodoros Theodoropoulos, and Konstantinos Tserpes. 2022. Towards a Distributed Storage Framework for Edge Computing Infrastructures. In Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge (FRAME '22). Association for Computing Machinery, New York, NY, USA, 9–14. https://doi.org/10.1145/3526059.3533617

[30] X.Hou,Y.Lu,andS.Dey,“WirelessVR/ARwithedge/cloudcomputing,” in Proc. Int. Conf. Comput. Commun. Netw., 2017, pp. 1–8.

[31] Sebastian Friston, Tobias Ritschel, and Anthony Steed. 2019. Perceptual rasterization for head-mounted display image synthesis. ACM Trans. Graph. 38, 4, Article 97 (August 2019), 14 pages. https://doi.org/10.1145/3306346.3323033

[32] L. Fink, N. Hensel, D. Markov-Vetter, C. Weber, O. Staadt and M. Stamminger, "Hybrid Mono-Stereo Rendering in Virtual Reality," *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, 2019, pp. 88-96, doi: 10.1109/VR.2019.8798283.

[33] Jie Guo, Xihao Fu, Liqiang Lin, Hengjun Ma, Yanwen Guo, Shiqiu Liu, and Ling-Qi Yan. 2021. ExtraNet: Real-time Extrapolated Rendering for Low-latency Temporal Super-sampling. ACM Trans. Graph. 40, 6, Article 278 (December 2021), 16 pages. https://doi.org/10.1145/3478513.3480531

[34] X. Hou and S. Dey, "Motion Prediction and Pre-Rendering at the Edge to Enable Ultra-Low Latency Mobile 6DoF Experiences," in *IEEE Open Journal of the Communications Society*, vol. 1, pp. 1674-1690, 2020, doi: 10.1109/OJCOMS.2020.3032608.

[35] K. Boos, D. Chu, and E. Cuervo, “Flashback: Immersive virtual reality on mobile devices via rendering memoization,” in *Proc. Int. Conf. Mobile Syst. Appl. Services*, 2016, pp. 291–304.

[36] L. Liu *et al.*, “Cutting the cord: Designing a high-quality untethered VR system with low latency

remote rendering," in *Proc. Int. Conf.* Mobile Syst. Appl. Services, 2018, pp. 68–80.

[37] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, and S. Savarese, "Social LSTM: Human trajectory prediction in crowded spaces," in Proc. Conf. Comput. Vis. Pattern Recognit., 2016, pp. 961–971.

[38] A. Gupta, J. Johnson, L. Fei-Fei, S. Savarese, and A. Alahi, "Social GAN: Socially acceptable trajectories with generative adversarial networks," in Proc. Conf. Comput. Vis. Pattern Recognit., 2018, pp. 2255–2264.

[39] J. Martinez, M. J. Black, and J. Romero, "On human motion prediction using recurrent neural networks," in *Proc. Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 4674-4683.

[40] J. Butepage, M. J. Black, D. Kragic, and H. Kjellstrom, "Deep representation learning for human motion prediction and classification," in Proc. Conf. Comput. Vis. Pattern Recognit., 2017, pp. 1591–1599.

[41] Dong, Chao, Chen Change Loy, and Xiaoou Tang. "Accelerating the super-resolution convolutional neural network." Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part II 14. Springer International Publishing, 2016.

[42] Lim, Bee, et al. "Enhanced deep residual networks for single image super-resolution." Proceedings of the IEEE conference on computer vision and pattern recognition workshops. 2017.

[43] Shi, Wenzhe, et al. "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[44] Lai, Wei-Sheng, et al. "Deep laplacian pyramid networks for fast and accurate super-resolution." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[45] Ledig, Christian, et al. "Photo-realistic single image super-resolution using a generative adversarial network." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[46] Wang, Xintao, et al. "Esrgan: Enhanced super-resolution generative adversarial networks." Proceedings of the European conference on computer vision (ECCV) workshops. 2018.

[47] Bias, Randy. "The History of Pets vs Cattle and How to Use the Analogy Properly." CloudScaling blog, September 2016.

[48] Makris, A., Psomakelis, E., Korontanis, I., Theodoropoulos, T., Protopsaltis, A., Pateraki, M., ... & Tserpes, K. (2023, October). "Streamlining XR Application Deployment with a Localized Docker Registry at the Edge". In European Conference on Service-Oriented and Cloud Computing (pp. 188-202). Cham: Springer Nature Switzerland.

[49] Psomakelis, E., Makris, A., Tserpes, K., & Pateraki, M. (2023). "A lightweight storage framework for edge computing infrastructures/EdgePersist". Software Impacts, 17, 100549.

[50] T. Taleb, Z. Nadir, H. Flinck, and J. Song, "Extremely-interactive and low latency services in 5G and beyond mobile systems," in IEEE Communications Standards Magazine, Vol. 5, No. 2, Jun. 2021, pp. 114-119.

[51] Z. Nadir, T. Taleb, H. Flinck, O. Bouachir, and M. Bagaa, "Immersive Services over 5G and Beyond Mobile Systems," in IEEE Network Magazine, Vol. 35, No. 6, Nov. 2021, pp. 299 – 306.

[52] O. El-Marai, T. Taleb, and J. Song, "AR-based Remote Command and Control Service: Self-driving Vehicles Use Case," in IEEE network magazine., Vol. 37, No. 3, Jun. 2023, pp. 170-177.

[53] T. Taleb, N. Sehad, Z. Nadir, and J. Song, "VR-based Immersive Service Management in B5G Mobile Systems: A UAV Command and Control Use Case," in IEEE IoT Journal, Vol. 10, No. 6, Mar.

2023, pp. 5349 – 5363.

[54] Mavridis, Ilias, and Helen Karatza. "Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing." Future Generation Computer Systems 94 (2019): 674-696.

[55] Lee, J., & Kim, Y. (2021, October). A design of MANO system for cloud native infrastructure. In 2021 International Conference on Information and Communication Technology Convergence (ICTC) (pp. 1336-1339). IEEE.

[56] Fornés-Leal, A., Lacalle, I., Vaño, R., Palau, C. E., Boronat, F., Ganzha, M., & Paprzycki, M. (2022). Evolution of MANO Towards the Cloud-Native Paradigm for the Edge Computing. In Advanced Computing and Intelligent Technologies: Proceedings of ICACIT 2022 (pp. 1-16). Singapore: Springer Nature Singapore.

[57] Mavridis, I., & Karatza, H. (2023). Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud. Concurrency and Computation: Practice and Experience, 35(11), e6365.

[58] Rejiba, Z., & Chamanara, J. (2022). Custom scheduling in Kubernetes: a survey on common problems and solution approaches. ACM Computing Surveys, 55(7), 1-37.

[59] Naranjo, D. M., Risco, S., de Alfonso, C., Pérez, A., Blanquer, I., & Moltó, G. (2020). Accelerated serverless computing based on GPU virtualization. Journal of Parallel and Distributed Computing, 139, 32-42.

[60] El Haj Ahmed, G., Gil-Castiñeira, F., & Costa-Montenegro, E. (2021). KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters. Software: Practice and Experience, 51(2), 213-234.

[61] T. Z. Benmerar et al., "Intelligent Multi-Domain Edge Orchestration for Highly Distributed Immersive Services: An Immersive Virtual Touring Use Case," 2023 IEEE International Conference on Edge Computing and Communications (EDGE), Chicago, IL, USA, 2023, pp. 381-392, doi: 10.1109/EDGE60047.2023.00061.

[end of document]