Estrategias y patrones en la codificación de vídeo para streaming en vivo sobre plataformas Cloud-Edge

Tesis Doctoral

Por:
Andoni Salcedo Navarro

Dirigida por:
Prof. Juan Gutiérrez Aguado
Prof. Miguel García Pineda

Programa de Doctorado en Tecnologías de la Información, Comunicaciones y Computación

Abril 2025

Índice general

Resumen ix

Resum xi

Abstract xiii

Agradecimientos xv

1. Introducción 1
- 1.1. Contexto y Trabajos Relacionados 5
- 1.2. Motivación de la tesis 9
- 1.3. Listado de trabajos resultantes 11
- 1.4. Estructura de la tesis 12

2. Herramientas Cloud-Native 15
- 2.1. K8sidecar 16
- 2.2. PodInsights 19
- 2.3. Integración de PodInsights y K8sidecar 21

3. Análisis de calidad 25
- 3.1. Codificación UHD CPU vs GPU 26

4. Arquitecturas GPU Cloud-Edge 31
- 4.1. Arquitectura Cloud-Native 32
- 4.2. Arquitectura Serverless 35
- 4.3. Arquitectura para vídeo 360VR 40

5. Conclusiones y trabajo futuro 45

6. K8sidecar: A Modular Kubernetes Chain of Sidecar Proxies for Microservices and Serverless Architectures **49**

7. UHD Video Encoding in CPU Versus GPU: Quality and Performance Trade-Offs **65**

8. Towards GPU-enabled serverless cloud edge platforms for accelerating HEVC video coding **81**

Índice de figuras

1.1. Modelos de servicios en la computación en la nube: comparación visual entre IaaS, PaaS, Serverless y SaaS, mostrando qué componentes gestiona el usuario y cuáles se entregan como servicio. 2

2.1. Arquitectura general de K8Sidecar. 18
2.2. Arquitectura para PodInsights en Kubernetes. PodInsights monitoriza un subconjunto de *pods* en cada nodo (basado en etiquetas). Las métricas recopiladas se almacenan en MongoDB. 21
2.3. Interacción entre el contenedor de codificación y el sidecar del *datamesh*, encargado de gestionar la subida y descarga de vídeos mediante un volumen compartido 22
2.4. Interacción entre el contenedor de codificación y el sidecar de monitorización GPU, que recolecta métricas con nvidia-smi y las envía a PodInsights sin afectar el rendimiento del proceso principal. 23
2.5. Interacción entre el contenedor de codificación y el sidecar de adaptación AMQP-HTTP, que traduce los mensajes provenientes de RabbitMQ en peticiones HTTP. 24

4.1. Arquitectura basada en Kubernetes para codificación de vídeo paralela 33
4.2. Un cluster con Kubernetes y Knative donde se despliégan las funciones. Un servidor de descarga con los videos, un servidor de carga para cargar los resultados y un registro de imágenes de contenedor están conectados al cluster. 36

4.3. Escenarios de réplicas sin GPU: a) máquinas virtuales *slim* con réplicas *slim*, b) máquinas virtuales *fat* con réplicas *slim*, y c) máquinas virtuales *fat* con réplicas *fat*. Escenarios con GPU: d) máquinas virtuales *slim* con réplicas *slim*, e) máquinas virtuales *fat* con réplicas *slim*, y f) máquinas virtuales *fat* con réplicas *fat*. 38
4.4. Arquitectura del sistema de la plataforma de transmisión y codificación de video en vivo 4K FoV de 360 grados Serverless. 41
4.5. Proceso de división espacial y temporal del streaming inicial. 41
4.6. Proceso de codificación en alta resolución (H) y en baja resolución (L) de cada tile. 42
4.7. Representación de como el Player elige la calidad de los *tiles* a representar según el FoV. 42

Índice de tablas

1.1. Ventajas y desventajas de las arquitecturas Serverless 4

3.1. Mejora de tiempos y factores entre codecs de la misma familia 28

Acrónimos

AMQP Advanced Message Queuing Protocol. 21, 24, 46

AV1 AOMedia Video 1. 6–8, 10, 25, 26, 28, 45, 46

AVC Advanced Video Coding. 6, 7, 25–27

BD Bjøntegaard Delta. 26, 27

CDN Content Delivery Network. 41, 42

CRD Custom Resource Definition. 16–18

CRF Constant Rate Factor. 8

DASH Dynamic Adaptive Streaming over HTTP. 8, 9, 43

FoV Field of View. 5, 7, 9, 32, 40–43, 46, 47

gRPC Google Remote Procedure Call. 6

HAS HTTP Adaptive Streaming. 1, 3, 29, 35

HEVC High Efficiency Video Coding. 6, 7, 10, 25, 26, 28, 32, 35, 37, 45

HTTP HyperText Transfer Protocol. 1, 6, 21, 24, 39

IaaS Infrastructure as a Service. 2, 6

MPD Media Presentation Description. 41

PaaS Platform as a Service. 2

PCI Peripheral Component Interconnect. 6, 35, 46

PSNR Peak Signal-to-Noise Ratio. 7, 8, 11, 26, 27, 45

QoE Quality of Experience. 1, 5, 29, 40

SaaS Software as a Service. 3

UHD Ultra High Definition. 5, 25, 26, 28

UVG Ultra Video Group. 26

VMAF Video Multi-Method Assessment Fusion. 7, 8, 11, 26, 27, 45

VoD Video on Demand. 1, 3

Resumen

A lo largo de esta tesis doctoral se aborda el reto de modernizar los sistemas de codificación de vídeo en entornos Cloud-Native y Cloud-Edge, combinando paradigmas de microservicios, arquitecturas Serverless y aceleración mediante GPU para lograr soluciones escalables, flexibles y de alta eficiencia en el procesamiento de contenido multimedia. El documento inicia describiendo cómo las arquitecturas de microservicios y los modelos Serverless han transformado la forma en que se despliegan y gestionan las aplicaciones, permitiendo una rápida integración de funcionalidades auxiliares mediante la inyección dinámica de sidecars y una monitorización de alta frecuencia. En este contexto se plantean herramientas innovadoras como K8Sidecar y PodInsights, que facilitan, respectivamente, la inyección dinámica de contenedores adicionales en pods de Kubernetes y la captura ultra precisa de métricas de CPU y memoria a nivel de milisegundos. Estas soluciones permiten no solo mejorar la agilidad en el despliegue y la gestión de recursos, sino también garantizar que la calidad y la eficiencia operativa se mantengan en entornos con cargas fluctuantes y requisitos de baja latencia.

La segunda contribución de la tesis se centra en un análisis comparativo entre implementaciones de codificación de vídeo en CPU y GPU, específicamente en el ámbito UHD (4K). Se evalúan tres familias de códecs (H.264/AVC, H.265/HEVC y AV1) utilizando tanto implementaciones tradicionales basadas en CPU (*libx264*, *libx265* y *librav1e*) como versiones aceleradas por GPU (*h264_nvenc*, *hevc_nvenc* y *av1_nvenc*). La metodología experimental, realizada sobre un conjunto de 15 vídeos UHD del dataset UVG, se fundamenta en métricas de calidad (PSNR y VMAF) y en el análisis de diferencias de bitrate mediante la métrica Bjøntegaard Delta, junto con la medición de tiempos de codificación por frame. Los resultados revelan que la aceleración mediante GPU reduce significativamente

los tiempos de procesamiento; por ejemplo, en el caso de AV1 se observó una mejora en el tiempo de codificación de más de 68 veces. En el caso de H.264/AVC se alcanzó una reducción promedio en el bitrate de alrededor del 20 % para obtener calidades equivalentes, y en HEVC, aunque la versión CPU mostró una ligera ventaja en bitrate, la aceleración GPU ofreció una reducción del tiempo de codificación de más del doble, lo que resulta decisivo en aplicaciones de live streaming y videoconferencias.

La tercera aportación se orienta hacia el desarrollo de arquitecturas avanzadas que integran la aceleración GPU en entornos Cloud-Native y Cloud-Edge para optimizar la codificación de vídeo. En primer lugar, se presenta un framework event-driven basado en Knative y Kubernetes, que orquesta funciones de codificación HEVC activadas mediante CloudEvents. La integración de GPUs, utilizando el códec *hevc_nvenc*, permite una reducción significativa en el tiempo de codificación en comparación con la versión basada en CPU, lo que se traduce en mejoras notables en escenarios de escalado dinámico y distribución de carga.

Además, se explora la codificación paralela mediante la división de tareas en múltiples pods dentro de un clúster Kubernetes, orquestados a través del broker de mensajes RabbitMQ. Esta estrategia posibilita el procesamiento concurrente de segmentos de vídeo, reduciendo drásticamente los tiempos de codificación y optimizando el uso de recursos, lo que se refleja en una mejora sustancial en la eficiencia operativa del sistema.

Por último, se aborda el desafío del live streaming inmersivo en 360 grados mediante un proceso que combina segmentación temporal y espacial (tiling) adaptativo basado en el Field-of-View del usuario. Esta arquitectura híbrida, que emplea técnicas aceleradas por GPU junto con orquestación Serverless, permite mantener los tiempos de procesamiento por debajo de 10 segundos (límite para considerarse live) y optimiza el consumo de ancho de banda, ajustando la calidad del stream según la zona de interés del espectador.

Resum

Al llarg d'aquesta tesi doctoral s'aborda el repte de modernitzar els sistemes de codificació de vídeo en entorns Cloud-Native i Cloud-Edge, combinant paradigmes de microserveis, arquitectures Serverless i acceleració mitjançant GPU per a aconseguir solucions escalables, flexibles i d'alta eficiència en el processament de contingut multimèdia. El document s'inicia descrivint com les arquitectures de microserveis i els models Serverless han transformat la forma en què es despleguen i gestionen les aplicacions, cosa que permet una ràpida integració de funcionalitats auxiliars mitjançant la injecció dinàmica de sidecars i un monitoratge d'alta freqüència. En aquest context es plantegen eines innovadores com K8Sidecar i PodInsights, que faciliten, respectivament, la injecció dinàmica de contenidors addicionals en pods de Kubernetes i la captura ultra precisa de mètriques de CPU i memòria a escala de mil·lisegons. Aquestes solucions permeten no sols millorar l'agilitat en el desplegament i la gestió de recursos, sinó també garantir que la qualitat i l'eficiència operativa es mantinguen en entorns amb càrregues fluctuants i requisits de baixa latència.

La segona contribució de la tesi se centra en una anàlisi comparativa entre implementacions de codificació de vídeo en CPU i GPU, específicament en l'àmbit UHD (4K). S'avaluen tres famílies de còdecs (H.264/AVC, H.265/HEVC i AV1) utilitzant tant implementacions tradicionals basades en CPU (*libx264*, *libx265* i *librav1e*) com a versions accelerades per GPU (*h264_nvenc*, *hevc_nvenc* i *av1_nvenc*). La metodologia experimental, realitzada sobre un conjunt de 15 vídeos UHD del conjunt de dades UVG, es fonamenta en mètriques de qualitat (PSNR i VMAF) i en l'anàlisi de diferències de bitrate mitjançant la mètrica Bjøntegaard Delta, juntament amb el mesurament de temps de codificació per frame. Els resultats revelen que l'acceleració mitjançant GPU redueix significativament els

temps de processament; per exemple, en el cas d'AV1 es va observar una millora en el temps de codificació de més de 68 vegades. En el cas d'H.264/AVC es va aconseguir una reducció mitjana en el bitrate del voltant del 20 % per a obtindre qualitats equivalents, i en HEVC, encara que la versió CPU va mostrar un lleuger avantatge en bitrate, l'acceleració GPU va oferir una reducció del temps de codificació de més del doble, la qual cosa resulta decisiu en aplicacions de live streaming i videoconferències.

La tercera aportació s'orienta cap al desenvolupament d'arquitectures avançades que integren l'acceleració GPU en entorns Cloud-Native i Cloud-Edge per a optimitzar la codificació de vídeo. En primer lloc, es presenta un framework event-driven basat en Knative i Kubernetes, que orquestra funcions de codificació HEVC activades mitjançant CloudEvents. La integració de GPUs, utilitzant el còdec *hevc_nvenc*, permet una reducció significativa en el temps de codificació en comparació amb la versió basada en CPU, la qual cosa es tradueix en millores notables en escenaris d'escalat dinàmic i distribució de càrrega.

A més, s'explora la codificació paral·lela mitjançant la divisió de tasques en múltiples pods dins d'un clúster Kubernetes, orquestrats a través del broker de missatges RabbitMQ. Aquesta estratègia possibilita el processament concurrent de segments de vídeo, reduint dràsticament els temps de codificació i optimitzant l'ús de recursos, la qual cosa es reflecteix en una millora substancial en l'eficiència operativa del sistema.

Finalment, s'aborda el desafiament del streaming immersiu en 360 graus mitjançant un procés que combina segmentació temporal i espacial (tiling) adaptatiu basat en el Field-of-View de l'usuari. Aquesta arquitectura híbrida, que empra tècniques accelerades per GPU juntament amb orquestració Serverless, permet mantindre els temps de processament per davall de 10 segons (límit per a considerar-se live) i optimitza el consum d'amplada de banda, ajustant la qualitat del stream segons la zona d'interés de l'espectador.

Abstract

Throughout this doctoral thesis, the challenge of modernizing video encoding systems in Cloud-Native and Cloud-Edge environments is addressed by combining microservices paradigms, Serverless architectures, and GPU acceleration to achieve scalable, flexible, and highly efficient solutions for multimedia content processing. The document begins by describing how microservices architectures and Serverless models have transformed the way applications are deployed and managed, enabling the rapid integration of auxiliary functionalities through the dynamic injection of sidecars and high-frequency monitoring. In this context, innovative tools such as K8Sidecar and PodInsights are proposed, which respectively facilitate the dynamic injection of additional containers into Kubernetes pods and the ultra-precise capture of CPU and memory metrics at the millisecond level. These solutions not only improve deployment agility and resource management but also ensure that quality and operational efficiency are maintained in environments with fluctuating loads and low-latency requirements.

The second contribution of the thesis focuses on a comparative analysis between video encoding implementations on CPU and GPU, specifically in the UHD (4K) domain. Three codec families (H.264/AVC, H.265/HEVC, and AV1) are evaluated using both traditional CPU-based implementations (*libx264*, *libx265*, and *librav1e*) and GPU-accelerated versions (*h264_nvenc*, *hevc_nvenc*, and *av1_nvenc*). The experimental methodology, carried out on a set of 15 UHD videos from the UVG dataset, is based on quality metrics (PSNR and VMAF) and the analysis of bitrate differences using the Bjøntegaard Delta metric, along with the measurement of encoding times per frame. The results reveal that GPU acceleration significantly reduces processing times; for example, in the case of AV1, an improvement in encoding time of more than 68 times was ob-

served. In the case of H.264/AVC, an average bitrate reduction of around 20 % was achieved to achieve equivalent qualities. In HEVC, although the CPU version showed a slight advantage in bitrate, GPU acceleration more than doubled the encoding time, which is crucial for live streaming and video conferencing applications.

The third contribution focuses on the development of advanced architectures that integrate GPU acceleration in cloud-native and cloud-edge environments to optimize video encoding. First, an event-driven framework based on Knative and Kubernetes is presented, which orchestrates HEVC encoding functions triggered by CloudEvents. The integration of GPUs, using the *hevc_nvenc* codec, enables a significant reduction in encoding time compared to the CPU-based version, resulting in notable improvements in dynamic scaling and load distribution scenarios.

Additionally, parallel encoding is explored through task division across multiple pods within a Kubernetes cluster, orchestrated via the RabbitMQ message broker. This strategy allows for the concurrent processing of video segments, drastically reducing encoding times and optimizing resource usage, resulting in a substantial improvement in the system's operational efficiency.

Finally, the challenge of immersive 360-degree live streaming is addressed through a process that combines temporal and spatial (tiling) segmentation adapted to the user's Field-of-View. This hybrid architecture, employing GPU-accelerated techniques along with Serverless orchestration, maintains processing times below 10 seconds (the threshold for being considered live) and optimizes bandwidth consumption by adjusting stream quality based on the viewer's area of interest.

Agradecimientos

En primer lugar, quiero expresar mi más profundo agradecimiento a mi familia, y en especial a mi madre, quien desde el principio creyó en mí y me impulsó a emprender esta aventura del doctorado. Su apoyo incondicional, sus palabras de ánimo en mis momentos más bajos y su constante esfuerzo por entender cada detalle de mi investigación, aunque a veces los términos técnicos fueran difíciles, han sido fundamentales para llegar hasta aquí.

A mis directores, Juan y Miguel, gracias infinitas por haber confiado en mí desde el primer día. Recuerdo perfectamente cuando me propusieron esta oportunidad que jamás hubiera imaginado y cómo, desde entonces, han sido mis guías, apoyándome, motivándome y aconsejándome en cada etapa de este recorrido.

A mis amigos, gracias por estar siempre ahí, por esos momentos de desconexión y risas, que me ayudaron a mantener el equilibrio necesario en esta intensa etapa.

Quiero agradecer especialmente a mis compañeros de trabajo y de comidas, que se convirtieron en mis compañeros de viaje en esta maratón académica. A los miembros del equipo Commlab, que me recibieron con los brazos abiertos cuando más lo necesitaba, y a Guillem, cuyo apoyo ha sido clave. Gracias por compartir conmigo tanto los momentos complicados como los felices, convirtiendo estos años en una experiencia inolvidable.

Por ultimo, quiero agradecer a los organismos que han hecho posible este trabajo gracias a su financiación (proyectos PID2021-126209OB-I00 y TED2021-131040B-C33).

Hoy, al finalizar esta etapa, me invade una mezcla de alegría, orgullo y también nostalgia. Lo que comenzó como un reto personal solitario se convirtió en un camino lleno de buena compañía y aprendizajes compartidos. Sin todos vosotros, nunca habría llegado hasta aquí con la misma ilusión y fuerza.

Gracias por acompañarme en uno de los capítulos más importantes de mi vida.

Capítulo 1

Introducción

Hoy en día, alrededor del 68 % de la población mundial tiene acceso a Internet, lo que equivale a unos 5.56 mil millones de usuarios [1]. Más de la mitad, aproximadamente un 52 %, indica que utiliza Internet principalmente para ver vídeos, programas de televisión y películas [2]. Dedican en promedio más de 3 horas al día a consumir contenido de vídeo. Por esta razón, YouTube se posiciona como el segundo sitio web más accedido a nivel mundial, solo superado por Google [3]. Además, los usuarios de Internet gastan más de 100 mil millones de dólares al año en plataformas de VoD (Video on Demand) como pueden ser Netflix, Amazon Prime Video, y Disney+, entre otras [4].

Todas estas plataformas transmiten sus vídeos a través de HTTP para ofrecer alta compatibilidad y acceso independiente del dispositivo. Más específicamente, la tecnología más utilizada es el HAS (HTTP Adaptive Streaming). En HAS, un vídeo se divide en fragmentos temporales más pequeños (o segmentos) y cada fragmento se codifica a varias resoluciones y tasas de bits, creando así múltiples representaciones de este modo, el reproductor puede solicitar la representación de cada segmento temporal en función de condiciones cambiantes (ancho de banda, tamaño de buffer disponible, etc). Estos sistemas de transmisión requieren sistemas de codificación de vídeo eficientes para ofrecer contenido en la mayor cantidad de resoluciones, tasas de bits y códecs posibles, a fin de brindar la mejor QoE (Quality of Experience) a los usuarios finales [5].

Estos procesos de codificación requieren sistemas distribuidos para reducir los tiempos de codificación [6]. Actualmente, estas tareas

se realizan en entornos de Cloud, ya que ofrecen el dinamismo que exigen las plataformas de streaming. Las soluciones de codificación de vídeo basadas en Cloud han evolucionado a medida que han surgido nuevos paradigmas. Existen diversos servicios Cloud, todos con el objetivo de ofrecer funcionalidades específicas adecuadas para un propósito. Específicamente, se trata de modelos de implementación de servicios que permiten al usuario elegir el nivel de control sobre la información y los servicios que se proporcionarán (ver Figura 1.1).

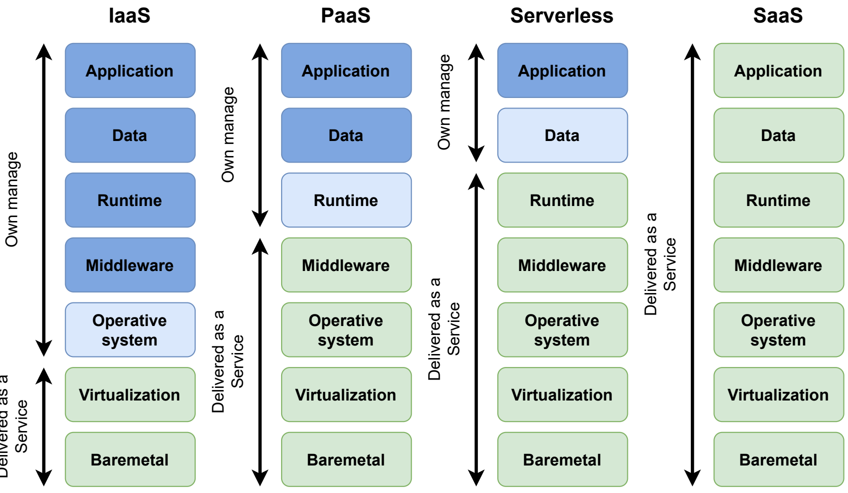


Figura 1.1: Modelos de servicios en la computación en la nube: comparación visual entre IaaS, PaaS, Serverless y SaaS, mostrando qué componentes gestiona el usuario y cuáles se entregan como servicio.

Los servicios más importantes ofrecidos por los proveedores de Cloud son:

- **IaaS (Infrastructure as a Service)**: En este modelo, el proveedor ofrece la infraestructura y el cliente tiene casi control total sobre ella. El proveedor es responsable de garantizar la fiabilidad y seguridad a largo plazo mediante el mantenimiento de la infraestructura.

- **PaaS (Platform as a Service)**: El proveedor suministra el sistema de hardware y una plataforma de software de aplicaciones. PaaS permite crear, ejecutar y gestionar sus aplicaciones

sin tener que diseñar y mantener la infraestructura y la plataforma.

- **Serverless**: Este tipo de arquitectura permite la ejecución de aplicaciones mediante contenedores efímeros. Estos contenedores se ejecutan en función de la demanda, por lo no hay que preocuparse por la gestión de la infraestructura en la que se ejecuta, concentrándose exclusivamente en la funcionalidad.
- **SaaS (Software as a Service)**: Este modelo ofrece una aplicación específica que se puede invocar a través de librerías de inmediato, sin necesidad de instalar, desplegar o mantener nada. Todo es gestionado por el proveedor del servicio.

La Tabla 1.1 recoge las principales ventajas e inconvenientes de las arquitecturas Serverless en comparación con otros paradigmas de Cloud.

De acuerdo con las características de las arquitecturas Serverless, estas han evolucionado para convertirse en una herramienta muy útil para desplegar funcionalidades en un entorno de producción, pagando solo cuando se ejecutan, sin necesidad de invertir en infraestructura [7]. Los sistemas de codificación de vídeo basados en HAS para VoD podrían ser un caso de uso adecuado para desplegarse en una arquitectura Serverless. En este tipo de escenario, el vídeo original se divide en varios fragmentos de 2 a 4 segundos, y cada fragmento generalmente se codifica para diferentes tasas de bits y/o resoluciones. Para cada una de estas codificaciones o conjuntos de codificaciones por fragmento, la nube podría ejecutar un fragmento de código encapsulado en una función, realizando una asignación dinámica de recursos. La plataforma Serverless sería capaz de ejecutar en paralelo la codificación de varios fragmentos, ya que este modelo permitiría que el número de réplicas que ejecutan la función se escale automáticamente en caso de mayores requerimientos, lo que podría reducir el tiempo total de codificación.

Además, el paradigma Cloud-Edge ha emergido como complemento de la computación en la nube [8, 9]. Al integrar nodos de procesamiento en el borde de la red, se reducen las latencias y se optimiza la distribución de cargas, permitiendo que las tareas de codificación y transmisión se acerquen al usuario final.

Ventajas	**Desventajas**
Funciones cercanas: La aplicación no se aloja en un servidor de origen. Las funciones se despliegan en la nube cuando se requieren, siendo el proveedor del servicio quien las ubica lo más cerca posible del usuario para reducir la latencia.	Pérdida de control sobre la infraestructura: El proveedor del servicio en la nube controla la infraestructura subyacente, por lo que no se podrá personalizar ni optimizar para satisfacer necesidades específicas.
Proceso de escalado: Generalmente, el rendimiento y la capacidad de memoria requieren un proceso de escalado cuando se necesitan más recursos. Con la tecnología Serverless, este proceso se puede realizar de forma automática.	Pruebas y depuración exigentes: La depuración es más complicada, ya que los desarrolladores no tienen visibilidad sobre los procesos del backend, y la aplicación se divide en funciones pequeñas y separadas.
Aplicación como suma de funciones: Esto permite a los usuarios actualizar, parchear, corregir o añadir nuevas características a partes específicas de la aplicación, sin necesidad de modificarla en su totalidad, ya que cada función se puede actualizar de manera independiente.	No diseñado para procesos a largo plazo: Los proveedores cobran por el tiempo de ejecución del código; puede resultar más costoso ejecutar una aplicación con procesos de larga duración en una infraestructura Serverless que en una tradicional.
Automatización: El proceso no requiere la implementación de protocolos de contingencia, ya que la tecnología ofrece tolerancia a errores en la aplicación.	Problemas de seguridad: Los desarrolladores de funciones deben cuidar los datos críticos de la empresa, evitando fugas de secretos, ataques de denegación de servicio, etc.
Delegar la administración del servidor: Los tiempos de ejecución del programa o servicio se definen automáticamente, sin requerir verificaciones constantes o instalación de plugins.	
Ahorro: Se paga únicamente por el tiempo en que la función está en ejecución.	

Tabla 1.1: Ventajas y desventajas de las arquitecturas Serverless

Dichas ventajas en escalabilidad y asignación dinámica se trasladan de forma natural a nuevos modelos de transmisión, como el vídeo en 360 grados (360VR). Este tipo de streaming está revolucionando el panorama digital al brindar experiencias inmersivas que superan la visualización convencional [10]. A diferencia de los vídeos convencionales, permite visualizar en todas las direcciones, aumentando el nivel de interacción y *engagement*. Sin embargo, ofrecer vídeo en 360 grados de alta calidad plantea desafíos técnicos debido al elevado consumo de ancho de banda y a la gran potencia computacional necesaria para codificar contenido en UHD (Ultra High Definition) [11].

El *FoV (Field of View) adaptive streaming* ofrece una solución al transmitir en alta calidad solo el área que se está visualizando, y reducir la calidad en las zonas periféricas no vistas [12]. Esto permite mantener una buena QoE durante la visualización, al tiempo que se reduce el consumo de ancho de banda, haciendo viable la transmisión de vídeo en 360 grados a resoluciones elevadas. Aunque existen técnicas de streaming adaptativo como el cambio dinámico de resolución o hacer una división espacial del vídeo, estas a menudo resultan insuficientes para vídeo inmersivo debido a los patrones de visualización impredecibles. La integración de mecanismos adaptativos basados en FoV, específicamente diseñados para entornos inmersivos, es crucial para avanzar en el campo del streaming de vídeo inmersivo [13]. Asimismo, la integración de sistemas Serverless para la codificación de vídeo 360 se muestra como una alternativa prometedora, ya que permite una escalabilidad automática y una gestión dinámica de recursos, adaptándose eficientemente a las variaciones en la demanda.

1.1. Contexto y Trabajos Relacionados

A lo largo de la última década se han propuesto y desarrollado diversas soluciones para abordar los crecientes retos en la codificación y transcodificación de vídeo en entornos Cloud, Serverless y de microservicios, integrando tecnologías de aceleración por GPU, contenedores y sidecar proxies, sin dejar de lado una rigurosa evaluación basada en métricas de calidad.

Por un lado, las soluciones basadas en la nube han evolucionado para adaptarse a los requerimientos de instanciación dinámica de en-

coders, fundamentales en aplicaciones de streaming adaptativo y de alta demanda. Gutiérrez-Aguado et al. [14] propusieron una arquitectura distribuida sobre el modelo de IaaS para la codificación de vídeo (HEVC, VP9 y AV1) basada en un *pool* elástico de *workers* que garantiza alta disponibilidad y tolerancia a fallos. Paralelamente, se ha avanzado en la integración de GPUs en entornos Cloud, como demuestran Yang et al. [15] y Gutiérrez-Aguado et al.[16], quienes utilizan tecnologías como el *PCI pass-through* para lograr un rendimiento similar al nativo. Esta aceleración ha permitido mejorar la codificación en estándares como H.264/AVC [17], H.265/HEVC [18] y VP9 [19], y se ha explorado su aplicación en entornos Cloud mediante *Network Function Virtualization*, como propone Comi et al.[20]. Sin embargo, las soluciones tradicionales basadas en máquinas virtuales presentan tiempos de despliegue relativamente altos, lo que ha impulsado la adopción de contenedores Linux [21] y sistemas de orquestación como Kubernetes [22]. Además, la creciente popularidad de la computación Serverless impulsada por modelos de gestión dinámica de recursos [23, 24] ha dado lugar a frameworks *open-source* como OpenFaaS [25], Knative [26], Apache OpenWhisk [27] e IronFunctions [28]. Ejemplos de sistemas de codificación de vídeo basados en el paradigma Serverless incluyen el encoder para VP8 orientado al paralelismo de Fouladi et al. [29], el framework Mu [30] y el framework Sprocket [31].

Por otro lado, la evolución de las arquitecturas de microservicios ha llevado a la incorporación del patrón sidecar para facilitar la comunicación, la observabilidad y la seguridad entre servicios. En este sentido, Envoy Proxy [32] se ha consolidado como un proxy distribuido de alto rendimiento, capaz de manejar tanto el tráfico entrante como el saliente, facilitando funciones como el descubrimiento dinámico de servicios, balanceo de carga y soporte para múltiples protocolos. Investigaciones como las de Jiao et al. [33] y Lublinsky et al. [34] abordan arquitecturas que integran sidecars para extender la funcionalidad de aplicaciones Cloud-Native, aunque se ha evidenciado que el encadenamiento de sidecars puede introducir *overhead* principalmente debido al parsing de protocolos como HTTP y gRPC, entre otras operaciones básicas tal como se analiza en [35]. Además, estudios en entornos Mobile-Cloud-Edge señalan que, aunque el uso de proxies sidecar puede incrementar la latencia y el consumo de recursos [36], la solución de encadenamiento de side-

cars propuesta en este trabajo resulta adecuada para escenarios IT convencionales, permitiendo una rápida experimentación de nuevas funcionalidades.

Finalmente, el panorama de la compresión de vídeo ha experimentado importantes avances con la transición de estándares de compresión de vídeologizado como de H.264/AVC a H.265/HEVC [37] y, más recientemente, a AV1 [38, 39], este último destacado por su alta eficiencia y carácter *open-source*. La evaluación comparativa entre códecs se basa en métricas de calidad como PSNR y VMAF, y en herramientas de análisis como la métrica Bjøntegaard Delta [40, 41], ampliamente utilizada para cuantificar la eficiencia en términos de bitrate y calidad entre pares de codificadores[42, 43, 44, 45]. Investigaciones adicionales han demostrado que la aceleración por GPU no solo mejora la velocidad de codificación, sino que también incrementa la eficiencia energética y reduce de manera significativa los tiempos de procesamiento, tal como se expone en estudios comparativos [46, 47]. Estos avances permiten, además, comparar de forma exhaustiva las implementaciones aceleradas por GPU frente a aquellas basadas en CPU, proporcionando una perspectiva integral sobre el impacto de la optimización mediante hardware especializado en la calidad de los vídeos codificados.

Además de las soluciones tradicionales para codificación y transcodificación en entornos Cloud y de microservicios, el streaming adaptativo de vídeos 360 ha ganado relevancia para optimizar la experiencia en entornos de realidad virtual y reducir el uso de ancho de banda. En este contexto, Zare et al. [48] proponen métodos para minimizar el consumo de ancho de banda manteniendo alta calidad visual en el FoV del usuario. Asimismo, se ha desarrollado un sistema de codificación y streaming de bajo retardo que predice el FoV por frame, codificando únicamente la zona de interés más un margen adicional, lo que permite reducir la latencia y adecuarse a condiciones de red variables [49]. Por otro lado, arquitecturas como la presentada en [50] logran disminuir el consumo de ancho de banda en aproximadamente un 60 % mediante el streaming del viewport del usuario y el ensamblaje de mosaicos en el servidor, mientras que otros enfoques han distribuido las tareas de codificación entre la nube, el edge y el fog para descargar la carga computacional del cliente [51]. Finalmente, sistemas viewport-aware que utilizan el particionado en tiles y MPEG-DASH para adaptar el bitrate a la región de

interés han sido estudiados en [52]. En contraste, nuestro trabajo centraliza la codificación en la nube mediante aceleración por GPU utilizando el estándar H.264, lo que garantiza mayor compatibilidad entre dispositivos y elimina la dependencia de hardware especializado o la necesidad de predicción del viewport.

El desarrollo de esta tesis parte de trabajos previos desarrollados por el grupo de investigación. Por una parte, hay una base sólida en codificación orientada a la calidad mediante diversos enfoques innovadores que combinaban encoding adaptativo, procesamiento basado en escenas y arquitecturas Serverless avanzadas. Se centró en dos algoritmos principales de encoding adaptativo [53]. El primero, *Quality-Driven Segment Encoding*, aprovecha métricas objetivas como VMAF para ajustar dinámicamente el CRF de cada segmento de vídeo, garantizando una calidad perceptual constante mientras se minimiza el tamaño del archivo. El segundo, un *Iterative Adaptive Algorithm*, refina este enfoque calculando el CRF en función de la complejidad espacial y temporal capturada mediante mapas SI-TI logrando así mejoras tanto en calidad como en eficiencia de almacenamiento frente a métodos tradicionales con parámetros fijos.

Junto a estas técnicas adaptativas, se exploró la codificación basada en escenas [54], donde se emplean representaciones en baja resolución para detectar rápidamente cambios de escena. Este método no solo reduce el tiempo de procesamiento y el consumo de recursos, sino que también alinea la segmentación del vídeo con transiciones naturales del contenido, resultando especialmente efectivo para streams en tiempo real y mejorando el empaquetado DASH [55].

Complementando estas estrategias de codificación, se desarrolló una arquitectura Serverless basada en Knative sobre Kubernetes [56]. Esta plataforma distribuye en paralelo tareas de codificación y evaluación de calidad mediante funciones contenerizadas, incluyendo encoders dedicados para x264 y AV1, así como un módulo de evaluación de calidad basado en VMAF. Análisis temporales y del uso de recursos demostraron que el enfoque Serverless es escalable y eficiente bajo diversos escenarios de carga. Además, se introdujo VQMTK [57], un *toolkit open-source* de métricas de calidad de vídeo que integra 14 métricas (como PSNR, SSIM, VMAF y CAMBI) en un entorno Dockerizado, compatible con ejecución multiplataforma.

1.2. Motivación de la tesis

A pesar de los significativos avances documentados en la literatura, persisten diversas brechas de investigación que motivan esta tesis. Por ello se han planteado un objetivo principal y objetivos secundarios con la finalidad de llenar estos vacíos en la literatura y expandir el estado del arte.

Objetivo Principal

- Diseñar, implementar y analizar arquitecturas para la codificación de vídeo de alta resolución en DASH basada en el uso de aceleradores en entornos Cloud-Edge.

Objetivos Secundarios

- **Evaluación comparativa de códecs en CPU y GPU**: Realizar un estudio comparativo de códecs implementados sobre CPU y GPU usando diferentes métricas objetivas.
- **Codificación paralela y escalable**: Diseñar e implementar un sistema de codificación paralela para vídeo bajo demanda que aproveche entornos en la nube, integrándolo en una arquitectura orientada a eventos (event-driven) capaz de escalar dinámicamente en función de la carga y de las necesidades de procesamiento.
- **Herramientas de monitoreo y despliegue**: Desarrollar un conjunto de herramientas integrales que faciliten el despliegue del sistema, la recolección y el análisis de métricas clave como tiempos de codificación y consumo de recursos.
- **Optimización del bitrate en vídeos inmersivos**: Investigar y proponer metodologías innovadoras para mejorar el manejo del bitrate en transmisiones de vídeo 360°, minimizando la transmisión de datos innecesarios. Incluyendo el análisis adaptativo que ajuste la calidad del stream en función del FoV para alcanzar los requisitos de live streaming.

- **Sistema de codificación live para video 360**: Desarrollar un sistema de codificación en vivo para la transmisión en live de vídeo 360 inmersivo en resolución 4K, que garantice eficiencia y baja latencia en entornos Cloud-Edge.

Teniendo en cuenta estos objetivos, se han realizado las siguientes contribuciones. En primer lugar, aunque la evolución de arquitecturas monolíticas hacia microservicios ha brindado escalabilidad sin precedentes, los métodos tradicionales de integración de sidecars permanecen en gran medida estáticos y manuales. Esto genera un cuello de botella en entornos Cloud-Native dinámicos donde la rapidez en despliegue y configuración es fundamental. Aunque estudios como los de Burns et al. [58] y Dragoni et al. [59] han explorado patrones sidecar, la falta de encadenamiento dinámico de sidecars limita su aplicabilidad práctica en infraestructuras en evolución constante. El trabajo *K8Sidecar: a modular Kubernetes chain of sidecar proxies for microservices and serverless architectures* aborda esta brecha proponiendo un framework de inyección dinámica de sidecars mediante operadores de Kubernetes.

En segundo lugar, aunque la computación Serverless ofrece soluciones escalables y rentables, su aplicación en entornos Cloud-Edge con recursos limitados no ha sido explorada en profundidad. La mayoría de las investigaciones existentes se centran en despliegues Cloud a gran escala, dejando un vacío en la comprensión de los desafíos al ejecutar funciones Serverless en dispositivos Edge con recursos restringidos. La integración de la aceleración GPU dentro de frameworks Serverless, especialmente en entornos Edge, ha sido aún menos estudiada. La contribución principal de *Towards GPU-Enabled Serverless Cloud Edge Platforms for Accelerating HEVC Video Coding* aborda directamente esta brecha al desarrollar una plataforma Cloud-Edge Serverless habilitada por GPU para la codificación de vídeo HEVC, combinando así la gestión nativa de eventos que ofrece Serverless con los beneficios de rendimiento de la aceleración GPU.

Las tecnologías de codificación vídeo continúan evolucionando, con códecs emergentes como AV1 ampliando los límites en eficiencia de compresión. Sin embargo, aún son escasas las evaluaciones exhaustivas comparando implementaciones CPU y GPU entre distintos códecs en resoluciones como 4K. Aunque estudios previos han abordado el rendimiento de códecs individuales [37, 60], sigue existiendo la necesidad de análisis sistemáticos sobre los compromisos

entre velocidad de codificación, consumo de recursos y métricas de calidad como PSNR, VMAF y BD-Rate. La aportación *UHD Video Encoding in CPU Versus GPU Quality and Performance Trade-offs* completa este vacío proporcionando un análisis comparativo detallado entre codificación con CPU y GPU en múltiples familias de códecs.

En conjunto, las tres contribuciones principales de esta tesis ofrecen un enfoque unificado para abordar estos desafíos. Al cerrar la brecha entre arquitecturas estáticas tradicionales y sistemas Cloud-Native modernos y dinámicos, nuestro trabajo no solo avanza en la comprensión académica, sino que también proporciona soluciones prácticas para la próxima generación de aplicaciones de procesamiento y streaming de vídeo. Esta síntesis entre microservicios, computación Serverless y codificación acelerada por GPU establece una base sólida para los estudios detallados presentados en los capítulos siguientes.

1.3. Listado de trabajos resultantes

A continuación, se muestran los trabajos publicados, derivados de los avances de esta tesis en el campo de la codificación de vídeo y su implementación en entornos Cloud-Native y Cloud-Edge, que optimizan el procesamiento y distribución de contenido multimedia.

Trabajos publicados en revistas JCR y que soportan la presentación de la tesis bajo la modalidad de compendio de publicaciones

- Salcedo Navarro, A., Garcia Pineda, M. and Gutiérrez Aguado, J. (2025), K8sidecar: A Modular Kubernetes Chain of Sidecar Proxies for Microservices and Serverless Architectures. Softw: Pract Exper. https://doi.org/10.1002/spe.3423
- Salcedo Navarro A., Gutiérrez Aguado J. and Garcia Pineda M. UHD Video Encoding in CPU Versus GPU: Quality and Performance Trade-Offs, in IEEE Access, vol. 13, pp. 55115-55129, 2025. https://doi.org/10.1109/ACCESS.2025.3553634.

- Salcedo Navarro, A., Peña Ortiz, R., Claver, J.M. Garcia Pineda, M. and Gutiérrez Aguado, J. Towards GPU-enabled serverless cloud edge platforms for accelerating HEVC video coding. Cluster Comput 28, 68 (2025). https://doi.org/10.1007/s10586-024-04692-0

Trabajos adicionales publicados en Conferencias Internaciones que aparecen en la clasificación CORE

- Salcedo Navarro A. Garcia Pineda M., and Gutiérrez Aguado, J. 2025. PodInsights: a millisecond pod metric collector for Kubernetes. In Proceedings of the 12th Euro American Conference on Telematics and Information Systems (EATIS 2024). Association for Computing Machinery, New York, NY, USA, Article 16, 1–4. https://doi.org/10.1145/3685243.3685281

- Salcedo Navarro, A., Peña Ortiz, R., Claver, J.M., Garcia Pineda, M., Gutiérrez Aguado, J. (2024). Cloud-Native GPU-Enabled Architecture for Parallel Video Encoding. In: Carretero, J., Shende, S., Garcia-Blas, J., Brandic, I., Olcoz, K., Schreiber, M. (eds) Euro-Par 2024: Parallel Processing. Euro-Par 2024. Lecture Notes in Computer Science, vol 14803. Springer, Cham. https://doi.org/10.1007/978-3-031-69583-4_23

- Salcedo Navarro, A., Garcia Pineda, M. and Gutiérrez Aguado, J. (2025). Cloud-Edge Architecture for 4K 360-Degree Video Encoding in FoV-based Live DASH, IEEE International Symposium on Circuits and Systems (ISCAS), London, UK. (Accepted). preprint: Enlace a la publicación

1.4. Estructura de la tesis

Esta tesis está organizada en varios capitulos clave que presentan en conjunto un estudio de arquitecturas avanzadas de codificación de vídeo en entornos Cloud-Native y Cloud-Edge. El documento inicia con un capítulo introductorio que establece el contexto exponiendo los desafíos actuales en despliegues Cloud-Native modernos, microservicios y codificación de vídeo. El apartado Contexto y Trabajos Relacionados ofrece una revisión exhaustiva de la literatura exis-

tente, identificando las brechas de investigación que motivan este trabajo.

El núcleo de la tesis se divide en tres grandes capítulos donde se describen cada una de las aportaciones principales y trabajos relacionados. La primera aportación, presentada en el capitulo 2 *Herramientas para la Integración Dinámica y Monitorización de Alta Frecuencia en Arquitecturas Cloud-Native*, introduce K8Sidecar y PodInsights herramientas fundamentales para el despliegue dinámico y la monitorización del rendimiento en las contribuciones posteriores. A continuación, el capitulo 3 *Aportación en UHD Video Encoding in CPU Versus GPU Quality and Performance Trade-offs* detalla un exhaustivo análisis comparativo de implementaciones de codificación de vídeo, analizando las ventajas y desventajas entre codecs basados en CPU y GPU respecto a métricas de calidad y eficiencia.

El capitulo 4, titulado *Arquitecturas avanzadas con GPU para codificación de vídeo en entornos Cloud-Native y Cloud-Edge*, está dedicada a las contribuciones centradas en arquitecturas Cloud-Edge que explotan la presencia de GPUs para realizar la codificación de vídeo. Esta sección incluye un análisis en profundidad del trabajo principal sobre *Towards GPU-Enabled Serverless Cloud Edge Platforms for Accelerating HEVC Video Coding*, así como dos contribuciones adicionales que extienden el enfoque al procesamiento paralelo y al streaming inmersivo adaptativo.

Finalmente, se presenta el capitulo 5, que sintetiza los hallazgos obtenidos a lo largo de todas las contribuciones, discute limitaciones y lineas futuras de investigación. La tesis finaliza con la inclusión de las contribuciones publicadas en revistas indexadas en JCR, las cuales respaldan el compendio de publicaciones.

Capítulo 2

Herramientas para la Integración Dinámica y Monitorización de Alta Frecuencia en Arquitecturas Cloud-Native

Los sistemas modernos Cloud-Native requieren tanto agilidad en el despliegue como precisión en la monitorización. La rápida evolución de las arquitecturas de microservicios y Serverless requieren herramientas que no solo permitan escalado dinámico e integración flexible, sino que también proporcionen información en tiempo real sobre el rendimiento del sistema. En el contexto de la tesis se han desarrollado dos soluciones especializadas para abordar estos desafíos, K8Sidecar y PodInsights. Estas herramientas fueron concebidas para superar las limitaciones inherentes a las técnicas de configuración estática y monitorización poco granular, que pueden perjudicar la capacidad de respuesta y escalabilidad en sistemas distribuidos. El despliegue dinámico de microservicios exige que funcionalidades auxiliares (seguridad, logging o integración de un *datamesh*) sean inyectadas de forma transparente en tiempo de ejecución, mientras que la recolección de métricas en altas frecuencias son esenciales para ajustar el uso de recursos en entornos que gestionan tareas de duración reducida o con fluctuaciones rápidas.

K8Sidecar, es un mecanismo modular de proxy que permite la

inyección dinámica de contenedores sidecar, contenedores auxiliares al contenedor de aplicación, en *pods* de Kubernetes. Aprovechando el *operator pattern* de Kubernetes y los CRD (Custom Resource Definition), K8Sidecar permite un control ajustado del orden y configuración de los contenedores sidecar. Esta flexibilidad permite a los desarrolladores añadir funcionalidades sin necesidad de modificar la aplicación principal, facilitando así procesos de despliegue e integración continua.

Complementando a K8Sidecar está PodInsights, un colector avanzado de métricas a nivel de *pods* diseñado para la adquisición de datos con una resolución de milisegundos. A diferencia de herramientas tradicionales como *Docker Stats* que típicamente ofrecen actualizaciones en intervalos mayores a un segundo, PodInsights accede directamente a los datos del cgroup del sistema operativo mediante *DaemonSets* de Kubernetes, posibilitando así una monitorización ultra precisa del uso de CPU y memoria. Actualmente se sigue extendiendo la aplicación para acceder a más datos como estadísticas de red o numero de operaciones de entrada y salida.

En conjunto, estas herramientas forman un ecosistema integrado que no solo simplifica la orquestación dinámica, sino que también asegura que las métricas de rendimiento sean recopiladas en tiempo real. Esta base sustenta el resto contribuciones, facilitando funcionalidades avanzadas como la integración con *datamesh* a través de despliegues en arquitecturas Cloud-Native.

2.1. Aportación K8Sidecar: a modular Kubernetes chain of sidecar proxies for microservices and serverless architectures

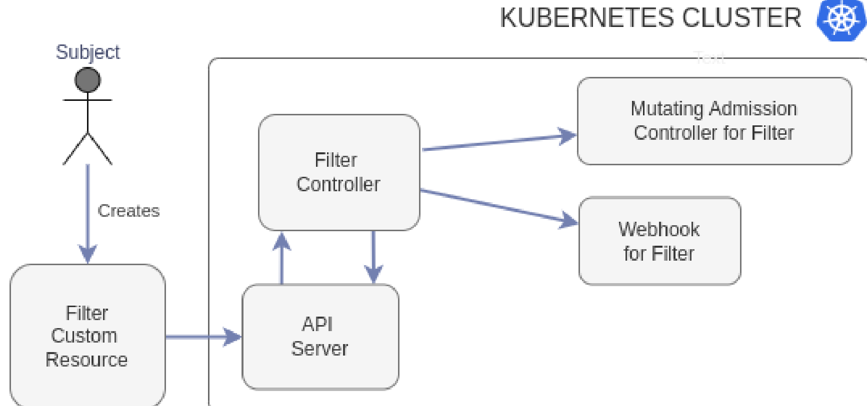
Las aplicaciones modernas Cloud-Native demandan una agilidad y flexibilidad sin precedentes. Las implementaciones tradicionales estáticas de sidecar, aunque útiles, a menudo requieren configuración manual, volver a desplegar o incluso recompilar, lo que puede ocasionar retrasos operativos e inconsistencias. K8Sidecar[1] (ver capitulo 6) fue desarrollado para superar estas limitaciones al permitir un enfoque dinámico, en tiempo de despliegue, para la inyección de sidecars en entornos Kubernetes. Al desacoplar funcionalidades au-

[1] GitHub: https://github.com/cloudmedialab-uv/k8sidecar

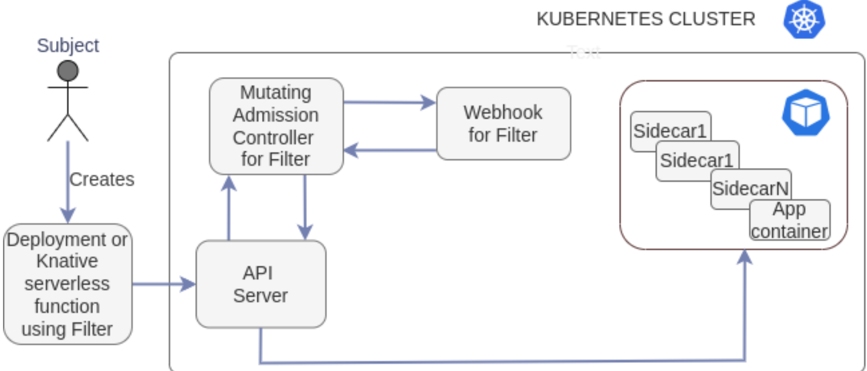
xiliares tales como seguridad, logging y monitorización de la lógica central de la aplicación, K8Sidecar refuerza el principio de responsabilidad única, asegurando que las aplicaciones se mantengan ligeras al mismo tiempo que acceden a funcionalidades operativas avanzadas.

Esta solución aprovecha capacidades nativas de Kubernetes introduciendo un recurso personalizado *Filter CRD* que define una cadena de contenedores sidecar a ser inyectados automáticamente al crear un *pod*. Esta automatización no solo reduce la carga manual, sino que también garantiza consistencia y escalabilidad en arquitecturas de microservicios y Serverless. En esencia, K8Sidecar permite integrar funcionalidades adicionales de forma transparente durante el *runtime*, apoyando así *pipelines* de integración continua y despligue automatico, al tiempo que mejora el rendimiento global del sistema.

La arquitectura de K8Sidecar está basada en el *operator pattern*, diseñando un nuevo CRD denominado *Filter*, que permite especificar una cadena ordenada de contenedores sidecar. Cada sidecar en la cadena se define con propiedades como la imagen del contenedor, el nombre, la prioridad, variables de entorno y volúmenes. Cuando un pod es desplegado con etiquetas que coinciden con un *Filter* definido, el operador K8Sidecar invoca un *Mutating Admission Webhook*. Este *webhook* intercepta la solicitud de creación del pod y modifica dinámicamente su especificación eliminando la lista original de contenedores y agregando una nueva lista ordenada que incluye tanto el contenedor principal de la aplicación como los sidecars adicionales especificados en el *Filter*. El mecanismo de orden basado en prioridades asegura que el contenedor de la aplicación conserve la máxima prioridad de ejecución, situándose al final de la cadena, asegurando así un correcto flujo del tráfico y procesamiento de datos. Este diseño no solo optimiza la integración de funcionalidades auxiliares, sino que también apoya casos de uso complejos como la integración con *datamesh*, donde la comunicación interna de servicios es necesaria. Además, K8Sidecar ofrece bibliotecas en Golang, Java y esta siendo ampliado a Python, ampliando su aplicabilidad y aprovechando las fortalezas únicas de cada lenguaje de programación. La arquitectura general se ilustra en las Figuras 2.1a y 2.1b, detallando las interacciones entre el CRD, webhook y los *pods* desplegados, destacando el proceso de inyección dinámica.



(a) El controlador del operador recibe una notificación cuando se crea un nuevo Filter. Se crea automáticamente un nuevo controlador de admisión que permite modificar la petición y su webhook para ese Filter.



(b) Cuando se despliega un recurso que utiliza el Filter, el webhook inyecta los sidecars especificados en el Filter.

Figura 2.1: Arquitectura general de K8Sidecar.

K8Sidecar está implementado como un operador que monitoriza continuamente la API de Kubernetes en busca de nuevos recursos *Filter* y despliegues de *pods* correspondientes. El bucle de control principal sincroniza el estado deseado definido en la CRD con el estado real del clúster. Cuando se detecta un pod que requiere inyección de sidecar, el *Mutating Admission Webhook* construye un *JSONPatch* que modifica en tiempo real la especificación del pod. Nuestras evaluaciones experimentales se centraron en medir tanto los tiempos de *coldstart* como las latencias introducidas en tiempo de ejecución al variar el número de sidecars especificadas en el filtro

como el número de peticiones concurrentes solicitando la creación de *pods*. Los resultados indican que la sobrecarga adicional es mínima cuando el número de sidecars no es elevado y se encuentra dentro de límites aceptables para microservicios modernos. En experimentos controlados comparando la inyección dinámica de K8Sidecar con configuraciones estáticas tradicionales, observamos mejoras notables en la agilidad del despliegue y la eficiencia operativa con un número moderado de sidecars. Además, casos prácticos demostraron que el diseño flexible de K8Sidecar permitió la integración transparente de funcionalidades auxiliares tales como filtros de seguridad y *datamesh* sin impactar el rendimiento central de la aplicación. Estos resultados validan la eficacia de K8Sidecar en entornos Kubernetes en el mundo real.

2.2. Propuesta para la mejora en la obtención de métricas de pods sobre kubernetes

En entornos dinámicos Cloud-Native, las herramientas tradicionales de monitorización como *Docker Stats* y *cAdvisor* típicamente proporcionan actualizaciones de datos en intervalos que no pueden bajar de un segundo. Esta granularidad a menudo no logra captar fluctuaciones rápidas en el consumo de recursos en trabajos de corta duración, algo crítico en escenarios como la codificación de vídeo en tiempo real o aplicaciones intensivas basadas en microservicios. Esta limitación puede llevar a una detección tardía de anomalías, asignación ineficiente de recursos y una falta generalizada de información útil para la optimización del rendimiento. Reconociendo estos desafíos, PodInsights[2,3] fue desarrollado para proporcionar monitorización a nivel de milisegundos del uso de CPU y memoria de los *pods* en clusters de Kubernetes. Al acceder directamente a los archivos cgroup del sistema operativo, PodInsights evita los retrasos asociados a abstracciones de más alto nivel. Esta monitorización de alta resolución permite observar comportamientos transitorios, como picos breves en el consumo de CPU o caídas repentinas de memoria, que a menudo quedan ocultos con herramientas convencionales. Consecuentemente, este nivel detallado de información facilita decisiones más ágiles de auto escalado y una gestión más precisa de los

[2]DOI: https://doi.org/10.1145/3685243.3685281
[3]GitHub: https://github.com/cloudmedialab-uv/PodInsights

recursos, garantizando que el rendimiento del sistema y la eficiencia de costes se optimicen en tiempo real. La precisión ofrecida por PodInsights es indispensable en entornos donde incluso mínimas variaciones en el uso de recursos pueden tener un impacto significativo en la calidad global del servicio.

PodInsights está diseñado para ofrecer monitorización con granularidad alta añadiendo una mínima carga adicional. Se despliega como un *DaemonSet* de Kubernetes, asegurando que una instancia del agente de monitorización se ejecute en cada nodo del cluster, proporcionando así una visión completa y en tiempo real del uso de recursos en todos los *pods*. Cada agente utiliza acceso directo al sistema de archivos cgroup donde se almacenan las estadísticas de CPU y memoria para extraer datos en intervalos configurables, establecidos mediante un *ConfigMap*, del orden de milisegundos. Este enfoque evita las limitaciones de las APIs tradicionales de monitorización a nivel de contenedor, que suelen promediar métricas en intervalos más largos.

Una vez extraídos los datos, una aplicación procesa y transmite estas métricas a una base de datos MongoDB para almacenamiento persistente y posterior análisis. Esta integración no solo permite análisis de tendencias a largo plazo, sino que también facilita la visualización inmediata y detección de anomalías mediante herramientas externas. Además, PodInsights puede obtener métricas solo de determinados *pods* según etiquetas, asegurando que aplicaciones críticas sean monitorizadas con una alta precisión. La arquitectura general se ilustra en la Figura 2.2, mostrando cómo el DaemonSet, los parámetros de configuración y el almacenamiento externo interactúan para conseguir una monitorización de grano fino. Este diseño modular y escalable permite a PodInsights integrarse fluidamente con otras herramientas del ecosistema, como K8Sidecar, posibilitando funcionalidades avanzadas como la centralización de métricas del sistema en una base de datos para facilitar el procesamiento.

Evaluaciones experimentales de PodInsights han demostrado su superioridad frente a soluciones convencionales de monitorización. En escenarios de alta carga, como tareas intensivas de codificación de vídeo, PodInsights logró recoger hasta 20 veces más muestras de datos por minuto que Docker Stats, capturando patrones transitorios de consumo de recursos previamente indetectables. Estos datos adicionales proporcionaron visibilidad sobre picos de corta duración

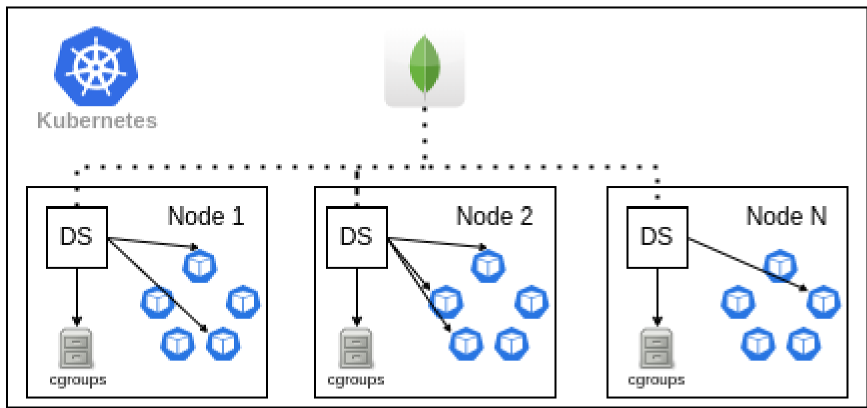


Figura 2.2: Arquitectura para PodInsights en Kubernetes. PodInsights monitoriza un subconjunto de *pods* en cada nodo (basado en etiquetas). Las métricas recopiladas se almacenan en MongoDB.

en la utilización de CPU y fluctuaciones rápidas de memoria, lo que permitiría decisiones más precisas y oportunas en materia de auto escalado y asignación de recursos. Adicionalmente, las pruebas de rendimiento confirmaron que el diseño de PodInsights introduce un *overhead* mínimo, asegurando que el proceso de monitorización no interfiera con el rendimiento de los *pods*. En definitiva, la mayor frecuencia y precisión en la toma de muestras de PodInsights han resultado fundamentales para optimizar estrategias de gestión de recursos en entornos Cloud-Native.

2.3. Integración de PodInsights y K8sidecar en otros trabajos realizados

La integración de PodInsights y K8sidecar en la arquitectura presentada en esta tesis cumple varios objetivos fundamentales. Facilitar la recolección y centralización de métricas de rendimiento, abstraer y gestionar el proceso de subida y descarga de vídeos a través de un *datamesh*, adaptación de mensajes AMQP a HTTP y Monitorizar recursos de la GPU.

En primer lugar, PodInsights se emplea para recolectar métricas tanto de CPU como de memoria en los experimentos donde se requiere un monitoreo detallado de estos recursos. La alta granularidad es fundamental para evaluar el comportamiento del sistema

durante la codificación y para tomar decisiones que optimicen la asignación de recursos en entornos Cloud-Edge.

Por otro lado, K8sidecar se utiliza para inyectar un sidecar dedicado a la gestión de los datos, cuya función es gestionar la descarga y subida de vídeos en las *pipelines* de codificación. Siguiendo el principio de única responsabilidad, los workers de codificación se centran exclusivamente en la tarea de codificar vídeos, mientras que la funcionalidad de transferencia de datos se delega a este contenedor sidecar (ver Figura 2.3). Concretamente, el sidecar opera de la siguiente manera:

- Descarga de vídeos: El sidecar intercepta el cuerpo de la petición, donde se especifican los vídeos que deben descargarse, y procede a bajarlo en la ruta designada para que el contenedor principal pueda acceder a él y realizar la codificación.
- Subida de vídeos: Una vez finalizado el proceso de codificación, el mismo sidecar se encarga de subir los vídeos resultantes al servidor especificado en el cuerpo de la petición.

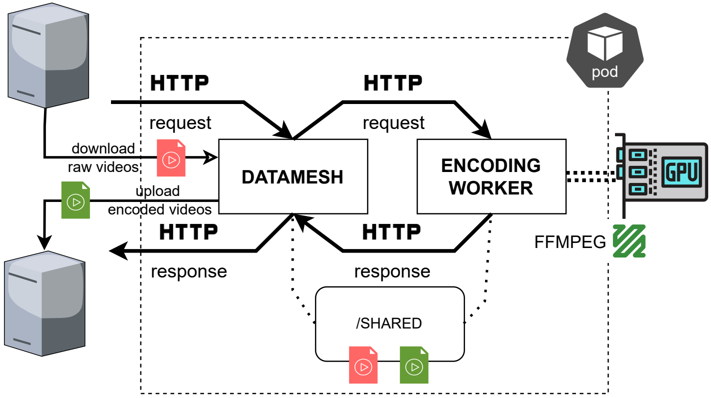


Figura 2.3: Interacción entre el contenedor de codificación y el sidecar del *datamesh*, encargado de gestionar la subida y descarga de vídeos mediante un volumen compartido

Esto se consigue gracias a la capacidad que ofrece K8sidecar para gestionar volúmenes compartidos entre el contenedor principal y el sidecar, permitiendo que la transferencia se realice sin interferir en la lógica de codificación. Esta separación de responsabilidades no solo

simplifica el diseño de la aplicación, sino que también permite una mayor flexibilidad. Por ejemplo, en caso de que se desee cambiar el sistema que gestiona los vídeos, como migrar a un servidor de almacenamiento de objetos como *Amazon S3*, bastará con modificar el sidecar sin necesidad de alterar la funcionalidad de la aplicación principal ni el despliegue del sistema.

Además, se ha combinado el uso de K8sidecar con PodInsights para abordar uno de los mayores retos en la monitorización de entornos distribuidos en Kubernetes que es la recolección de métricas de utilización de la GPU. Obtener datos precisos sobre el porcentaje de uso y la memoria empleada de la GPU de forma directa desde el contenedor principal implicaría una carga adicional que podría afectar al rendimiento del proceso de codificación. Para evitar este *overhead*, se ha implementado un sidecar adicional dedicado exclusivamente a monitorizar la GPU durante la ejecución del proceso de codificación. Este sidecar se activa con el inicio de un evento de codificación, recoge de forma continua las métricas de la GPU y, una vez finalizado el proceso, envía los datos a la API de PodInsights. De esta forma, se centralizan las métricas de GPU junto con las de CPU y memoria, garantizando una monitorización completa sin impactar el rendimiento del proceso principal, facilitando su posterior procesamiento y análisis (ver Figura 2.4).

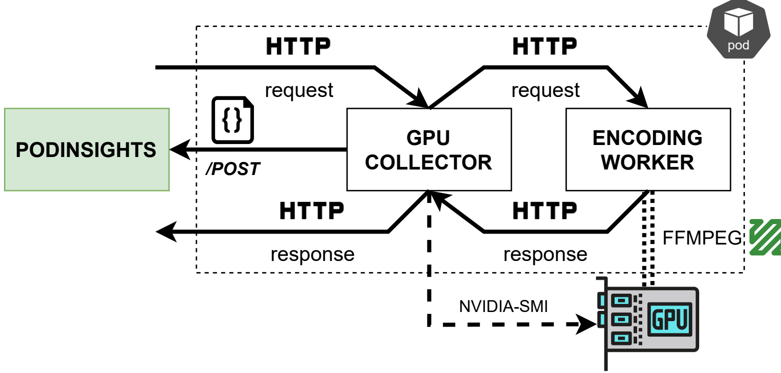


Figura 2.4: Interacción entre el contenedor de codificación y el sidecar de monitorización GPU, que recolecta métricas con nvidia-smi y las envía a PodInsights sin afectar el rendimiento del proceso principal.

Por último, el uso de K8sidecar también ha permitido adaptar las peticiones enviadas mediante el protocolo AMQP desde RabbitMQ a CloudEvents, encapsulados en HTTP, permitiendo aprovechar las funciones y sidecars desarrollados sin cambiar una línea de código. Dado que el worker de codificación está diseñado para operar mediante llamadas HTTP/CloudEvents, se ha implementado un sidecar adicional que actúa como puente entre los mensajes AMQP y el ecosistema de servicios HTTP (ver Figura 2.5). Este adaptador traduce los mensajes recibidos desde RabbitMQ en peticiones HTTP estandarizadas, permitiendo que tanto el contenedor principal como el resto de sidecars involucrados en el proceso de codificación puedan interpretar correctamente las instrucciones, manteniendo así una arquitectura desacoplada, coherente y extensible.

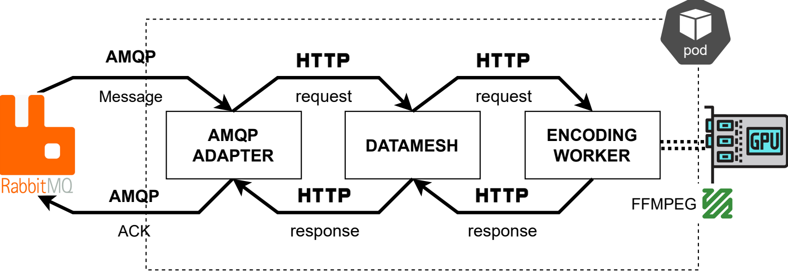


Figura 2.5: Interacción entre el contenedor de codificación y el sidecar de adaptación AMQP-HTTP, que traduce los mensajes provenientes de RabbitMQ en peticiones HTTP.

En resumen, la integración de PodInsights y K8sidecar permite tanto una recolección de métricas precisa y centralizada como una gestión flexible y desacoplada de la transferencia de vídeos, optimizando así las *pipelines* de codificación en entornos Cloud-Edge.

Capítulo 3

Análisis de calidad y tiempos de codificación para codecs basados en CPU y en GPU en vídeo UHD

En el panorama actual de los medios digitales, la demanda de contenidos de vídeo de alta resolución ha aumentado drásticamente. La codificación de vídeo UHD (4K) constituye una tecnología clave para los servicios de streaming, videoconferencias y almacenamiento multimedia. Sin embargo, el desafío es doble. Por un lado, comprimir eficientemente los datos de vídeo y por otro, mantener al mismo tiempo una alta fidelidad de imagen. Los métodos tradicionales de codificación, al ejecutarse en CPUs, a menudo tienen dificultades para equilibrar estas demandas, resultando en vídeos con mayor bitrate o tiempos de codificación más largo. Esta situación subraya la importancia de comparar las implementaciones basadas en CPU con aquellas aceleradas mediante GPU. Las GPUs, gracias a sus capacidades inherentes de procesamiento paralelo, prometen mejoras significativas en velocidad y eficiencia de codificación. En este contexto, nuestro estudio analiza rigurosamente la eficiencia de codificación de tres grandes familias de códecs H.264/AVC, H.265/HEVC y AV1.

El objetivo principal de la tesis era extender arquitecturas previamente estudiadas sobre CPUs para permitir el uso de GPUs. Sin embargo, una parte importante es caracterizar cómo se comportan

las implementaciones de los códecs que usan GPU frente a las que se basan exclusivamente en CPU

3.1. Aportación: UHD Video Encoding in CPU Versus GPU Quality and Performance Trade-offs

El artículo (ver capitulo 7) se fundamenta en la necesidad de mitigar los cuellos de botella computacionales encontrados en la codificación de vídeo UHD para entornos live. A medida que continúan aumentando las resoluciones y tasas de fotogramas de vídeo, resulta imperativo alcanzar niveles aceptables de calidad (medidos mediante métricas como PSNR y VMAF), manteniendo simultáneamente una baja latencia en la codificación. Utilizando las métricas BD (Bjøntegaard Delta), nuestro estudio cuantifica las diferencias en bitrate y calidad entre los enfoques basados en CPU y GPU. Esta evaluación comparativa no solo resalta el potencial para ahorrar tiempos significativos especialmente en códecs emergentes como AV1, sino que también informa sobre decisiones relativas a la selección de códecs en entornos con ancho de banda limitado.

Para evaluar de forma integral el rendimiento de la codificación de vídeo basada en CPU y GPU, se llevaron a cabo una serie de experimentos sobre un conjunto diverso de 15 vídeos UHD (4K) del dataset UVG (Ultra Video Group) [61]. Estos vídeos, que abarcan un amplio rango de complejidades espaciales y temporales, sirven como un *benchmark* riguroso para probar la eficiencia de codificación. El estudio se centra en tres familias de códecs: H.264/AVC, H.265/HEVC y AV1. Para cada familia se seleccionaron dos implementaciones: una que se basa exclusivamente en procesamiento sobre la CPU (*libx264*, *libx265* y *librav1e*) y otra que aprovecha la aceleración GPU (*h264_nvenc*, *hevc_nvenc* y *av1_nvenc*).

La configuración experimental se diseñó meticulosamente para asegurar la consistencia en todas las pruebas. Las tareas de codificación se ejecutaron en el mismo entorno de contenedor para evitar interferencias de procesos externos. El cluster se configuró en dos máquinas físicas conectadas mediante una red de 10 Gb/s. Una aloja el nodo maestro de Kubernetes, mientras que la otra se utiliza para desplegar múltiples nodos *worker* equipados con GPUs, em-

pleadas en los escenarios que lo requieren. Para la codificación de vídeo se empleó la herramienta FFmpeg, con parámetros específicos de línea de comandos para establecer bitrates constantes y asegurar una configuración consistente en todas las pruebas, el comando utilizado se muestra en el Listado 3.1. El rango de bitrate seleccionado, las configuraciones predefinidas (presets) y el modo Random Access garantizaron resultados representativos de escenarios reales de streaming.

```
ffmpeg -i video.mp4 -c:v <codec> -b:v <bitrate>
    -maxrate <bitrate> -minrate <bitrate> enc-video.mp4
```

Listado 3.1: Ejemplo de comando ffmpeg utilizado para codificar los vídeos

Las métricas de evaluación fueron fundamentales para nuestro análisis. Se utilizó BD (Bjøntegaard Delta) [45], que mediante interpolación polinómica, estima la diferencia promedio en bitrate o en calidad de dos códecs como la diferencia entre las integrales de las dos curvas interpoladas. En este caso se calcularon las curvas utilizando las medidas de calidad PSNR y VMAF, lo que permitió cuantificar tanto la eficiencia del bitrate como la degradación en la calidad. Para interpolar con precisión los puntos discretos de bitrate/calidad, se aplicó el método de interpolación Akima [40]. Este método es reconocido por generar curvas suaves sin el comportamiento oscilatorio típico de otras interpolaciones polinómicas. Además, se registró el tiempo medio de codificación por frame para cada códec, proporcionando una medida directa de la eficiencia computacional. Estas metodologías aseguran colectivamente una evaluación cuantitativa completa del equilibrio entre la codificación de vídeo basada en CPU y GPU.

Los resultados experimentales destacan claramente una mejora en cuanto a rendimiento en favor a la codificación de vídeo acelerada por GPU. Para la familia H.264/AVC, el análisis mediante BD-Rate revela que la implementación basada en CPU (*libx264*) requiere, en promedio, más de un 20 % adicional de bitrate en comparación con su contraparte basada en GPU (*h264_nvenc*) para lograr métricas de calidad equivalentes en terminos de PSNR y VMAF. Este hallazgo se traduce en una reducción notable en los requerimientos de ancho de banda de red cuando se utiliza aceleración GPU. El tiempo medio de codificación por frame corrobora aún más esta ventaja, *h264_nvenc* logra un tiempo promedio de codificación por frame de aproximadamente 109 milisegundos, en comparación con los 139 mi-

lisegundos para *libx264* lo que indica una ganancia de rendimiento de aproximadamente 1.3 veces.

La familia H.265/HEVC presenta un contraste interesante. Aquí, la implementación en CPU (*libx265*) requiere alrededor de un 7.5 % menos de bitrate que la versión acelerada por GPU (*hevc_nvenc*) según las métricas BD-Rate. Sin embargo, considerando los tiempos de codificación, *hevc_nvenc* es en promedio 2.35 veces más rápido que *libx265*. Esto sugiere que, aunque la versión CPU podría ofrecer una ligera ventaja en bitrate, la significativa reducción del tiempo de procesamiento ofrecida por la aceleración GPU probablemente tenga más valor en escenarios donde la latencia es crítica.

Las mejoras más notables se observaron con los códecs AV1. La implementación acelerada por GPU (*av1_nvenc*) supera a la versión CPU (*librav1e*) por un factor de 68.14 veces en relación al tiempo de codificación. Aunque las diferencias en bitrate para AV1 son menos pronunciadas, esta sustancial aceleración es crítica para aplicaciones que requieren procesamiento cercano al tiempo real, como eventos de live streaming. Estos resultados están representados en la tabla Tabla 3.1, en la cual se observa la el tiempo medio en milisegundos por frame para cada familia de códecs.

Name	**libx264 (ms/f)**	**h264_nvenc (ms/f)**	**Factor GPU vs. CPU**	***libx265* (ms/f)**	**hevc_nvenc (ms/f)**	**Factor GPU vs. CPU**	**librav1e (ms/f)**	**av1_nvenc (ms/f)**	**Factor GPU vs. CPU**
Bosphorus	116.42	93.57	1.24	208.95	86.42	2.42	4361.20	102.44	42.57
HoneyBee	123.03	114.36	1.08	222.38	122.50	1.82	8091.18	144.47	56.01
Jockey	143.25	98.98	1.45	267.53	102.15	2.62	5148.62	118.36	43.50
ReadySetGo	178.06	149.30	1.19	202.34	100.59	2.01	5830.54	94.32	61.82
ShakeNDry	130.71	119.18	1.10	203.00	116.41	1.74	3883.01	119.99	32.36
YachtRide	118.74	98.35	1.21	234.55	109.04	2.15	4501.35	93.47	48.16
CityAlley	143.03	97.56	1.47	205.32	99.21	2.07	12351.46	121.93	101.30
FlowerFocus	137.17	116.43	1.18	284.53	120.13	2.37	10608.62	145.43	72.95
FlowerKids	122.95	93.98	1.31	217.22	113.27	1.92	9153.12	102.60	89.21
FlowerPan	142.73	117.12	1.22	219.61	113.63	1.93	10533.68	132.60	79.44
Lips	155.94	146.05	1.07	232.91	121.74	1.91	4715.39	124.39	37.91
RaceNight	153.82	121.02	1.27	279.10	135.40	2.06	8534.09	139.40	61.22
RiverBank	114.33	109.34	1.05	205.63	115.75	1.78	7857.04	142.67	55.07
SunBath	149.25	68.65	2.17	403.21	63.46	6.35	10356.18	65.38	158.40
Twilight	156.65	101.48	1.54	219.75	106.14	2.07	9796.44	119.09	82.26
Average	*139.07*	*109.69*	*1.30*	*240.40*	*108.39*	*2.35*	*7714.79*	*117.77*	*68.14*

Tabla 3.1: Mejora de tiempos y factores entre codecs de la misma familia

En general, los resultados demuestran que los encoders acelerados por GPU reducen significativamente los tiempos de codificación. Estos hallazgos ofrecen evidencia convincente a favor de adoptar enfoques basados en GPU para *pipelines* de codificación de vídeo UHD,

especialmente en entornos de streaming de alta demanda donde la latencia y la eficiencia del ancho de banda son cruciales.

Las implicaciones de estos hallazgos para el streaming multimedia son sustanciales. La reducción en los tiempos de codificación se traduce directamente en menor latencia y mejor capacidad de respuesta para aplicaciones de live streaming y videoconferencias. Una codificación más rápida no solo minimiza el retraso entre la captura y la entrega del vídeo, sino que también permite a los proveedores de servicios soportar técnicas de adaptive streaming de manera más efectiva. Con la codificación acelerada por GPU, se pueden entregar resoluciones más altas con calidad óptima utilizando bitrates reducidos, lo que genera importantes ahorros en ancho de banda. Esto es especialmente crucial para plataformas que utilizan HAS, donde una codificación eficiente puede mejorar significativamente el QoE, asegurando una reproducción más fluida y con menos interrupciones. Además, las ganancias de rendimiento observadas con implementaciones GPU facilitan la gestión de mayores volúmenes de vídeo y tasas de frames más altas, contribuyendo en última instancia a infraestructuras de streaming más escalables y rentables.

Las conclusiones de esta aportación son utilizadas como base para el enfoque Serverless habilitado por GPU de la aportación *Towards GPU-Enabled Serverless Cloud Edge Platforms for Accelerating HEVC Video Coding*. Los mecanismos de despliegue dinámico proporcionados por K8Sidecar ofrecen un marco flexible capaz de adaptarse al escalado rápido que requieren los *pipelines* de codificación acelerados por GPU. Del mismo modo, la plataforma Serverless Cloud-Edge discutida en la tercera aportación aprovecha estas mejoras en rendimiento, utilizando la aceleración GPU para proporcionar codificación cercana al tiempo real en escenarios de live streaming.

Capítulo 4

Arquitecturas avanzadas con GPU para codificación de vídeo en entornos Cloud-Native y Cloud-Edge

El crecimiento exponencial del streaming multimedia, impulsado por la creciente demanda de contenido inmersivo y de alta resolución por parte de los usuarios finales, ha puesto de manifiesto la imperante necesidad de contar con soluciones de codificación de vídeo altamente eficientes y adaptadas a los nuevos requerimientos tecnológicos. A medida que las plataformas de streaming transicionan hacia contenidos en 4K e incluso en vídeo inmersivo (360 grados), los métodos tradicionales de encoding basados en CPU presentan desafíos cada vez mayores debido a los requisitos de baja latencia y alta calidad. En este contexto de rápida evolución tecnológica, la aceleración mediante GPU se ha consolidado como un componente clave, al proporcionar la capacidad de procesamiento paralelo requerida para afrontar la elevada complejidad computacional asociada a los códecs de vídeo de última generación. Como ya se ha comprobado en el anterior capítulo las GPUs permiten tiempos de encoding más rápidos y posibilitan el procesamiento en tiempo real, algo esencial para aplicaciones interactivas y de live streaming. Además, la integración de soluciones basadas en GPU tanto en entornos Cloud-Native como Cloud-Edge permite una asignación dinámica de recursos y streaming adaptativo, garantizando que el contenido

de vídeo en alta resolución sea reproducido con apenas retrasos. Este cambio de paradigma hacia la utilización de la aceleración GPU no solo resuelve los cuellos de botella en el rendimiento de los sistemas tradicionales, sino que también abre nuevas oportunidades para *pipelines* escalables y eficientes en el procesamiento de vídeo.

Esta investigación presenta tres contribuciones interrelacionadas que conjuntamente amplían las fronteras tecnológicas en el ámbito del video encoding. La contribución principal es *Towards GPU-Enabled Serverless Cloud Edge Platforms for Accelerating HEVC Video Coding*, que introduce una arquitectura Serverless que aprovecha las GPUs para acelerar drásticamente la codificación HEVC en entornos Cloud-Edge. Complementando esta contribución con *Cloud-Native GPU-Enabled Architecture for Parallel Video Encoding*, se enfoca en escalar las operaciones de codificación de vídeo mediante procesamiento paralelo sobre Kubernetes, reduciendo así la latencia y mejorando la tasa de transferencia. Por otro lado, la contribución *Cloud-Edge Architecture for 4K 360-Degree Video Encoding in FoV-based Live DASH*, aborda los desafíos de la codificación y el streaming inmersivo mediante técnicas adaptativas basadas en FoV (Field of View) para optimizar la calidad y el uso del ancho de banda. En conjunto, estas contribuciones impulsan las contribuciones en procesamiento de vídeo, combinando orquestación Serverless, codificación paralela y metodologías de streaming adaptativo, satisfaciendo así las demandas crecientes de las aplicaciones multimedia modernas.

4.1. Propuesta de arquitectura sobre Kubernetes para codificar de forma paralela los segmentos de vídeo

Esta contribución[1,2] se fundamenta en la elevada demanda de recursos computacionales que requiere la codificación de video, especialmente en implementaciones basadas en estándares modernos como H.264, lo que posibilita distribuir la carga de trabajo entre múltiples nodos de procesamiento. En los sistemas convencionales, las cargas de trabajo se procesan de forma secuencial o con parale-

[1]DOI: https://doi.org/10.1007/978-3-031-69583-4_23
[2]GitHub: https://github.com/cloudmedialab-uv/k8s-work-queue-video-coding

lismo limitado, lo que provoca obtener unos mayores tiempos de codificación y una mala utilización de recursos. La solución propuesta aprovecha las capacidades de contenerización y orquestación proporcionadas por Kubernetes, permitiendo que las tareas de encoding se dividan en segmentos independientes y se procesen de manera paralela. La arquitectura desarrollada se muestra en la Figura 4.1, donde se resume los componentes centrales y sus interacciones.

Esta arquitectura está diseñada para aprovechar la potencia de la aceleración GPU dentro de un marco Cloud-Native desplegado sobre un cluster Kubernetes con recursos GPU integrados. En el núcleo del sistema se encuentran contenedores especializados para encoding, diseñados para manejar dos modos de operación, uno utilizando encoding tradicional basado en CPU (*libx264*), y otro aprovechando la aceleración GPU (*h264_nvenc*). Estos contenedores incluyen versiones optimizadas de FFmpeg y están configurados para ejecutarse como *pods* independientes dentro del cluster.

Un componente clave del sistema es el uso de RabbitMQ, *message broker*, que orquesta la distribución de tareas de codificación de vídeo. Cuando un nuevo segmento de vídeo está disponible, la URL donde se encuentra se encapsula en un mensaje junto con sus metadatos (como resolución, bitrate objetivo y parámetros de codificación) y se envía a un *exchange* que tiene dos colas asociadas, una para tareas de codificación sobre CPU y la otra para tareas de codificación sobre GPU. Los *workers* de codificación se suscriben a esta cola y procesan las tareas en función de la disponibilidad de sus recursos. Este diseño permite un *pipeline* de procesamiento al-

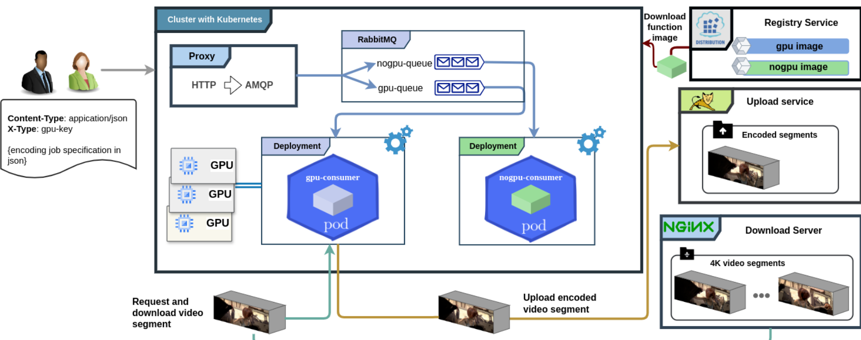


Figura 4.1: Arquitectura basada en Kubernetes para codificación de vídeo paralela

tamente desacoplado y escalable, donde los trabajos de codificación pueden distribuirse entre múltiples nodos.

Para aprovechar eficientemente los recursos GPU, el sistema emplea el soporte nativo de Kubernetes para la asignacion de GPUs mediante el *Nvidia Operator*[3]. Cada pod acelerado por GPU solicita una GPU dentro del *pool* de GPUs disponibles. La arquitectura también admite escalado dinámico, a medida que la carga de trabajo aumenta, se crean automáticamente más workers, permitiendo al sistema procesar múltiples segmentos de vídeo de forma concurrente. Esta asignación dinámica minimiza los tiempos muertos y la utilización de recursos del sistema.

En los experimentos se utilizan un conjunto de cuatro vídeos en resolución 4K, la implementación acelerada por GPU (*h264_nvenc*) mostró una reducción significativa en el tiempo de codificación en comparación con la versión basada únicamente en CPU (*libx264*). En promedio, la solución habilitada por GPU redujo el tiempo de codificación aproximadamente en un 15.98 % respecto a la configuración base. Además, al paralelizar tareas de codificación en múltiples *pods*, la arquitectura consiguió acelerar el tiempo de codificación en un factor de 3.29 en comparación con el enfoque basado en CPU. Estas mejoras fueron validadas además mediante el análisis de métricas de utilización de recursos, donde los *pods* acelerados por GPU mantuvieron menores cargas de CPU y memoria, indicando una delegación más eficiente de tareas computacionales hacia la GPU.

Esta contribución valida el funcionamiento eficiente de una cola de mensajes gestionada en Kubernetes mediante RabbitMQ para la codificación distribuida de vídeo. Además, esta arquitectura ha sido validada en un entorno real para gestionar 200.000 eventos de trabajos de codificación de segmentos de vídeos necesarios para generar un dataset [62], lo que avala su escalabilidad y robustez en aplicaciones de procesamiento masivo de vídeo. Dados estos resultados, se quiere explorar la adopción de un enfoque Serverless que delega la gestión de la mensajería al framework nativo del entorno, permitiendo evaluar la eficacia de la codificación distribuida mediante funciones Serverless utilizando otro códec de alta eficiencia.

[3] URL: https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/index.html

4.2. Aportación: Towards GPU-Enabled Serverless Cloud Edge Platforms for Accelerating HEVC Video Coding

Esta aportación (ver capitulo 8) propone un novedoso framework dirigido por eventos (event-driven) construido sobre la plataforma Serverless Knative y Kubernetes, que gestiona dinámicamente las cargas de trabajo de codificación. Al desacoplar las funciones de codificación de la infraestructura subyacente, esta solución logra un escalado rápido, y una utilización eficiente de recursos con baja latencia. El uso de GPUs, integradas en Kubernetes mediante PCI passthrough y gestionadas por el Nvidia Operator, permite acelerar significativamente las tareas de codificación HEVC mediante el códec Nvidia *hevc_nvenc*. En contraste, el encoder tradicional x265, que depende exclusivamente de CPUs, presenta una mayor sobrecarga computacional. La motivación detrás de este trabajo es reducir esta brecha de rendimiento aprovechando el paralelismo inherente de las GPUs, disminuyendo así el tiempo de codificación y asegurando que las aplicaciones de live streaming cumplan con estrictos requisitos de latencia. Esta solución es esencial en entornos como el HAS donde hay que generar diferentes representaciones (resolución, bitrate, etc.) al codificar cada segmento de vídeo. La arquitectura desarrollada se muestra en la Figura 4.2, donde se resumen los componentes centrales y sus interacciones dentro del marco Serverless.

La arquitectura presentada en esta aportacion está diseñada en torno a un paradigma event-driven y Serverless que se integra fluidamente con infraestructuras Cloud-Edge. En su núcleo, el sistema se aprovecha de Knative para orquestar funciones sin estado, que son desplegadas como aplicaciones contenerizadas en Kubernetes. Estas funciones se activan mediante CloudEvents, asegurando que las tareas de codificación se inicien automáticamente cuando nuevos segmentos de vídeo están disponibles. Dos implementaciones diferentes del códec HEVC están encapsuladas dentro de estas funciones, una utilizando el códec GPU acelerado *hevc_nvenc* y otra utilizando el encoder *libx265* basado en CPU.

La arquitectura del sistema es modular y escalable. El flujo de trabajo de codificación comienza con la inyección de eventos de codificación de segmentos de vídeo, que se almacenan en un broker y son enviados a las funciones a través de un *trigger* (sirve para

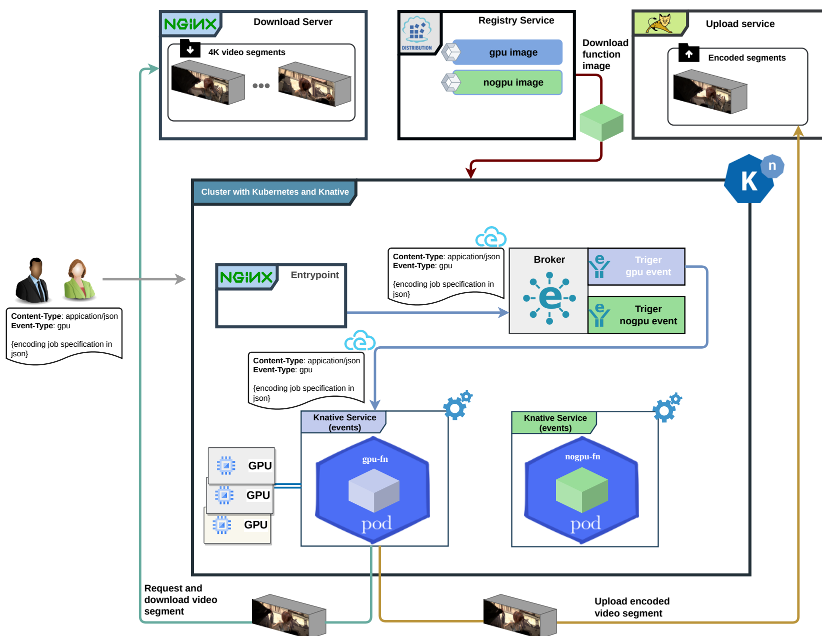


Figura 4.2: Un cluster con Kubernetes y Knative donde se despliégan las funciones. Un servidor de descarga con los videos, un servidor de carga para cargar los resultados y un registro de imágenes de contenedor están conectados al cluster.

enrutar automáticamente eventos de un broker hacia un servicio según criterios predefinidos). El uso de CloudEvents no solo permite la activación en tiempo real, sino que también proporciona una manera estandarizada de transmitir metadatos como requisitos de bitrate y configuraciones de resolución. Dentro de cada función, se despliegan procesos contenerizados que aprovechan la aceleración GPU mediante el soporte nativo de Kubernetes y el Nvidia Operator. Estos procesos están diseñados para minimizar el tiempo de arranque *coldstart*, tiempo inicial que tarda una función Serverless en cargarse en su entorno de ejecución y asignar los recursos necesarios para gestionar la petición. La latencia inicial es crítica en escenarios de alta demanda, ya que incluso pequeños retardos pueden afectar significativamente al tiempo total de respuesta. Adicionalmente, el diseño incorpora aislamiento de recursos asignando recursos específicos de GPU a cada réplica de función, garantizando un rendimiento predecible. Entre las decisiones relevantes de diseño se incluyen el uso de imágenes de contenedor optimizadas para baja latencia y la

implementación de técnicas de procesamiento paralelo para gestionar múltiples trabajos de codificación de manera concurrente.

La evaluación experimental se centró en cuantificar las mejoras de rendimiento de H.265/HEVC obtenidas al comparar diferentes configuraciones de máquinas virtuales y número de replicas de las funciones, contrastando la versión del códec acelerada por GPU con la versión tradicional basada únicamente en CPU. El entorno de pruebas consistió en un clúster Kubernetes con nodos dedicados equipados con GPU, en el que se implementaron funciones de codificación H.265 en dos configuraciones. Una utiliza la aceleración por GPU y otra se ejecuta sin dicha aceleración, manteniendo condiciones experimentales idénticas. Se midieron métricas de rendimiento como el tiempo de *coldstart*, el tiempo de codificación y la utilización de recursos.

Los experimentos de *coldstart* demostraron que, a pesar de presentar una sobracarga inicial ligeramente superior debido a la asignación de recursos GPU, no se observó una diferencia significativa entre las funciones habilitadas con GPU y aquellas sin GPU. Respecto a la gestión de recursos, se evaluaron tres configuraciones distintas (con recursos totales equivalentes):

- *Configuración 1 - Escenario a) y d):* Cuatro *slim* VMs con cuatro *slim* functions, donde cada función ejecuta una única tarea

- *Configuración 2 - Escenario b) y e):* Una *fat* VM que aloja cuatro *slim* functions, donde cada función ejecuta una única tarea

- *Configuración 3 - Escenario c) y f):* Una *fat* VM que ejecuta una *fat* function capaz de gestionar hasta cuatro tareas concurrentes

Estas configuraciones se pueden visualizar en la Figura 4.3. Las métricas recopiladas mediante PodInsights mostraron que las funciones habilitadas por GPU mantuvieron un uso estable de CPU y presentaron un menor consumo general de memoria, gracias a la delegación de cálculos intensivos hacia la GPU.

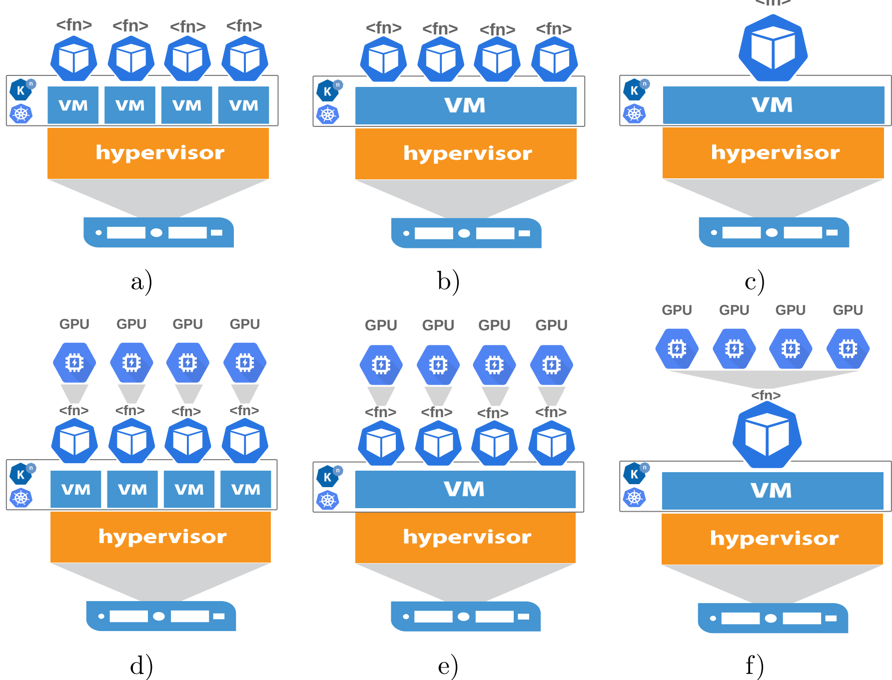


Figura 4.3: Escenarios de réplicas sin GPU: a) máquinas virtuales *slim* con réplicas *slim*, b) máquinas virtuales *fat* con réplicas *slim*, y c) máquinas virtuales *fat* con réplicas *fat*. Escenarios con GPU: d) máquinas virtuales *slim* con réplicas *slim*, e) máquinas virtuales *fat* con réplicas *slim*, y f) máquinas virtuales *fat* con réplicas *fat*.

Para ambos casos, funciones con GPU y funciones sin GPU, la configuración que mejor funcionó fue la que usaba una sola maquina virtual *fat* y desplegaba cuatro funciones *slim*, corresponientes a los escenarios b) y e). Obteniendo para el caso de codificación solo con CPU una reducción del tiempo de un 8,42 % y de un 15,21 % para el caso de la codificación con GPU. El tiempo de codificación para h265_nvenc fue significativamente menor en comparación con *libx265* logrando un tiempo de procesamiento de 8,33 veces menor. Además, al evaluar escenarios multiresolution con la mejor arquitectura en donde cada segmento de vídeo se codifica en múltiples representaciones, el despliegue con GPU redujo el tiempo total de codificación del segmento hasta 12.4 veces en comparación con la implementación basada únicamente en CPU.

En conjunto, los resultados experimentales validan la eficacia de

la arquitectura propuesta para lograr encoding de vídeo de baja latencia en entornos Cloud-Edge. El uso de aceleración GPU no solo reduce drásticamente los tiempos de codificación, sino que también permite al sistema manejar eficientemente cargas dinámicas de trabajo, algo crucial para aplicaciones de streaming adaptitivo sobre HTTP en tiempo real, como son las transmisiones de eventos en vivo.

Las contribuciones presentadas en la aportación *Towards GPU Enabled Serverless Cloud Edge Platforms for Accelerating HEVC Video Coding* representan un avance significativo en el campo del video encoding. Al integrar arquitecturas Serverless con aceleración GPU, este enfoque aborda desafíos críticos asociados a la codificación en tiempo real de contenidos de vídeo en alta resolución. La capacidad del sistema para escalar dinámicamente y reducir la latencia de codificación tiene un impacto particularmente significativo en aplicaciones de live streaming, donde una baja latencia y una alta calidad son fundamentales. Además, el diseño modular de la arquitectura permite una integración fluida con otras herramientas y servicios Cloud-Native, mejorando aún más su aplicabilidad en entornos multimedia modernos. Los resultados de este trabajo no solo demuestran la viabilidad técnica del video encoding Serverless habilitado por GPU, sino que también establecen un nuevo benchmark en rendimiento y escalabilidad para despliegues Cloud-Edge.

La arquitectura y los resultados experimentales de esta contribuciónn forman la piedra angular de una agenda de investigación más amplia sobre codificación de vídeo. El framework Serverless habilitado por GPU sirve como plataforma flexible y escalable que apoya contribuciones posteriores, tales como el streaming adaptativo inmersivo en 360 grados. Al establecer una base de rendimiento y demostrar los beneficios de la aceleración GPU, este trabajo abre el camino para la integración de funcionalidades adicionales. Estas técnicas son directamente aprovechadas *Cloud-Edge Architecture for 4K 360-Degree Video Encoding in FoV-based Live DASH*, que extiende la arquitectura hacia el streaming adaptativo Field-of-View para vídeos 360 grados en 4K. Colectivamente, estas contribuciones forman un ecosistema integrado que responde a las demandas crecientes del streaming multimedia moderno, permitiendo en última instancia una entrega de vídeo más eficiente, rentable y de alta calidad en entornos Cloud-Native y Cloud-Edge.

4.3. Exploración de la adaptación de la arquitectura anterior a la codificación de vídeo 360VR con resolución 4K para live DASH

Este trabajo[4] propone una novedosa arquitectura Cloud-Edge que integra codificación acelerada por GPU para live streaming adaptativo de vídeo 360 basado en el FoV. Aprovechando tecnologías Cloud-Native como Kubernetes y Knative, el sistema orquesta en tiempo real la segmentación temporal de vídeo y su posterior segmentación espacial (tiling). Esta arquitectura está diseñada para asignar dinámicamente los recursos donde más se necesitan, garantizando un procesamiento de baja latencia incluso en condiciones fluctuantes de red. La motivación detrás de esta contribución radica en la necesidad de entregar streams de vídeo 360 grados en resolución 4K que cumplan con estrictos requisitos de latencia manteniendo los tiempos totales de procesamiento muy por debajo del umbral crítico de 10 segundos, al mismo tiempo que se obtienen significativos ahorros en ancho de banda en la transmisión. Esta aportación aborda estas cuestiones adaptando el proceso de encoding para la distribución adaptativa dependiendo del FoV del usuario, optimizando así el bitrate del streaming y manteniendo un buen QoE para el usuario final.

La arquitectura Cloud-Edge propuesta para la codificación de vídeo 4K en 360 grados se basa en un framework híbrido que combina la escalabilidad del Cloud-Computing con los beneficios de proximidad del procesamiento edge. La Figura 4.4 muestra los componentes centrales de la arquitectura desarrollada y las relaciones de los distintos componentes que la forman.

En el núcleo del sistema hay una capa de orquestación Serverless basada en Knative sobre Kubernetes, que administra el ciclo de vida de funciones de encoding activadas por eventos en tiempo real. La arquitectura está diseñada para procesar streams entrantes de vídeo 360 4K en formato equirectangular mediante la segmentación inicial del vídeo en intervalos cortos generalmente segmentos de 0.5, 1 a 2 segundos usando un servicio dedicado de *Nginx Entry Point*. Creando eventos de segmentación de vídeo que se envían al *Watcher Component*, encargado de configurar los parámetros de tiling

[4] Trabajo aceptado, que será presentado en IEEE ISCAS 2025 a finales de mayo. Enlace a la publicación

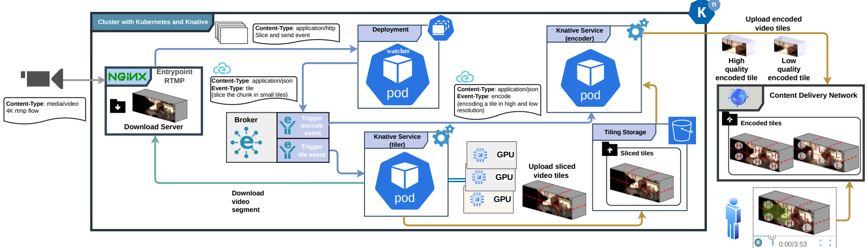


Figura 4.4: Arquitectura del sistema de la plataforma de transmisión y codificación de video en vivo 4K FoV de 360 grados Serverless.

espacial, tales como número de filas y columnas, según ajustes predefinidos adaptados al streaming adaptativo por FoV. Una vez concluida la segmentación, el Watcher desencadena CloudEvents que se propagan mediante un Knative Broker hacia un conjunto de *Tiling Workers*. Estos workers descargan cada segmento de vídeo y aplican división espacial para segmentar el frame completo de 360 grados en tiles más pequeños. El proceso de tiling está optimizado con aceleración GPU para manejar la intensidad computacional del slicing en tiempo real. Tras el tiling, los tiles en crudo se almacenan en un servicio *Tiling Storage*. El proceso de división del flujo de video en segmentos temporales y posteriormente segmentos espaciales se puede observar en la Figura 4.5.

Los tiles son procesados por un conjunto separado de *Encoding Workers*. Estos workers descargan los tiles en crudo del *Tiling Storage* y los codifican en representaciones de alta (H) y baja (L) calidad utilizando funciones de encoding basadas en CPU utilizando el códec H.264, como muestra la Figura 4.6. Una vez finalizado el encoding, los tiles se cargan en una CDN (Content Delivery Network) donde mediante un archivo MPD (Media Presentation Description)

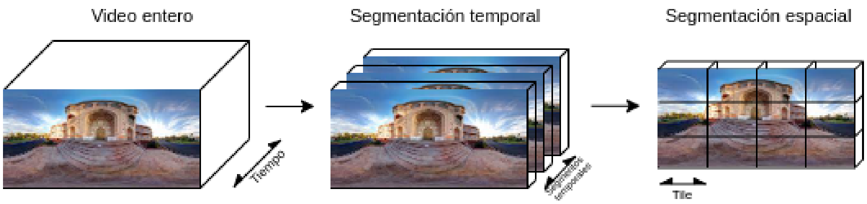


Figura 4.5: Proceso de división espacial y temporal del streaming inicial.

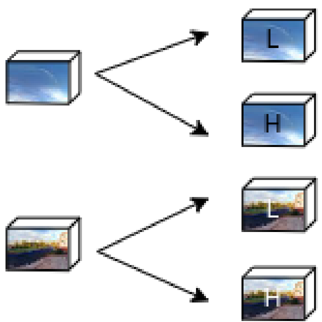


Figura 4.6: Proceso de codificación en alta resolución (H) y en baja resolución (L) de cada tile.

se disponen para la reproducción adaptativa del usuario. Todo este proceso de extremo a extremo desde segmentación, tiling, encoding y hasta distribución en la CDN se orquesta de manera totalmente automatizada. Esto permite que cada Player pueda seleccionar la representación más adecuada al FoV de quien visualiza el vídeo, minimizando así el bitrate del streaming. En este modelo, es el Player quien, en función del punto de vista del usuario, solicita en alta calidad los tiles correspondientes al campo de visión, y el resto en baja calidad como se observa en la Figura 4.7.

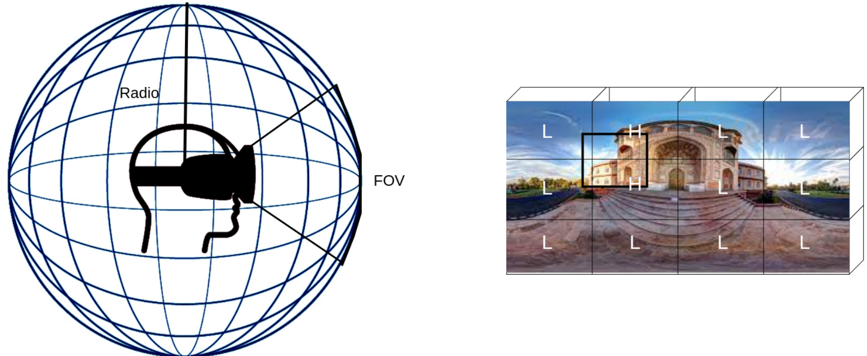


Figura 4.7: Representación de como el Player elige la calidad de los *tiles* a representar según el FoV.

Para validar la arquitectura Cloud-Edge propuesta, se realizó una evaluación experimental usando un conjunto representativo de vídeos 360VR en 4K. El entorno experimental consistió en un cluster Kubernetes con nodos tanto CPU only como GPU enabled. Se reco-

pilaron métricas clave de rendimiento, incluyendo latencia extremo a extremo, tiempo de codificación por segmento y reducción general del bitrate lograda mediante streaming adaptativo basado en FoV.

Los resultados experimentales demostraron que el sistema mantuvo los tiempos totales de procesamiento por debajo del umbral de 10 segundos requerido para live streaming. Los procesos acelerados por GPU de tiling contribuyeron significativamente a este rendimiento. Además, el mecanismo de adaptive bitrate streaming demostró reducir el consumo de bitrate hasta en un 77 % en el mejor de los casos y un ahorro del 31.11 % para el peor de los casos, comparado con un enfoque no adaptativo donde no se adapta la calidad del video al FoV.

Esta contribución se basa directamente en los principios arquitectónicos establecidos en la contribución *Towards GPU-Enabled Serverless Cloud Edge Platforms for Accelerating HEVC Video Coding*. Al enfocarse en la codificación de vídeo inmersivo en 360 grados con adaptación basada en FoV, este trabajo aborda un nicho crucial en los sistemas live DASH streaming. El enfoque Cloud-Edge no solo cumple con estrictos requisitos de latencia, sino que también reduce considerablemente el uso del bitrate mediante ajustes adaptativos de calidad. La exitosa integración de tiling y encoding acelerado por GPU dentro de un framework Serverless destaca el potencial para desplegar soluciones de streaming escalables y adaptativas en entornos Cloud-Edge. El impacto de este trabajo va más allá de mejorar el rendimiento de estos sistemas, sino que establece un nuevo marco para distribuir experiencias de vídeo inmersivas y de alta calidad adaptadas al campo de visión de cada usuario y a las condiciones de red, avanzando así el estado del arte en streaming de vídeo en vivo.

Capítulo 5

Conclusiones y trabajo futuro

Esta tesis doctoral ha aportado un enfoque unificado para abordar los desafíos en entornos Cloud-Native y Cloud-Edge mediante la integración de tres contribuciones principales. En primer lugar, se ha desarrollado K8Sidecar, una solución modular basada en el patrón *Operator* que permite la inyección dinámica de contenedores sidecar en microservicios y funciones Serverless en Kubernetes. Este mecanismo ha demostrado mejorar la flexibilidad y escalabilidad, permitiendo la integración de funcionalidades complementarias (como seguridad, monitoreo o gestión de datos) sin modificar la lógica central de la aplicación.

En segundo lugar, el análisis comparativo entre implementaciones de codificación de vídeo en CPU y GPU, enfocado en la calidad y el rendimiento (medidos a través de métricas como PSNR, VMAF y BD-Rate), ha permitido evidenciar que la aceleración mediante GPU puede reducir drásticamente los tiempos de codificación especialmente en el caso de codecs como AV1, aunque con algunas variaciones en la eficiencia de bitrate en el caso de HEVC. Estos hallazgos ofrecen una base sólida para la toma de decisiones en escenarios de streaming y videoconferencias, donde la latencia y la calidad son factores críticos.

Por último, se han presentado diversas arquitecturas que integran la aceleración por GPU en entornos Cloud-Native y Cloud-Edge para optimizar la codificación de vídeo. Estas soluciones combinan el escalado dinámico, la distribución paralela de tareas y la asignación eficiente de recursos, aprovechando plataformas Serverless como Knative y la gestión automatizada mediante operadores. La

integración de *workers* de codificación, que acceden directamente a GPUs a través de *PCI passthrough*, ha permitido alcanzar una codificación de baja latencia y alta capacidad de respuesta. Además, estas arquitecturas abarcan tanto *pipelines* de codificación de vídeo 4K multirresolución para aplicaciones de live streaming como sistemas de distribución inmersiva de vídeo 360VR en 4K basados en el Field-of-View, abriendo nuevas posibilidades para la entrega de contenido multimedia en entornos dinámicos.

A pesar de los avances logrados, existen diversas líneas de investigación que podrían complementar y ampliar el alcance de este trabajo:

- **Mejora en la modularidad y extensibilidad de K8Sidecar**: Ampliar la librería de sidecars disponibles y desarrollar un *Proxy Store* comunitario que permita a otros desarrolladores compartir y reutilizar componentes especializados, abarcando además protocolos de comunicación adicionales (por ejemplo, AMQP o MQTT).

- **Optimización de la gestión de recursos**: Integrar técnicas de inteligencia artificial o machine learning para predecir la carga y ajustar dinámicamente la asignación de recursos (tanto en los sidecars como en las funciones de encoding), lo que permitiría reducir aún más los tiempos de *coldstart* y optimizar el uso de GPUs en entornos Cloud-Edge.

- **Extensión a nuevos codecs y escenarios inmersivos**: Se propone explorar la integración de implementaciones GPU para codecs emergentes (como AV1) y la aplicación de estas técnicas en el procesamiento y streaming de contenido 360° con adaptación basada en el FoV (Field of View), para mejorar la experiencia en entornos de realidad virtual.

- **Exploración de estrategias de tiling**: Investigar técnicas de segmentación adaptativas para dividir el vídeo en segmentos temporales según la complejidad del contenido y el FoV del usuario, con el fin de reducir la latencia y mejorar la eficiencia en la codificación y el streaming.

- **Desarrollar métricas de calidad basadas en el FoV**: Crear nuevas métricas que integren el análisis del campo visual combinándolo con la evaluación diferencial de la calidad en los distintos tiles del vídeo. De esta forma, se podrá medir de manera precisa la calidad real percibida por el usuario en función de la calidad asignada a cada área.
- **Explorar estrategias para *green video streaming*** Optimización energética en el procesamiento y distribución del contenido multimedia, lo que podría contribuir a reducir la huella de carbono y los costes operativos, estableciendo un marco más sostenible para futuras infraestructuras de streaming.

En resumen, esta tesis sienta las bases para futuras investigaciones en vídeo streaming y codificación en entornos Cloud-Native y Cloud-Edge. Cabe destacar que algunas de estas líneas, como la exploración de estrategias de tiling para la optimización de recursos y el desarrollo de métricas basadas en el FoV, ya se están explorando en nuestro grupo de investigación, lo que asegura avances prometedores hacia soluciones cada vez más eficientes y adaptativas.

Capítulo 6

K8sidecar: A Modular Kubernetes Chain of Sidecar Proxies for Microservices and Serverless Architectures

Software: Practice and Experience

WILEY

RESEARCH ARTICLE

K8sidecar: A Modular Kubernetes Chain of Sidecar Proxies for Microservices and Serverless Architectures

Andoni Salcedo-Navarro | Miguel Garcia-Pineda | Juan Gutiérrez-Aguado

Departamento de Informática, Universitat de València, Burjassot (Valencia), Spain

Correspondence: Juan Gutiérrez-Aguado (juan.gutierrez@uv.es)

Received: 12 November 2024 | **Revised:** 12 February 2025 | **Accepted:** 19 March 2025

Funding: This study was supported by grant PID2021-126209OB-I00 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A Way of Making Europe".

Keywords: kubernetes | microservices | operator pattern | proxy sidecar | serverless computing

ABSTRACT

Background: Modern microservice architectures demand not only modularity, scalability, and maintainability but also adaptation to dynamic requirements. For applications deployed on Kubernetes, sidecar proxies have been used to separate the operational features (such as security, networking, or monitoring) from the application business logic by intercepting traffic to and from microservices or functions in serverless platforms. Existing proxy sidecar implementations such as Envoy offer these operational features as chains of logic that cannot be composed at deployment time.

Objective: This work introduces K8Sidecar, which leverages the operator pattern to dynamically integrate a chain of proxy sidecar containers into microservices or serverless architectures deployed on top of Kubernetes.

Methods: The study presents the architecture of the software and describes the Java and Go libraries provided to develop new proxy sidecars.

Results: We provide illustrative examples and evaluate the impact of the number of injected sidecars on cold-start and latency and compare them with Envoy.

Conclusion: Results show that for up to 5 sidecars, the chain-of-sidecars approach (especially using the Go library) remains reasonably close to Envoy in terms of both cold start and latency.

1 | Introduction

The last decade has been marked by a rapid transition from monolithic architectures to microservices and ephemeral replicas of functions. While this shift provides modular, scalable, and fault-tolerant systems, it also raises a new set of challenges. There are two opposing requirements, on the one hand, individual services must be lightweight and single-purposed. On the other hand, they must integrate functionalities like logging, monitoring, or security, which are an integral part of achieving maintainable and observable systems [1].

Among the various patterns utilized in cloud-native architectures, the proxy sidecar pattern has been identified as a potential solution to this problem [2]. The proxy sidecar is a design pattern intended for containers that cooperate closely and that are deployed on the same pod [3]. The sidecar proxy pattern has been widely adopted to implement a service mesh that provides unified control and observability with fine granularity. Acting as complementary auxiliary containers in the data plane, proxy sidecars allow functionalities to be abstracted away from the application, ensuring that microservices adhere to the single responsibility principle. However, a lack of a dynamic and

© 2025 John Wiley & Sons Ltd.

intuitive method for the chaining of sidecars at deployment time has posed a challenge for their use in Kubernetes or Serverless architectures [4].

A recent study on monitoring tools for DevOps [5] identified key research challenges. These include assessing the runtime overhead of monitoring-based systems and assessing the risk of introducing bugs through instrumentation code in microservices. Therefore, a good practice could be to externalize the monitoring code to a proxy sidecar that can be tested in isolation and its runtime impact can be clearly measured.

Envoy [6] is one of the most known and widely used proxy sidecar implementations. Envoy proxy intercepts traffic enabling security policies, rate limiting, and other transformations. Besides, it offers dynamic routing, load balancing, service discovery, or protocol support. Envoy can be extended by adding custom filters for specific tasks. However, these filters must be developed and integrated into a filter chain during the development phase; therefore, it is not possible to combine filters or add more filters dynamically in a chain when deploying an application.

The proposed solution addresses the challenge of seamlessly integrating and chaining proxy sidecars into Kubernetes environments [7] by dynamically injecting them by using the operator pattern. This approach bridges the gap between the necessity of adding extra common functionalities and the goal of maintaining the simplicity of microservices. Manual integration of sidecars can lead to inconsistencies and substantial maintenance overheads. By automating this process, our solution offers a consistent, scalable, and modular approach to sidecar chaining.

From the user's perspective, the proposed solution provides a smooth experience. Users can implement custom sidecars such as logging, monitoring, authentication, data management, adapters, etc. by using our libraries. These sidecars can be tested and optimized in isolation. The chain of sidecars, or filter, is deployed as a custom resource, and when a deployment or a serverless function is labeled to use the filter, the system automatically injects the required sidecars into the respective microservices or functions. This abstraction not only reduces the need for manual intervention, but also ensures that the primary focus remains on developing the core service logic.

The main contributions of this proposal are:

- *On-the-fly configuration*: Enables the addition, removal, or update of a chain of proxy sidecars without the need to rebuild or redeploy the core microservice or function. This functionality supports rapid iteration and deployment cycles, significantly reducing downtime and increasing development agility.
- *Custom Resource Definition (CRD) for Sidecars*: Leverages Kubernetes CRDs for dynamic chain configurations, enabling developers to define individual sidecar properties such as image, name, environment variables, or volumes. This allows for customized functionality and behavior within the microservices architecture.
- *Label-Based Injection Control*: Utilizes Kubernetes labels to control the sidecar injection into specific deployments, providing granular control over which microservices are paired with certain sidecars, thus enhancing the architecture's modularity and flexibility.
- *Priority-based injection*: Allows developers to assign priorities to sidecars at deployment time, dictating the order in which they are injected and executed. This ensures that functionalities, such as security or logging, are processed in the desired order, optimizing the flow of requests or dealing with dependencies through the chain.
- *Environment variables for configuration*: Allows the injection of Kubernetes environment variables into sidecars, enabling them to adjust seamlessly to different deployment environments. This feature is useful for configuring sidecars to operate under various conditions without modifying their core logic.
- *Golang and Java Libraries*: Offers dedicated libraries for Golang and Java, simplifying the development of new custom sidecar functionalities by abstracting sidecar integration complexities. This allows developers to focus on implementing the specific functionality of the sidecar.
- *Simplified Middleware Function Definition*: The libraries provide interfaces for defining middleware functions that manage HTTP requests and CloudEvents, accommodating both traditional web applications and event-driven architectures. This versatility supports a wide range of use cases and application scenarios.
- *Knative Support*: Enhances serverless computing models managed by Knative, facilitating sidecar injection in serverless frameworks. This feature enables developers to utilize dynamic sidecar management benefits in cloud-native applications that scale automatically based on demand.
- *Event-Driven Architecture Compatibility*: With CloudEvents support, the software facilitates the development of event-driven microservices, ensuring the integration among decoupled services.

The remainder of the paper is organized as follows: Section 2 provides an overview of the technical stack used in this study, including Kubernetes, the Function-as-a-Service model, and the Operator Pattern. Section 3 presents a high-level overview of the proposed solution. Section 4 offers a detailed description of the various components and libraries available for filter development. Validation using Kubernetes resources of type `Deployment` and Knative resources of type `Service` (for serverless functions), performance experiments, and a comparison with related work are discussed in Section 5. Finally, the conclusions and future work are presented in Section 6.

2 | Background

This section briefly presents the technological stack upon which the proposal has been developed.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitaet De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

2.1 | Kubernetes

Kubernetes [7] is an open-source container orchestration platform that has emerged as the *de facto* standard for managing and orchestrating containerized workloads. Initially designed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes provides a powerful and flexible way to automate deployments, scale applications, and ensure high availability through self-healing mechanisms. At a high level, Kubernetes offers a REST API to manage resources such as nodes, pods, services, volumes, deployments, replicas, etc., that can be used to run containerized applications. Nodes represent the physical or virtual machines where containers can be allocated. Pods are units that encapsulate one or more containers, which are deployed together and share the same network and storage. Services provide stable IP addresses and DNS names for pods. Deployments define the desired state of a group of replicas, with each replica running a pod, ensuring redundancy and availability. Kubernetes also provides advanced features such as rolling updates, blue-green deployments, and horizontal scaling.

2.2 | Function-As-A-Service

Function as a Service (FaaS), a key component of serverless computing, represents a new approach to building and running applications in the cloud [8]. In traditional architectures, developers are responsible for managing the infrastructure where the service will be deployed. This includes the management of elasticity to adapt the underlying infrastructure to the demand. With FaaS, instead of managing servers or containers, developers focus solely on writing and deploying individual functions that respond to specific events or requests, transferring the responsibility of the allocation and scaling of these functions to cloud providers.

Cloud providers have widely adopted this technology: AWS Lambda, Google Cloud Functions, Azure Functions, and others manage the underlying infrastructure, automatically provisioning resources and executing functions in response to events from other services or user requests. The FaaS platform will handle the scaling required to run that function to deal with the demand. For on-premises, different serverless platforms can be deployed on top of Kubernetes such as Knative or OpenWisk.

Overall, FaaS offers a granular pay-per-use model and simplifies application design, development, and deployment by abstracting away the underlying infrastructure, allowing developers to focus on writing efficient, event-driven code.

2.3 | Operator Pattern

In Kubernetes, the Operator pattern is a powerful approach to manage and deploy complex applications [9]. An Operator allows the definition of new custom resources and controllers for managing specific applications or workloads, extending the set of resources provided by Kubernetes. The Operator pattern provides a standard way to create, configure, and manage instances of these applications. This can include starting and stopping containers, scaling resources, handling backups and restores, and even automatically managing rolling updates and rollbacks.

By leveraging the Operator pattern, the need for manual intervention in complex environments is reduced. Additionally, Operators can be shared within organizations or the community to enable consistent deployment and management of applications across multiple clusters.

3 | High-Level Overview of the Chain of Sidecars Proposal

The architecture rests upon its modular and dynamic design. It is built around the primary principle of *on-the-fly* sidecar injection. Instead of incorporating all non-core logic within the main service, sidecars are treated as peripherical components that can be attached at deployment time to other resources.

The proposed solution integrates within the Kubernetes ecosystem through the development of an operator that consists of a CRD and a controller. A new `Filter` CRD has been defined, which allows the specification of an array of proxy sidecars (each filter can be tailored to a specific use case with the sidecars needed). Each sidecar within this array is distinctly identified by a unique name and a specific container image. Additionally, it is possible to define optional properties for each sidecar, such as environment variables, volumes, or priority settings. Conversely, the controller is responsible for creating a Mutating Admission Controller and deploying its associated webhook, which is tailored to a particular `Filter`. This process, illustrated in Figure 1, allows the specification of sidecar configurations by merely creating a Custom Resource of the type `Filter`. The dynamic injection of the sidecars specified in a `Filter` can be achieved by referencing the filter name using Kubernetes labels.

When a Kubernetes resource that makes use of a `Filter` is deployed, the associated webhook modifies the request to inject and configure the array of containers into the resource. The mutated request is finally processed by the kubelet on a node, which creates the pod containing the sidecars along with the application, as illustrated in Figure 2.

4 | Implementation Details and Usage

In this section we provide a detailed description of the architecture and functionalities of our software, emphasizing its design and capabilities of sidecar integration.

4.1 | Filter Operator: Custom Resource Definition and Controller Components

The CRD `Filter` allows the specification of an array of sidecars, where each sidecar has a name and uses a container image.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitaet De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

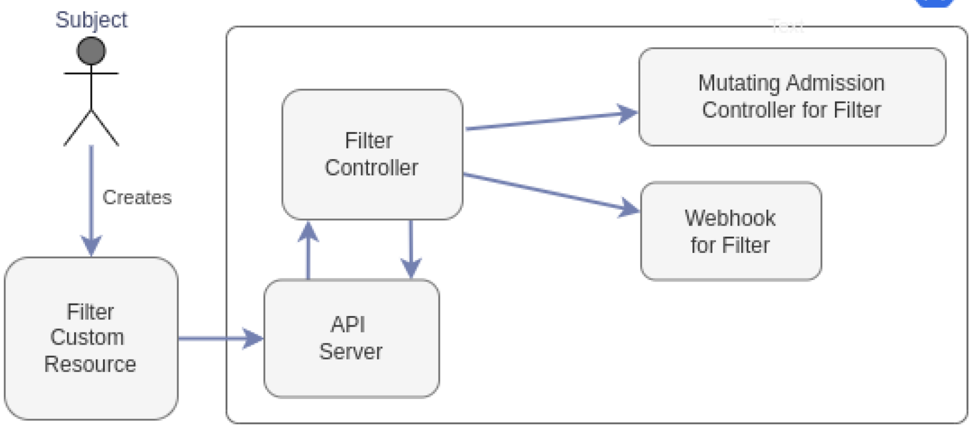


FIGURE 1 | The operator controller is notified when a new Filter is created. A new Mutating Admission Controller and its webhook are automatically created for that Filter.

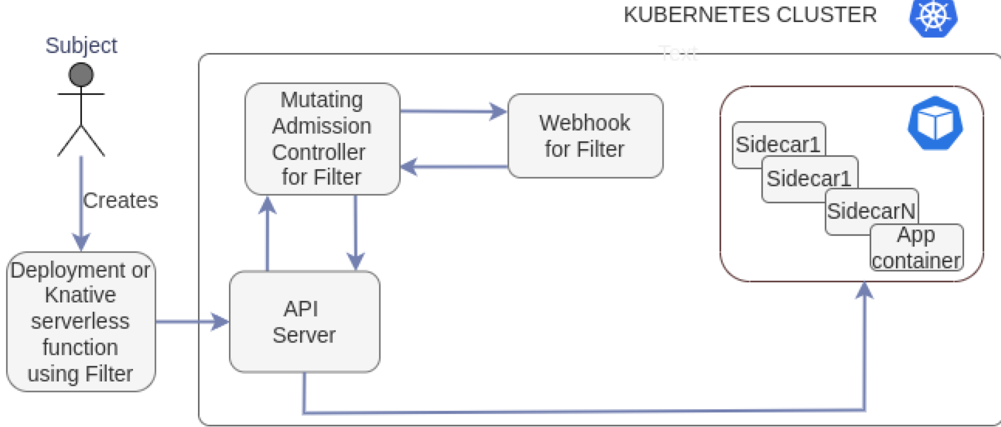


FIGURE 2 | When a resource that makes use of the Filter is deployed the webhook injects the sidecars specified in the Filter.

Additionally, each sidecar can have a priority and environment variables.

Listing 1 shows the allowed fields of each sidecar in a `Filter`, where:

- `name`: serves as the unique identifier of the sidecar within the filter. It is crucial for managing and referencing individual sidecars.
- `image`: represents the container image with the implementation of the sidecar, defining its operational environment and capabilities.
- `priority`: an optional attribute that facilitates the sorting of sidecars by assigning a priority level ranging from 0 to 255. This feature allows users to determine the order in which requests are processed by different sidecars. By default, this priority is set to 0, indicating the highest priority level.
- `Env`: this option enables the injection of environment variables into the sidecar container, providing a means to adapt the behavior of the sidecars to each deployment.
- `Vol`: volume specification for a sidecar. Each sidecar can mount its own volume.

LISTING 1: Allowed fields in the declaration of each sidecar in a Filter.

```
properties:
    name:
        type: string
    image:
        type: string
    priority:
        type: integer
        format: int8
        nullable: true
    env:
        type: Env
    vol:
        type: Volume
```


1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitat De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

The other key component of the operator is a custom *controller* that receives the request when a new `Filter` is created or deleted. The controller, implemented in Golang[1], executes a control loop to synchronize the desired and actual states of the cluster. If a `Filter` is created, the controller creates and registers a *Mutating Admission Controller*, whose webhook will perform the injection of the sidecars if the deployed resource has a label with the name of the filter.

The steps performed by the webhook when a resource using the `Filter` is deployed are shown in Algorithm 2. The webhook receives the request and returns a `JSONPatch` with all the changes made to the original resource in the request. The priority in the sidecars determines the order of injection, ensuring that sidecars with higher priority are closer to the application container. The container with the application gets the maximum priority so it will be the last one in the generated chain. Besides, an `emptyDir` volume for temporary data sharing is created and mounted in all the containers as illustrated in Figure 3. Sharing a volume across all the containers in the pod can be useful in some scenarios. For instance, downloading data required by the application or uploading data produced can be managed by a sidecar.

ALGORITHM 2 | Algorithm used to inject sidecars specified in a Filter.

Input: resources;
Input: filter_name from environment
Input: filter_data from environment
  if resource has label matching the filter_name **then**
    $port$ = user container port from annotation;
    $mount_dir$ = mount point from annotation
    $list$ = containers in resource
    $sidecars$ = sidecars in filter
    set_pritority(user_container,MAX_PRIORITY)
    $sorted_list$ = sort_by_priority(sidecars.append(list))
    $patch$ = [ ]
    $patch$.append_operation(Delete list of containers);
    $patch$.append_operation(Add emptydir volume);
    while C in $sorted_list$ **do**
      $patch$.append_operation(Add C to containers array);
      $patch$.append_operation(Assign $port$ to C);
      $patch$.append_operation(Mount volume in $mount_dir$)
      $port \leftarrow port + 1$;
    end while
  end if
Output: response with $patch$;

4.2 | Libraries to Develop New Proxy Sidecars

The intended behavior for each proxy sidecar in a chain is as follows: when the sidecar receives an HTTP request it performs some action, then sends the HTTP request to the next sidecar (or the application container if the sidecar is the last one in the chain), and finally can perform some action when the callee returns. This can be implemented in any language capable of dealing with HTTP requests.

By providing libraries that simplify the process of defining the middleware function that handles requests, the development of new proxy sidecars is facilitated. We have implemented libraries in Golang and Java, following a common specification that leverages the specific features and mechanisms of each language. The implementation of the libraries follows the idea of filter chains in `Servlets`.

The libraries handle standard HTTP requests and `CloudEvents`, offering two types of functions `TriFunction` and `QuaFunction` respectively. The `TriFunction` is designed to handle standard HTTP requests, whereas the `QuaFunction` is designed to process CloudEvents, extending the applicability to event-driven architectures.

Although we provide ready-to-use libraries in both Java and Go, developers are not constrained to these languages. Indeed, creating a functional sidecar in any language compatible with the system is straightforward: the sidecar only needs to listen for HTTP requests on the port defined by the `PPORT` environment variable and then, once any intended processing is completed, forward the request to `PPORT + 1` to pass it to the next filter in the chain. This design ensures that users can adapt the pattern to their existing technology stacks and leverage the wide range of libraries or dependencies available in different ecosystems.

4.2.1 | Golang Library

The Golang library developed[2] uses the standard HTTP library due to its performance, wide adoption, and extensive support within the Golang ecosystem. Listing 3 provides the definitions of `TriFunction` and `QuaFunction` for a Golang implementation.

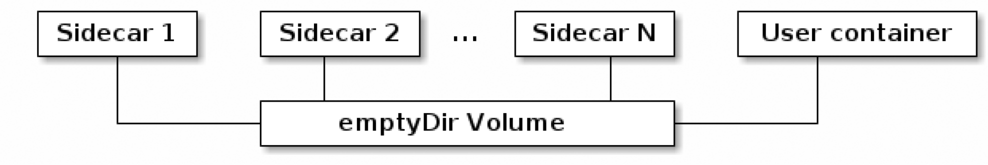


FIGURE 3 | The webhook creates and mounts an `emptyDir` volume shared by all the sidecars and the user container. This allows the sharing of data among all the containers in the pod.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitat De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

LISTING 3: Interfaces defined in the Golang library.

```
// TriFunction is a type representing a function that takes in
// an HTTP request, an HTTP response writer, and a FilterChain.
type TriFunction func(req *http.Request,
    res http.ResponseWriter, chain *FilterChain)

// QuaFunction is a type representing a function that takes in an HTTP
// request, an HTTP response writer, a cloud event, and a FilterChain.
type QuaFunction func(*http.Request,
    http.ResponseWriter, cloudevents.Event, *FilterChain)
```

4.2.2 | Java Library

The provided Java library[3] conforms to the Servlet specification, a widely accepted standard for web application development in the Java ecosystem. A class `SidecarFilter` with the method `accept` is implemented in the library (this class uses Jetty to start an HTTP server). The `accept` method expects an object that can be referenced by one of the functional interfaces (so lambda functions can be used) shown in Listing 4.

LISTING 4: Functional interfaces defined in the Java library. The actions to be performed in the sidecar are implemented in the accept method.

```
/**
 * A functional interface representing a function that accepts three arguments.
 * For a sidecar the first argument is a HttpServletRequest,
 * the second argument is a HttpServletResponse, and the third one
 * is of type FilterChain to pass the request to the next container in the chain
 */
@FunctionalInterface
public interface TriFunction<T, U, V> {
  void accept(T t, U u, V v);
}
/**
 * A functional interface representing a function that accepts four arguments.
 * For a sidecar the first argument is a HttpServletRequest,
 * the second argument is a HttpServletResponse, the third one
 * is of type CloudEvent, and the last one is of type FilterChain
 * to pass the request to the next container in the chain
 */
@FunctionalInterface
public interface QuaFunction<T, U, V, R> {
  void accept(T t, U u, V v, R r);
}
```

As an illustrative example of usage, Listing 5 shows the use of the `QuaFunction` to develop a sidecar using the provided `SidecarFilter` class.

LISTING 5: Sample usage of QuaFunction to develop a sidecar.

```
// SidecarFilter is provided by the Java library
SidecarFilter server = new SidecarFilter((req, res, event, chain) -> {
      try {
        // Use req and event to obtain information
        // Do some action before passing the request to the next element in the chain
```


1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitaet De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
        chain.next();
        // Do some action before returning to the previous element in the chain
      } catch (Exception e) {
      }
    });
```

4.3 | Custom Resource to Define a Filter With a Chain of Proxy Sidecars

Let's assume that the user has developed a set of sidecars $\{S_1, S_2, \dots, S_n\}$ using the provided libraries and wants to apply some of them to a Kubernetes deployment. A *Custom Resource* of type `Filter`, that conforms to the Custom Resource Definition explained in Subsection 4.1, must be specified and deployed into Kubernetes. An example of deployment is shown in Listing 6.

LISTING 6: YAML specification of a new Filter. Each Filter defines an array of proxy sidecars and the priority defines the order.

```yaml
kind: Filter
metadata:
  name: <filtername>
spec:
  sidecars:
    - image: <sidecar-image1:tag>
      name: sidecar_name1
      priority: 1
    - image: <sidecar-image2:tag>
      name: sidecar_name2
      priority: 2
```

As mentioned previously, when this custom resource is created, the controller dynamically creates and configures a Mutating Admission Controller with its webhook for this filter. This is illustrated in Figure 4.

4.4 | Using the Filter in a Deployment or a Serverless Function

The association of a filter with a deployment or a serverless function is achieved by using labels. In our system, the name of the filter serves as the key of the label, and the value is set to the string `sidecar`. This approach permits the injection of one or more filters into the deployment, based on the specified criteria. The webhook must know the port of the user container, for this purpose, a new annotation is used. This annotation holds the name of the environment variable that stores the port. This design allows accommodating the sidecar to existing deployments that use an arbitrary name to pass the port to the application.

Listing 7 illustrates de use of a filter in a Kubernetes deployment.

LISTING 7: Sample usage of a filter in a Kubernetes deployment.

```yaml
apiVersion: apps
kind: Deployment
metadata:
  name: app-deployment
  annotations:
    k8sidecar.port.env-name: "APP_PORT"    # default: PORT
    k8sidecar.volume.mount-dir: "/shared-data"    # default: /shared
  labels:
    app: app-label
    <filtername>: "sidecar" # Enable injection of sidecars defined in the filter
spec:
  selector:
    matchLabels:
      app: app-label
  template:
```


1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitaet De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```yaml
spec:
  containers:
  - name: <name>
    image: <user-container-image:tag>
    env:
      - name: APP_PORT
        value: <port>
    ports:
    - containerPort: <port>
```

When this resource is deployed in Kubernetes, the webhook of the mutating admission controller is activated to inject the sidecars. This process is shown in Figure 5.

Listing 8 illustrates de use of a filter in a Knative serverless function. As can be seen, the syntax to use the filter is identical to that used for Kubernetes deployments.

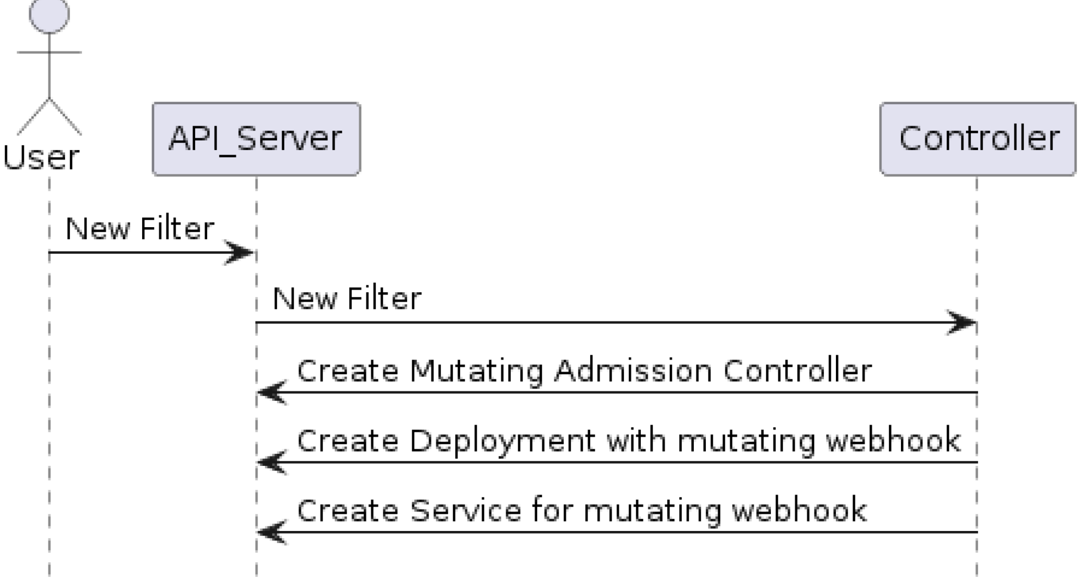


FIGURE 4 | When the user creates a new Filter, the controller automatically creates a new Mutating Admission Controller with its webhook.

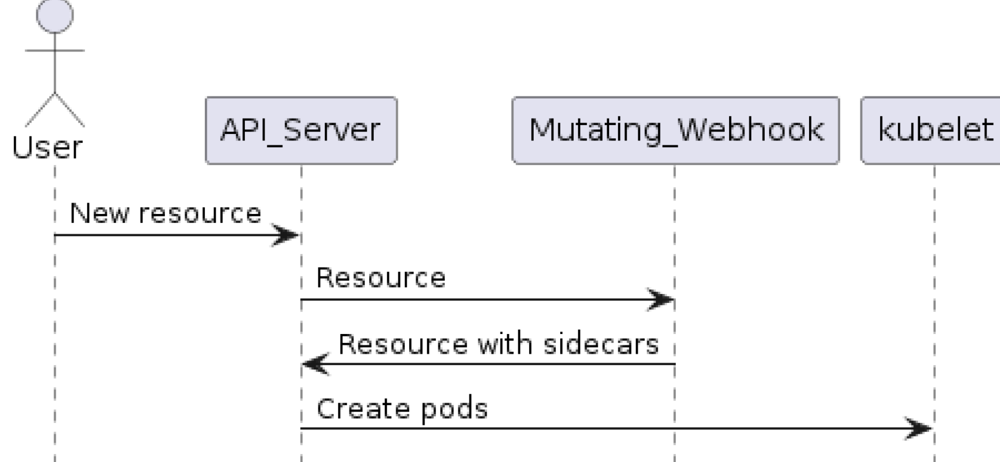


FIGURE 5 | When the user creates a new Filter, the controller automatically creates a new Mutating Admission Controller with its webhook.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitaet De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

LISTING 8: Sample usage of a filter in a Knative function.

```
kind: Service
apiVersion: serving.knative.dev/v1
metadata:
  name: serverless-fn
  annotations:
    k8sidecar.port.env-name: "APP_PORT"    # default: PORT
    k8sidecar.volume.mount-dir: "/shared-data"    # default: /shared
  labels:
    <filtername>: "sidecar"
spec:
  template:
    metadata:
      name: serverless-fn-v2
    spec:
      containers:
        - name: <name>
          image: <user-container-image:tag>
          env:
            - name: APP_PORT
              value: <port>
          ports:
          - containerPort: <port>
```

5 | Validation and Performance Evaluation

5.1 | Validation

This section provides a comprehensive overview of the diverse range of sidecars that have been implemented, deployed, and subjected to rigorous testing on our platform. It encompasses a multitude of essential functionalities, including security, data management, fault tolerance, and monitoring. Each sidecar is designed to address a specific operational need without modifying the core application code, thereby enhancing the system's modular architecture.

- *Data Management*: Facilitates the efficient handling of data, enabling both the downloading and uploading of data consumed or produced by the application. This sidecar ensures data synchronization across distributed components.
- *CloudEvent Generation*: Converts internal system events into standardized CloudEvents, thereby promoting interoperability across different cloud-based services and platforms.
- *Rate limiting*: Limits the number of requests to a server[4].
- *Logging*: Captures and stores logs systematically, aiding in troubleshooting and maintaining the system's operational integrity over time[5].
- *Token Authorization*: Ensures secure service-to-service communication through token checks, enforcing authentication and authorization[6].
- *Event Adapter for RabbitMQ*: This sidecar acts as a middleware that subscribes to RabbitMQ queues to retrieve messages, which are then transformed into CloudEvents and forwarded as HTTP requests to subsequent services in the chain. This decouples the application from specific message queue technologies.
- *GPU Usage Monitoring*: Monitors and logs GPU usage statistics, providing insights for resource optimization and system performance tuning.

To illustrate the practical application of these sidecars, consider a scenario where an application is developed to consume CloudEvents wrapped in HTTP requests. Traditionally, if the event data is stored in RabbitMQ, adapting the application to read directly from RabbitMQ would necessitate substantial code modifications. However, with the implementation of *Event Adapter for RabbitMQ* sidecar, there is no need to alter the application's code. The sidecar integrates with RabbitMQ, fetching messages, converting them into the appropriate format, and ensuring they are correctly injected into the application flow. This not only simplifies the architecture but also enhances its adaptability and scalability, demonstrating the sidecar's utility in practical cloud-native environments. This integration process is illustrated in the diagram shown in Figure 6.

5.2 | Performance Evaluation

This section describes the testbed and the experiments performed. The first experiment evaluates the cold start time in serverless platforms. Cold start refers to the time taken for a service to become available to handle requests when the number of replicas is zero. We measured this time as the elapsed time from the moment a request was sent to the instant when the user application received the request. This experiment has

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitaet De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

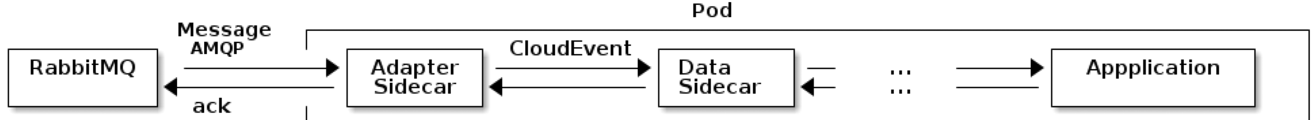


FIGURE 6 | Workflow diagram of the RabbitMQ adapter sidecar integration.

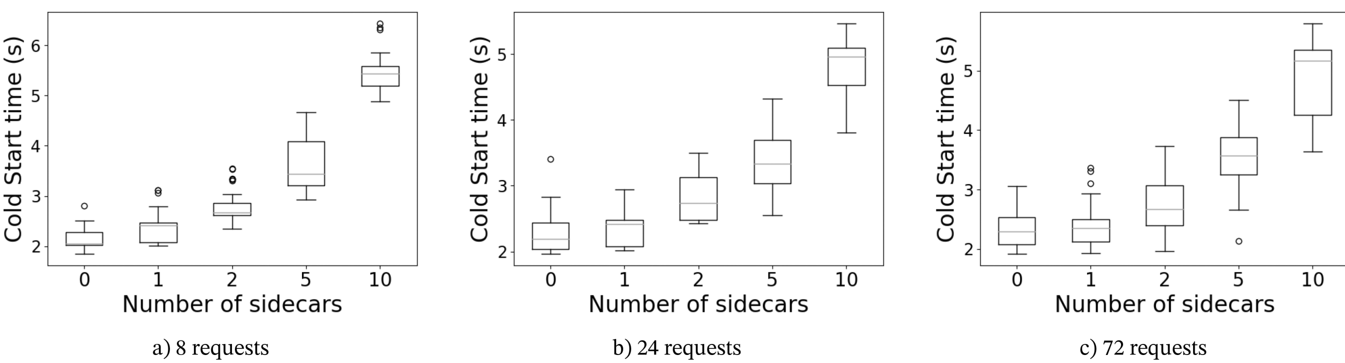


FIGURE 7 | Cold-start comparison depending on the number of proxy sidecars, implemented using Go, per pod. In the more demanding scenario (72 requests and 10 sidecars) each worker node has to start 108 containers (each pod has the queue-proxy, 10 sidecars, and the user container).

been performed with small and big sidecars implemented in Go and Java, respectively. We also want to measure the impact of the number of proxy sidecars on the latency. We measured the latency as the duration of time that has elapsed since the initial receipt of the request in the first proxy, the requests pass through all the subsequent proxy sidecars, ultimately arriving at the user container, and returning to the original proxy that initially received the request. This experiment was also performed with two sidecars implemented in Go and Java using the provided libraries.

5.2.1 | Test Bed

In the experimental setup we used three servers interconnected via a 10 Gb/s network. Each server hosts virtual machines (VMs) configured with SR-IOV enabled NICs, assigning a Virtual Function (VF) to each VM; the Kubernetes cluster consists of a master VM with 12 vCPUs and 32 GB RAM, and eight worker VMs each equipped with 10 vCPUs and 24 GB RAM, running Kubernetes version 1.29 and Knative version 1.14.

5.2.2 | Impact of the Number of Small Sidecars in the Cold Start Time

In this experiment, we will evaluate the impact of the number of sidecars in the cold start time, measured as the time that elapses from when the request is sent until it reaches the application container and can be attended. As we have 8 worker nodes, and a constrained platform in terms of resources, we performed three tests:

- 8 simultaneous calls ensuring that each replica lands in a different worker node (low demand)
- 24 simultaneous calls so that each worker node allocates three replicas (medium demand)
- 72 simultaneous calls with 9 replicas per worker node (high demand)

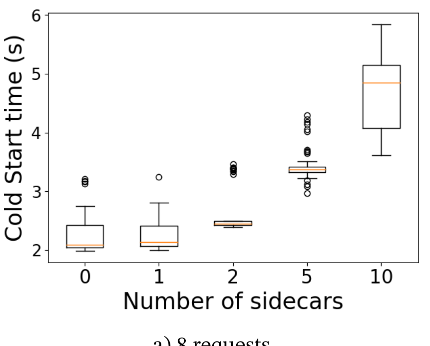
Besides, this experiment was repeated across different scenarios, specifically with deployments containing 0, 1, 2, 5, and 10 sidecar containers.

In the most demanding scenario, each worker node will be requested to start 108 containers (9 pods with 12 containers: the queue-proxy injected by Knative, the 10 proxy sidecars, and the user container). In particular, we used a sidecar implemented in Go that simply forwards the request to the next proxy sidecar. The container image of this sidecar is 17.2 MB. The queue-proxy image has 164 MB, and the user image (in this experiment an application with an echo) has 125 MB.

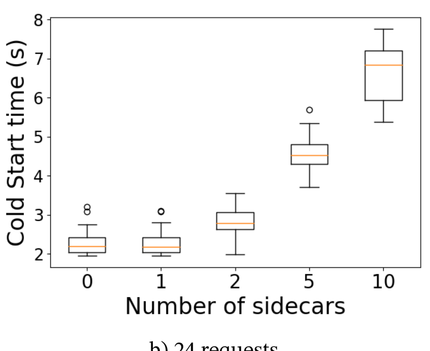
The experiment has been repeated 100 times to account for variability and transient environmental conditions that could affect the results. During this experiment no other workload was executed in the platform. Figure 7 shows the boxplot of the distributions of response times across different numbers of sidecar containers. From the data we can conclude, as expected, that as more sidecars are included, the cold start time increases. Therefore, for deployments where cold start performance is critical the number of sidecars should be carefully considered. While sidecars add valuable capabilities, their impact on startup time can be detrimental to the user experience in latency-sensitive applications. Results show that the cold start mean overhead increases a 66% with 5 proxy sidecars compared to the base case when each worker node has to start one replica of the pod and rises to 103% when each worker node has to start 9 replicas of the pod. We consider that use cases with more than five proxy sidecars are unlikely.

Another consequence of this data is that the number of simultaneous requests performed has not a big impact. This can be due to the fact that the start time is dominated

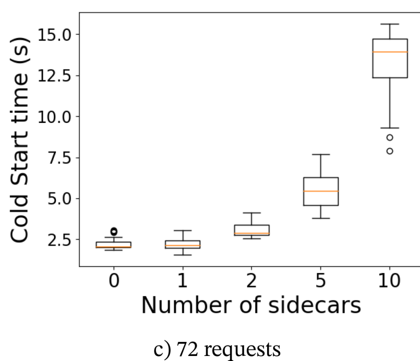
1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitaet De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



a) 8 requests



b) 24 requests



c) 72 requests

FIGURE 8 | Cold-start comparison depending on the number of proxy sidecars, implemented using Java, per pod. In the more demanding scenario (72 requests) each worker node has to start 108 containers.

by the two bigger containers (the queue-proxy and the user container).

5.2.3 | Impact of the Number of Big Sidecars on the Cold Start Time

In this case, we have used the Java library to implement a sidecar that, as in the previous section, simply forwards the request to the next element in the chain. The image of this proxy sidecar is 62.4 MB. Using this proxy sidecar, we repeated the same tests described in the previous section.

Figure 8 shows the boxplot of the distributions of response times across different numbers of sidecar containers.

As in the results with the Go implementation, we conclude that as more sidecars are included, the cold start time increases. This overhead is more evident when the demand to start more pods simultaneously increases. With 24 simultaneous requests, the mean overhead to start 10 sidecars is 4.3282 s (from 2.2558 s with no proxy sidecars to 6.5959 s with ten proxy sidecars). With 72 simultaneous requests, the mean overhead of using 10 sidecars increases the time by 10.9131 s (from 2.2027 to 13.1158 s).

5.2.4 | Impact of the Number of Sidecars and Implementation Language on Latency

The objective of this experiment was to quantify the impact of sidecar containers on the latency of requests processed by microservices. Specifically, the objective was to quantify the time taken from when a request is sent to a server until it is received by the core application, across varying numbers of sidecars when the sidecars merely pass the request to the subsequent element in the chain. In this case, the focus was on the impact of the network stack on latency, given that the packets must traverse this network stack at each sidecar proxy.

The latency experiment involved sending 600 requests to a server configured with different numbers of sidecars (0, 1, 2, 5, and 10).

Latency was measured in milliseconds, capturing the duration from the transmission of the request to its receipt by the core application. This approach allowed us to assess the additional latency attributable to the presence of sidecar containers. Figure 9 illustrates the distribution of request latency for each configuration across different numbers of sidecar containers and implementations.

The impact of sidecars on latency within microservices architectures demonstrates that deployments with a single sidecar result in minimal latency.

As the number of sidecars increased, a gradual rise in latency was observed. As illustrated in the optimal scenario depicted in Figure 9a, the mean latency increased from 7.80 ms with one sidecar to 9.30 ms with two sidecars and to 12.40 ms with five sidecars. The deployment with ten sidecars exhibited a notable increase in latency, with a mean latency of 16.11 ms. This indicates a total latency addition of 7.51 ms from the baseline to deployment with ten sidecars. Despite the progressive increase in latency with each additional sidecar, even with ten sidecars, the resulting increase remains modest. This increase is negligible when considered in the context of typical usage patterns in most applications, which would not typically involve the deployment of ten sidecars.

5.2.5 | Performance Comparison With Envoy

Envoy is a well-established single-proxy solution widely used in production due to its comprehensive set of built-in filters and relatively low overhead when managing standard functionalities (e.g., load balancing, telemetry, security). However, our chain-of-sidecars approach targets a different set of objectives. We allow developers to implement sidecars in any programming language, integrate any required libraries, and enable specialized use cases (e.g., data meshes that download data to a shared filesystem before handing it off to the main application). These customization needs are harder to achieve with Envoy filters, which require C++ or WebAssembly development and recompilation or complex plugin integration.

Thus, while we include Envoy in our performance comparisons to illustrate trade-offs, our approach is not intended as a direct competitor for Envoy or as a drop-in replacement for its extensive out-of-the-box features. Instead, it aims to provide modular sidecar containers that can be composed or updated independently. If minimum latency or cold-start time is paramount and existing Envoy filters suffice, Envoy remains a strong candidate.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitaet De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

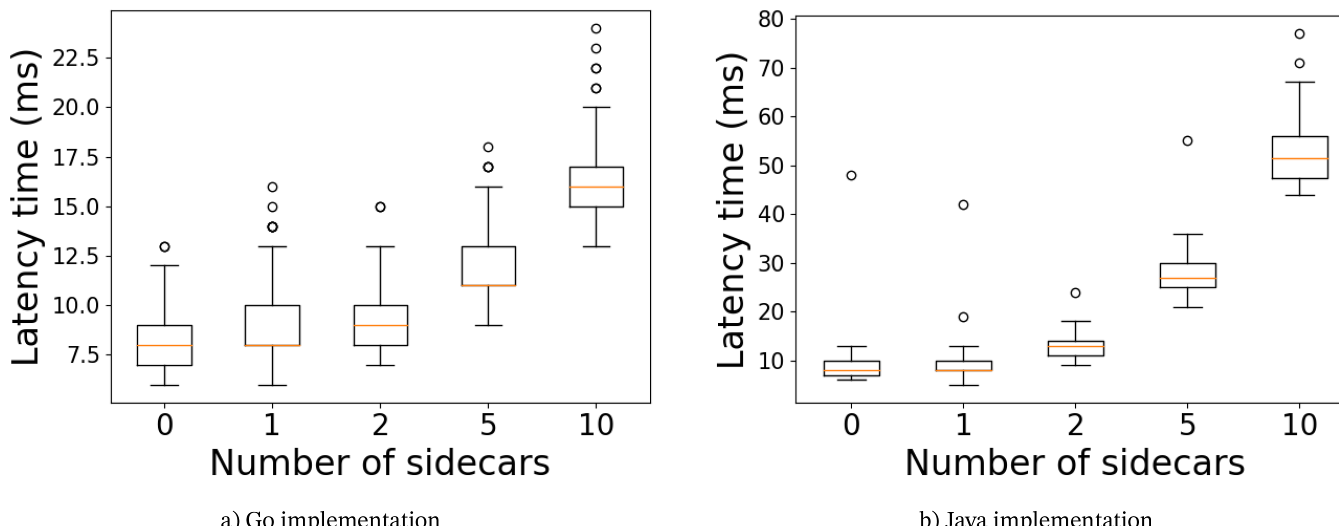


FIGURE 9 | Latency depending on the number of proxy sidecars, implemented using Go (left) and Java (right).

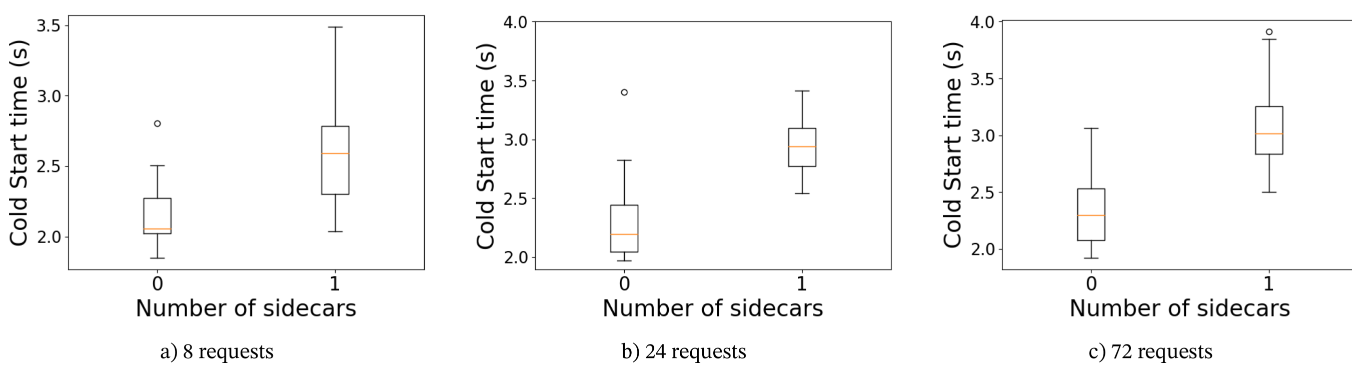


FIGURE 10 | Cold-start depending on the number of concurrent requests using Envoy.

Conversely, in scenarios that demand highly specialized logic or nonstandard dependencies, our sidecar approach may prove more flexible even if it entails higher overhead and startup costs.

The experiments in this section have been performed in the same test bed described in Section 5.2.1. In the first experiment we will measure the impact on cold start time when Envoy is used. The Envoy image used has a size of 135 MB. We will measure the cold start time when performing 8, 24, and 72 simultaneous requests. In the more demanding scenario, each worker node must start 18 containers (9 requests per worker node and two containers, Envoy and the application container, per request). Results are shown in Figure 10.

The mean cold start time increases by 20.7% with 8 requests (from 2.144 to 2.588 s), by 28.58% with 24 requests (from 2.281 to 2.933 s), and by 33.99% with 72 requests (from 2.327 to 3.118 s). These times are between the times obtained for 2 and 5 sidecars in Figures 7 and 8. This outcome is reasonable since the image sizes of the proxy sidecars developed in Go were smaller compared to Envoy.

Finally, we have performed an experiment to evaluate the latency introduced by a chain of filters in Envoy. Each filter in the chain just sends the request to the next filter. The experiment was performed with 1, 2, 5, and 10 filters in the chain. Figure 11 shows the results obtained.

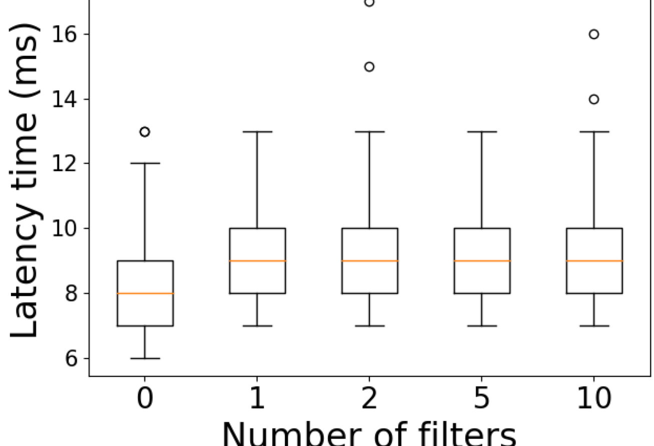


FIGURE 11 | Latency depending on the number of filters in the chain of Envoy.

Table 1 presents a side-by-side comparison of the average cold-start times and steady-state latency metrics for the Go library

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitaet De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 | Comparison of average cold-start times (in seconds) and latency (in milliseconds) for different approaches and numbers of sidecars/filters.

	Cold start (s)				Latency (ms)			
Approach	**1**	**2**	**5**	**10**	**1**	**2**	**5**	**10**
Go library	2.32	2.27	2.74	3.57	9.35	9.91	12.25	16.89
Java library	2.20	3.08	5.47	13.11	8.97	15.55	36.89	72.54
Envoy	3.18	3.18	3.18	3.18	9.65	9.66	9.73	10.09

implementation, the Java library implementation, and Envoy, each tested with varying numbers of sidecars/filters (1, 2, 5, and 10). These measurements showcase the trade-off between flexibility and the overhead introduced by additional containers. Cold-start times correspond to the scenario with 72 concurrent requests.

To cold start time, with up to two sidecars, K8Sidecar with Go and Java libraries actually outperforms Envoy. With five sidecars, the Go implementation outperforms Envoy; however, the Java implementation is worse. With ten sidecars, the Go implementation has an overhead of 12% with respect to Envoy, while the Java implementation shows a big degradation in the cold start time, this degradation is due to the bigger size of the Java image, 60 MB, compared to the Go image, 10 MB.

In terms of latency, with one sidecar K8Sidecar shows a similar behavior as Envoy using the Java and Go libraries. With 5 sidecars, the Go implementation shows an overhead of 25.9% compared to Envoy (just 2.52 ms), while the Java implementation imposes an overhead of 279%. With 10 sidecars, the Go implementation shows an overhead of 67.4% compared to Envoy (just 6.8ms), and the Java implementation induces an overhead of 62.45 ms.

From these results, it can be observed that for up to 1 and 2 filters and nearly 5, the chain-of-sidecars approach (especially using the Go library) remains reasonably close to Envoy in terms of cold start and latency. Overall, these measurements indicate that our approach is not designed for minimal latency or very large sidecar chains, nor does it aim to compete directly with Envoy's suite of built-in features. Instead, the chain-of-sidecars design deliberately prioritizes developer flexibility, allowing sidecars in any programming language with arbitrary libraries, at the cost of increased overhead and cold-start time, especially when the chain grows beyond a few sidecars. If a scenario demands only standard traffic management or load-balancing features already provided by Envoy's built-in filters, then Envoy is likely the superior choice in terms of raw performance. However, for specialized logic beyond the scope of Envoy filters-such as advanced data mesh workflows, our approach enables easy sidecar creation and chaining, even incorporating Envoy itself as one more sidecar if desired.

5.3 | Related Work

The evolution of microservices architectures has required the development of tools and patterns that facilitate service communication, observability, and security. One of the significant patterns that have emerged is the utilization of sidecar proxies, which act as intermediaries for managing requests in microservices.

Kubernetes, from version 1.29, has enabled the use of sidecar containers on a pod by declaring containers in the `initContainers` section with an `OnFailure` restart policy. However, they are not intended to act as proxies and there is no clear mechanism about how to compose and chain them.

Envoy Proxy [6] is a high-performance distributed proxy designed to mediate all inbound and outbound traffic for services in a microservices architecture. It offers dynamic service discovery, load balancing, web protocol support, and observability. Envoy is designed to be transparent to the application, requiring no changes to the code of the services themselves. It can be deployed as a proxy sidecar, handling communication between services, or as an edge proxy, managing inbound and outbound traffic.

Although Envoy Proxy offers a range of functionalities for traffic management and service mesh architectures, our software solution prioritizes a more adaptable and dynamic approach to sidecar management at deployment time specifically tailored for Kubernetes environments. The key distinctions between the two approaches are outlined in Table 2.

In Reference [11], the sidecar pattern is one of the patterns used to propose an architecture for cloud-native applications deployed on Kubernetes. In their proposal the proxy sidecar performs token-based authentication, therefore it is a reusable proxy across different microservices that can be tailored by using `ConfigMaps`.

The work [12] presents a Kubernetes operator (CRDs and their controllers) to launch and monitor workloads in external HPC clusters managed by Slurm, LSF or Ray. When a Custom Resource (CR) is created, the controller starts a pod (a bridge) in Kubernetes that runs the job in the external cluster. Besides the status of the CR is updated when the status of the job in the external cluster changes. The paper provides an example of a bridge for Slurm and an example of a deployment managed by Kubeflow Pipelines, which offers a Python SDK for defining and controlling computational directed acyclic graphs, where each component execution corresponds to a single container execution.

The performance impact of service mesh sidecars has been studied in Reference [13], where the main sources of overhead (IPC, read, write, notification, sidecar protocol parsing, sidecar filter, and processing) were identified. The total performance overheads

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitaet De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 2 | Comparison of features between Envoy and k8sidecar.

Nr.	Envoy feature	K8Sidecar feature
F1	Employs a singular proxy instance that aggregates all filters, necessitating all traffic to flow through this centralized point.	Opts for a decentralized approach, where each filter operates as an independent proxy. Filters can be managed and scaled independently.
F2	Requires recompilation for most changes, integrating updates directly into the proxy binary. This process ensures consistency but can be less flexible.	Enables specific proxy updates, reducing downtime and accelerating feature deployment.
F3	Extension development is limited to C++ and WebAssembly, restricting the choice of programming languages.	Provides a more flexible environment for proxy development, supporting any language that can interact with HTTP/CloudEvents [10].
F4	Uses own library for extensions, with specific specifications and functions. This requires developers to adapt to Envoy's framework.	Leverages standard HTTP/CloudEvents protocols using conventional library methods.
F5	Has a wide range of use cases supported by an extensive but limited library of extensions. The scope for community contributions is bound by the complexity of Envoy's architecture.	Envisions a "Proxy Store" for future development, aiming to host a repository of open-source proxies created by the community.
F6	Configurations are defined at compilation time, embedding settings directly into the proxy.	Allows for deployment time configuration, enabling on-the-fly adjustments without recompilation.
F7	Integrates all filters within the same container, streamlining the filter chain.	Distributes each filter across different containers.
F8	Employs a strict priority system defined in the configuration, dictating the order of filter processing.	Implements a numerical priority system, offering granular control over filter execution order.
F9	Extensive support for various web protocols, targeting a wide array of networking needs and applications.	Focuses primarily on HTTP and CloudEvents, but other protocols such as AMQP can be implemented.

of a service mesh will be a function of these basic operations. The paper shows that HTTP and gRPC protocol parsing is the main source of overhead. The adequacy of service mesh to mobile edge cloud has been discussed in Reference [14] where DevOps is identified as the main driver of service mesh implementations. In DevOps, the requirements in terms of performance, observability, communication patterns and protocols, and traffic control among others are substantially different from those required in mobile edge cloud. The paper argues that the use of the proxy sidecar incurs both latency and resource overheads. Based on these two works, we can conclude that the chain of sidecars proposed in this work might not be suitable for mobile edge cloud but can be suitable for other IT scenarios allowing rapid experimentation of new features.

6 | Conclusions and Future Work

This paper has introduced K8Sidecar, a Kubernetes software solution that leverages the sidecar proxies pattern to dynamically integrate sidecar containers into microservices and serverless architectures. By intercepting resource creation requests, K8Sidecar not only facilitates modularity at deployment time but also bolsters observability while maintaining the integrity of service logic within the Kubernetes ecosystem. The utilization of custom resource definitions allows the dynamic specification of sidecar configurations, enhancing flexibility and adaptability.

Our study provides a thorough explanation of the proposed architecture, libraries offered to develop new sidecars, and sample use cases such as monitoring, data management, or authentication. Furthermore, measuring the impact on cold start and latency, we have determined that while the number of sidecars should be carefully considered in deployments where cold start performance is critical, their impact on latency remains modest (less than 10 ms, even with 10 sidecars included in a Kubernetes architecture).

Moving forward, the development of K8Sidecar could greatly benefit from exploring the integration of additional communication protocols beyond HTTP and CloudEvents, such as AMQP or MQTT, to broaden its applicability in diverse IoT and real-time data processing scenarios. Further, implementing AI-driven automation within the sidecar management process could enhance the system's ability to predict and adapt to dynamic scaling needs and optimize resource allocation.

Additionally, future studies could investigate the trade-offs between functionality and performance overhead in more depth, particularly in edge computing environments where resources are limited and latency is critical. These advancements could help in refining the K8Sidecar's capabilities, making it even more versatile and efficient for modern cloud-native applications.

Author Contributions

Andoni Salcedo-Navarro: conceptualization, Investigation, Software, Experiments, Writing – original draft, Writing – review and editing. **Miguel Garcia-Pineda:** supervision, Writing – review and editing,

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitat De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Funding acquisition. **Juan Gutiérrez-Aguado:** conceptualization, Software, Supervision, Writing – review and editing, Funding acquisition.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The authors have nothing to report.

Endnotes

[1] The code repository with the operator is available at https://github.com/cloudmedialab-uv/k8sidecar.

[2] https://github.com/cloudmedialab-uv/k8sidecar-go-lib.

[3] https://github.com/cloudmedialab-uv/k8sidecar-java-lib.

[4] An example of a rate-limiting sidecar, implemented using the Go library, can be found at https://github.com/cloudmedialab-uv/k8sidecar-go-lib/tree/main/examples/ratelimiter.

[5] An example of a logging sidecar using the Java library can be found at https://github.com/cloudmedialab-uv/k8sidecar-java-lib/tree/main/examples/logging.

[6] The source code for this sidecar using the Java library can be found at https://github.com/cloudmedialab-uv/k8sidecar-java-lib/tree/main/examples/authentication.

References

1. N. Dragoni, S. Giallorenzo, A. L. Lafuente, et al., *Microservices: Yesterday, Today, and Tomorrow:195-216* (Springer International Publishing, 2017), https://doi.org/10.1007/978-3-319-67425-4_12.

2. S. R. Goniwada, *Cloud Native Architecture and Design Patterns:127-187; Berkeley* (Apress, 2022).

3. B. Burns and D. Oppenheimer, "Design Patterns for Container-Based Distributed Systems," in *USENIX Association* (Denver, CO, 2016), 108–113.

4. J. T. Duarte Maia and F. Figueiredo Correia, "Service Mesh Patterns," in *EuroPLop '22* (Association for Computing Machinery, 2023), https://doi.org/10.1145/3551902.3551962.

5. L. Giamattei, A. Guerriero, R. Pietrantuono, et al., "Monitoring Tools for Devops and Microservices: A Systematic Grey Literature Review," *Journal of Systems and Software* 208 (2024): 111906, https://doi.org/10.1016/j.jss.2023.111906.

6. Lift, "Envoy Proxy," 2024.

7. K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running Dive Into the Future of Infrastructure*, 1st ed. (O'Reilly Media, Inc., 2017).

8. V. E. Eyk, L. Toader, S. Talluri, L. Versluis, A. Uţă, and A. Iosup, "Serverless Is More: From PaaS to Present Cloud Computing," *IEEE Internet Computing* 22, no. 5 (2018): 8–17, https://doi.org/10.1109/MIC.2018.053681358.

9. B. Ibyram and R. Huss, *Kubernetes Patterns*, 2nd edi ed. (O'Reilly Media, 2023).

10. Cloud Native Computing Foundation, *Cloudevents Project* (2024), https://cloudevents.io/.

11. Q. Jiao, B. Xu, and Y. Fan, "Design of Cloud Native Application Architecture Based on Kubernetes," in *IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)* (IEEE, 2021), 494–499, https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech52372.2021.00088.

12. B. Lublinsky, E. Jennings, and V. Spišaková, "A Kubernetes Bridge Operator Between Cloud and External Resources," in *IEEE 8th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)* (IEEE, 2024), 263–269, https://doi.org/10.1109/icccbda56900.2023.10154770.

13. X. Zhu, G. She, B. Xue, et al., "Dissecting Overheads of Service Mesh Sidecars," in *15th ACM Symposium on Cloud Computing (SoCC)* (Association for Computing Machinery, 2023), 142–157, https://doi.org/10.1145/3620678.362465.

14. A. Duque, C. Klein, J. Feng, X. Cai, B. Skubic, and E. Elmroth, "A Qualitative Evaluation of Service Mesh-Based Traffic Management for Mobile Edge Cloud," in 2022 22nd *IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* (IEEE, 2022), 210–219, https://doi.org/10.1109/CCGrid54584.2022.00030.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3423 by Universitaet De Valencia, Wiley Online Library on [10/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Capítulo 7

UHD Video Encoding in CPU Versus GPU: Quality and Performance Trade-Offs

Received 10 February 2025, accepted 12 March 2025, date of publication 21 March 2025, date of current version 3 April 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3553634

UHD Video Encoding in CPU Versus GPU: Quality and Performance Trade-Offs

ANDONI SALCEDO-NAVARRO, JUAN GUTIÉRREZ-AGUADO, AND MIGUEL GARCIA-PINEDA

Department of Computer Science, ETSE-UV, Universitat de València, 46010 Valencia, Spain

Corresponding author: Miguel Garcia-Pineda (miguel.garcia-pineda@uv.es)

This work was supported in part by the Ministerio de Ciencia, Innovación y Universidades /Agencia Estatal de Investigación (AEI)/10.13039/501100011033 under Grant PID2021-126209OB-I00; and in part by "European Regional Development Fund (ERDF) A Way of Making Europe," by "European Union."

ABSTRACT In the contemporary digital media environment, video encoding is of paramount importance for applications such as streaming and video conferencing. With the increasing demand for higher resolutions and immersive video, optimizing videos for Internet consumption necessitates the use of the latest codecs. Additionally, live event broadcasts require fast and efficient codecs, making GPU codecs essential. This paper analyses the coding efficiency of video encoders on CPUs and GPUs with 4K content. We have selected three families of encoders (H264/AVC, HEVC, and AV1) and have computed the Bjøntegaard Delta (BD) using a set of 15 UHD videos to compare the results of the CPU implementation (libx264, libx265, and librav1e) with their GPU version (h264_nvenc, h265_nvenc, and av1_nvenc) of each family. Besides, we have measured the mean encoding time per frame (as the test videos have different duration). The BD rate analysis shows that the CPU implementations of the codecs for H264 and AV1 (libx264 and librav1e) require in mean more than 20% bitrate (for both metrics PSNR and VMAF) compared to the GPU implementations (h264_nvenc and av1_nvenc). However, the CPU implementation of the H265 (libx265) codec required around 7.5% less bitrate (for both metrics) than the GPU implementation (hevc_nvenc). The encoding times reveal that the mean encoding time per frame for the GPU encoders is very similar (109 ms, 108.39 ms, and 117.77 ms) and they achieve a great time reduction compared to the CPU version (1.3 times faster for H264, 2.35 times faster for HEVC, and 68.14 times faster for AV1). Therefore, the GPU implementations of these encoders reduce the encoding time drastically, specially in the case of AV1, and lead to a noticeable reduction of the bitrate (except for HEVC).

INDEX TERMS AV1, H264/AVC, HEVC, NVENC, processing time, PSNR, video encoding, VMAF.

I. INTRODUCTION

In the contemporary landscape of digital media, video encoding emerges as a pivotal technology, facilitating a wide array of applications ranging from streaming services and video conferencing to the storage of multimedia content [1]. As the volume of video data continues to increase, driven by a demand for higher quality and more immersive experiences, the importance of efficient video compression cannot be overstated. It plays a crucial role not only in reducing the necessary bandwidth for video transmission but also in minimizing storage costs, all while maintaining or enhancing the viewer's experience. This interplay between compression efficiency and quality preservation underscores the technological and economic significance of advancements in video encoding algorithms and their implementations [2].

The rapid growth in demand for online video content has driven the development of advanced codecs capable of delivering high-quality video at lower bit rates. The need to encode and stream videos at higher resolutions, such as 4K and 8K, while maintaining acceptable quality and minimizing data consumption has posed a challenge [3]. As a result, codecs like H264/AVC [4], H265/HEVC [5], and AV1 [6] have emerged to meet the demands of modern

The associate editor coordinating the review of this manuscript and approving it for publication was Miaohui Wang.

© 2025 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

video streaming. These advanced codecs employ increasingly sophisticated algorithms to compress video data while preserving high-quality image fidelity. For instance, H264/AVC was widely adopted due to its ability to deliver high-definition (HD) content at relatively low bit rates. The subsequent development of H265/HEVC further improved compression efficiency by up to 50% compared to H264. Most recently, AV1 has emerged as a promising next-generation codec that leverages advanced techniques, such as intra-frame prediction and adaptive quantization to achieve even better compression ratios. However, the enhanced complexity of these codecs inherently leads to longer encoding times, presenting a critical bottleneck for both offline processing and real-time applications.

To mitigate the increased computational demands, Graphical Processing Units (GPUs) have been increasingly employed in the video encoding process to improve performance and efficiency [7]. GPUs excel in handling the parallelizable and computationally intensive tasks characteristic of modern video encoding, thereby enabling faster processing essential for live streaming and broadcasting [8], [9]. It is important to emphasize, however, that while GPU implementations yield substantial speedups, the actual performance gains are highly contingent on the specific hardware configurations employed. This dependency underscores the critical need for a comprehensive evaluation of both algorithmic optimizations and the underlying CPU/GPU architectures when assessing encoding performance.

The assessment of encoded video quality is paramount for ensuring an optimal viewer experience. To this end, a suite of objective quality metrics such as the Peak Signal-to-Noise Ratio (PSNR), the Structural Similarity Index Measure (SSIM), and the Video Multimethod Assessment Fusion (VMAF) have been proposed [10]. These metrics serve as critical tools in quantitatively evaluating the visual integrity of encoded videos, offering insights into the perceptual impact of various compression techniques. The challenge lies not only in accurately measuring the effects of compression on visual quality but also in addressing the inherently subjective nature of human visual perception. Objective metrics, therefore, help bridge this gap by providing a standardized means of comparison that can guide both the development of new encoding technologies and the optimization of existing ones.

In comparing the performance of different encoding approaches, the Bjøntegaard Delta (BD) metrics [11] including BD-Rate and BD-Quality (assessed using measures such as PSNR or VMAF) provide a quantitative measure of the differences in compression efficiency and quality degradation or enhancement between codec implementations. These metrics offer a comprehensive way to assess the impact of various encoding strategies on both bitrate savings and potential quality shifts. To compute these metrics, the sampled points of the bitrate-distortion curve must be interpolated. The Akima interpolation method, known for its ability to produce smooth curves without the oscillations often associated with polynomial interpolations, is particularly suited for this purpose [12].This approach facilitates a deeper understanding of the trade-offs between bitrate and perceived video quality, thereby informing the ongoing development and evaluation of both CPU- and GPU-based encoding solutions.

The main goal of this study is to evaluate the impact of GPU acceleration on video quality across various codecs, using the Bjøntegaard metrics in conjunction with Akima interpolation. While the reduction in overall encoding time achieved through GPU offloading is well understood, detailed comparisons between CPU and GPU implementations at UHD (4K) resolution remain limited. To address this gap, our contributions are as follows:

- **Comprehensive UHD Analysis:** We present the first in-depth comparison of CPU- and GPU-based encoder implementations (H.264/AVC, HEVC, and AV1) specifically at 4K (UHD) resolution.
- **Rate-Distortion Evaluation:** Utilizing the Bjøntegaard metrics with Akima interpolation, we assess rate-distortion efficiency across codecs, highlighting the impact of GPU acceleration on bitrate savings and video quality.
- **Encoding Speed vs. Quality Trade-offs:** We analyze the trade-offs between encoding speed and visual fidelity, comparing CPU- and GPU-accelerated approaches to inform optimal workflow decisions.

The paper is organized as follows: section II presents related work. The testbed describing the videos and the platform used to perform the experiments is presented in section III. Section IV describes the experiments and presents the results obtained. Finally, section V presents the conclusion and future work.

II. RELATED WORK

The landscape of video compression standards has evolved significantly, the introduction of High Efficiency Video Coding (HEVC or H.265) marked a major advancement in video compression, significantly improving efficiency over the previous H.264/AVC standard. This enhancement is essential for meeting the growing demand for high-definition video across various applications [5]. While H.264/AVC set a precedent in video compression, offering a balance of complexity and performance that has been widely adopted across numerous platforms [13], it is the newer standards like AV1 that are pushing the boundaries even further. AV1, in particular, has been noted for outperforming H264 and VP9 in practical applications, demonstrating its potential for broader adoption in the industry, especially given its open-source nature and royalty-free licensing [14]. The Alliance for Open Media's detailed overview of AV1 reveals the codec's innovative features designed to provide high compression efficiency without compromising on video quality, marking it as a significant advancement in the field of video compression [15]. These improvements in video

codec development underscore a clear trajectory towards more efficient video streaming.

In the realm of video compression, the quest for optimal quality metrics remains paramount as new codecs emerge and compete for market dominance. Uhrina et al. [16] examine the performance of state-of-the-art video codecs, including VVC, AV1, HEVC, and AVC at high resolutions, utilizing PSNR and VMAF to objectively evaluate video quality. Their analysis provides insights into the calculation and effectiveness of these metrics, reinforcing their critical role in assessing video fidelity. The intricacies of these quality metrics, especially the challenges of applying them consistently across different codecs and resolutions, are further analyzed by Barman et al. [11], who offer a tutorial overview of the Bjøntegaard Delta metric while also discussing the variable behavior of SSIM and VMAF. Nawaz et al. [17] extend this examination to mobile devices, considering additional factors that influence the Quality of Experience (QoE), highlighting VMAF's capacity to reflect video quality degradation more accurately in comparison to other metrics like MS-SSIM, which was considered as less reliable. Furthermore, the work by Amestoy et al. [18]. evaluates the Versatile Video Coding (VVC) standard, not only in terms of video quality — using PSNR, SSIM, and VMAF — but also considering the coding complexity, thereby providing a comprehensive perspective on the efficiency and practicality of codec implementations. Together, these studies underscore the interplay between quality assessment metrics and encoding standards, guiding the industry in balancing the trade-offs between computational demand and the viewer's visual experience.

In the comprehensive analysis of codec performance, particularly focusing on the interplay between computational efficiency and output quality, the Bjøntegaard Delta stands out as crucial tool. BD rate, as an established metric, facilitates a comparative analysis of the efficiency of different video codecs, as it can quantify the trade-off between bitrate and quality. As detailed in [19], it is widely used due to its computational simplicity and effectiveness, despite its results not aligning perfectly with subjective quality assessments. Different interpolations have been proposed to approximate the bitrate-distortion curve, being the Akima interpolation the one that provides better results [12]. Our study leverages these tools to elucidate the differences not just among codecs, but specifically between GPU-accelerated and non-accelerated implementations, shedding light on how GPU acceleration can affect quality of the encoded videos.

While previous research has made significant strides in evaluating video codec performance using traditional and emerging quality metrics, recent advancements in video encoding optimization underscore the role of GPU acceleration, offering a notable contrast to traditional CPU-based methods in both efficiency and performance. The study by Katsigiannis et al. [7] provides a comprehensive evaluation of GPU versus CPU in the context of video compression, revealing that GPUs not only enhance encoding speed but also improve energy efficiency. Xiao et al. [20] extend this narrative by exploring HEVC encoding optimizations using multicore CPUs and GPUs, focusing on the theoretical underpinnings of encoding processes rather than direct quality comparisons. Their findings highlight the potential of GPU acceleration to significantly reduce computational demands. Similarly, Su et al. [21] examine the benefits of GPU-based optimizations for high-definition video codecs on mobile devices, emphasizing performance improvements in terms of encoding time and energy consumption. Additional research has also utilized GPUs to develop multiresolution video encoding systems, accelerating the encoding process significantly. For instance, a comparative study revealed that using a GPU over a CPU can achieve a reduction factor of 3.75 in encoding time [22].

Although several studies highlight the need for further exploration in this domain, the unique intersection of hardware acceleration with codec efficiency and quality has not been thoroughly addressed. Our study aims to fill this void by juxtaposing GPU-accelerated and non-accelerated codec implementations, applying the latest advancements in video quality metrics. This approach not only augments the existing corpus of knowledge but also provides a novel perspective by examining how the utilization of GPU resources can influence the delicate balance between encoding speed and visual fidelity. By establishing a framework that isolates hardware acceleration as a variable, we enable a deeper understanding that contributes to both academic research and practical applications in video encoding. This framework lays the foundation for the following sections, which discuss the experimental setup, results, and the implications of the findings.

III. TESTBED

A. EXPERIMENTAL SETUP

The experimental setup comprises two physical machines linked by a 10 Gb/s switch, on which a Kubernetes cluster has been deployed. The first machine hosts the Kubernetes control plane node within a virtual machine that is equipped with 16 GB of RAM and 10 vCPUs. The second machine is configured with 128 GB of RAM and 48 cores Intel Xeon Gold 5320, and it includes an Nvidia 4090 RTX GPU with 8 GB of RAM and 2 NVENC sessions. This GPU was chosen for its compatibility with all NVIDIA encoders utilized in this research. This machine supports four Kubernetes worker VMs, each allocated 10 vCPUs and 28 GB of RAM; only one of these VMs is equipped with the Nvidia 4090 RTX GPU.

The virtual infrastructure has been carefully designed to maximize performance. SR-IOV NICs provide virtual functions to the VMs, while the GPUs are assigned using PCI passthrough. Each VM benefits from its own dedicated SSD, which ensures swift data access. In this environment, Docker is used as the Container Runtime, with Flannel implementing the Container Network. Furthermore, the

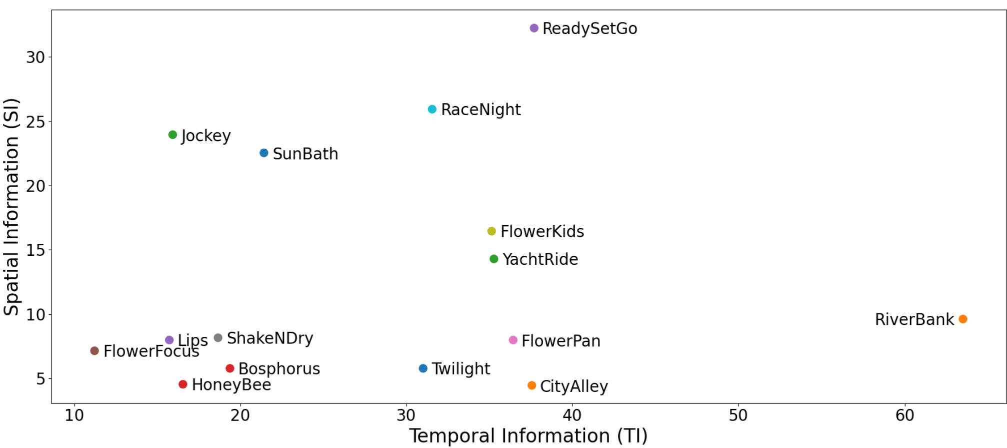


FIGURE 1. Spatial Information and Temporal Information (SI-TI) of the UVG 4K Video Sequences.

configuration includes the Nvidia Operator (version v23.3.1), which enables a dynamic allocation of GPUs to VMs.

B. VIDEOS AND CODECS

To evaluate the different codec implementations and their performance under GPU acceleration, a comprehensive testbed was set up using a set of fifteen 4K raw video sequences.

These sequences, extracted from the Ultra Video Group (UVG) dataset [23], provide a diverse range of content with varying spatial and temporal complexity (see Figure 1). The UVG dataset is a publicly available collection of fifteen versatile 4K (3840×2160) test video sequences recorded at high frame rates of 50 or 120 frames per second (fps). They are stored in 10-bit 4:4:4 YUV formats. For the purposes of our study, we selected the fifteen sequences to ensure a wide range of visual content, from fast-moving action to slower more detailed scenes.

Table 1 summarize the properties of each video sequence, including the video name, file size in gigabytes (GB), duration in seconds, and frame rate in fps.

TABLE 1. Properties of 4K video sequences from the UVG dataset.

Name	Acronym	Size (GB)	Duration (s)	Framerate (fps)
Bosphorus	BOS	2.3	5	120
HoneyBee	HON	2.4	5	120
Jockey	JOC	2.7	5	120
ReadySetGo	REA	2.1	5	120
ShakeNDry	SHA	3.0	2,5	120
YachtRide	YAC	3.0	5	120
CityAlley	CIT	2.8	12	50
FlowerFocus	FLOF	3.5	12	50
FlowerKids	FLOK	3.2	12	50
FlowerPan	FLOP	2.4	12	50
Lips	LIP	2.6	5	120
RaceNight	RAC	1.5	12	50
RiverBank	RIV	0.76	12	50
SunBath	SUN	2.6	6	50
Twilight	TWI	2.3	12	50

Table 2 shows the codecs analyzed in this work. For each codec we selected an implementation that uses only CPU and another implementation that uses also the GPU. For H264/AVC we have used libx264 and h264_nvenc. For H265/HEVC the two implementations are lib265 and hevc_nvenc. Finally, for AV1 we selected librav1e and av1_nvenc.

TABLE 2. Codecs and implementations considered in this work. For each codec, two implementations have been considered: one that uses only the CPU and other that uses also the GPU.

Codec	CPU implementation	GPU implementation
H.264/AVC	libx264	h264_nvenc
H.265/HEVC	libx265	hevc_nvenc
AV1	rav1e	av1_nvenc

IV. EXPERIMENTS AND RESULTS

We encoded the videos from the dataset, described in the previous section, at a comprehensive range of bitrates: 2.5, 5.0, 7.5, 10, 12.5, 15, 17.5, 20.0, 22.5, and 25.0 Mbps. The study encompassed three different codecs (H.264/AVC, H.265/HEVC, and AV1) using both traditional CPU-based and GPU-accelerated versions of each codec. The bitrate range was selected to thoroughly evaluate the performance of these codecs in environments ranging from limited to extensive bandwidth availability.

```
ffmpeg -i video.mp4 -c:v <codec> -b:v <bitrate>
  -maxrate <bitrate> -minrate <bitrate> enc-video.mp4
```

LISTING 1. Sample ffmpeg command used to encode the videos.

Listing 1 presents an illustrative example of the `ffmpeg` command executed to encode a video. The parameters used in the command are explained in detail below.

- `-c:v <codec>`: Sets the video codec to be used for encoding (`libx264`, `libx265`, `rav1e`, `h264_nvenc`, `hevc_nvenc`, or `av1_nvenc`).

- `-b:v <bitrate>`: Sets the target average bitrate for the encoder.
- `-maxrate <bitrate>`: Sets the maximum bitrate allowed during encoding, matching the target bitrate for constant bitrate encoding.
- `-minrate <bitrate>`: Sets the minimum bitrate allowed during encoding, also matching the target bitrate.

The remaining parameters are set to their default values. For example, all encoders utilize the available CPU cores which in our setup amounts to 10 cores performing the encoding. By default, medium preset is used for all encoders, and all videos are encoded using the Random Access configuration.

Listing 2 shows an example of the `ffmpeg` command executed to compute the PSNR and VMAF for quality assessment between an original video and its encoded version.

```
ffmpeg -i video.mp4 -i enc-video.mp4 -lavfi
 [0:v][1:v]libvmaf=model_path=./vmaf_4k_v0.6.1.json
 :log_fmt=json:log_path=log.json:psnr=1 -f null -
```

LISTING 2. Sample ffmpeg command used to obtain PSNR and VMAF.

All experiments were conducted in a controlled environment to ensure fairness and consistency. Both CPU and GPU encoding tasks were executed within the same container, with no other processes being executed in the container. Besides, no other containers were executed during the experiments at each node to avoid CPU resource contention. This setup guarantees that the comparison between CPU and GPU encoders is accurate and solely reflects their performance differences.

In the subsequent phase of our analysis, each encoded video was compared with its original ultra-high definition 4K raw source. This process was essential to extract objective quality metrics, specifically PSNR, and VMAF. From these metrics, VMAF provides an overall score that reflects the subjective assessment of human observers, making it a more comprehensive measure of video quality.

With the bitrate and the quality it is possible to compute the Bjøntegaard Delta metrics. The BD metrics are a pivotal element of our study, offering valuable insights into codec performance disparities. They achieve this by quantifying the variance in quality at an equivalent bitrate, as well as the difference in bitrate necessary to maintain a consistent quality level.

To guarantee the accuracy of our BD metrics, we utilized Akima interpolation. This method allowed us to construct smooth bitrate-quality curves from our collected discrete data points. Akima interpolation produces less error in the interpolation when compared with other interpolation methods [12], which is vital for a precise computation of the subtle differences in codec performance. The interpolated curves not only illustrate the rate-distortion traits of each codec but also facilitate a comparative analysis, underscoring the influence of GPU acceleration on the encoding workflow.

In all the experiments, for each pair of codecs we have computed BD-Rate and BD-PSNR for PSNR quality metric, alongside BD-Rate and BD-VMAF for VMAF quality metric. To compute the BD metrics with the VMAF quality we followed the recommendation to transform the VMAF to a logarithmic scale [12], as in (1):

$$-10\log_{10}\left(1-\frac{\text{VMAF}}{100}\right) \tag{1}$$

The insight for the interpretation of the data in the tables is: a positive delta value in BD-Rate indicates that the test codec requires more bitrate than the anchor codec to achieve the same quality (better the anchor). A negative delta value in BD-Quality indicates that the test codec produces less quality to achieve the same bitrate (better the anchor).

A. H264/AVC CODECS: H264_NVENC VS LIBX264

Table 3 provides the Bjøntegaard Delta metrics between two H.264/AVC implementations: h264_nvenc as the anchor codec and libx264 as the test codec.

TABLE 3. Table comparing BD-Rate and BD-Quality for h264_nvenc and libx264.

Video	PSNR		VMAF	
	BD-Rate (%)	BD-PSNR (dB)	BD-Rate (%)	BD-VMAF
Bosphorus	21.3	-0.81	38.22	-4.59
HoneyBee	-17.21	-0.07	-12.85	-0.68
Jockey	-10.48	-0.66	-0.27	-2.07
ReadySetGo	-6.2	-0.96	-2.69	-3.87
ShakeNDry	14.84	-0.49	36.20	-3.65
YachtRide	12.39	-0.80	20.53	-3.62
CityAlley	-9.34	0.22	27.41	-0.18
FlowerFocus	64.60	-0.21	65.49	-1.72
FlowerKids	9.71	-0.36	14.61	-1.74
FlowerPan	14.32	0.10	54.01	-0.12
Lips	181.40	-0.56	110.70	-6.65
RaceNight	12.73	-0.43	12.60	-1.90
RiverBank	6.10	-0.19	62.39	-4.58
SunBath	28.68	-1.85	37.74	-7.49
Twilight	14.84	-0.13	54.69	-1.06

Taking into account the data of the column BD-Rate for the PSNR in Table 3, we see that h264_nvenc outperforms libx264 in 10 out of the 15 videos. The mean value for that column indicates that libx264 needs a 23.19% more bitrate to achieve the same quality as the one obtained using h264_nvenc for the videos considered. Taking into account the data of column BD-Rate for the VMAF in Table 3, the number increases to 12 videos, and the mean value for that column is 34.58. In terms of BD-PSNR h264_nvenc outperforms libx264 in 12 out of the 15 videos, being the mean value for that column -0.48 db. If the VMAF quality is considered, h264_nvenc outperforms libx264 in 14 videos, and the mean in that case is -2.98. Therefore, in most cases h264_nvenc outperforms libx264 either considering PSNR or VMAF in both BD-Rate and BD-Quality.

Figure 2 displays the rate-distortion curves and the interpolation computed for these codecs with the PSNR metric for all the videos and bitrates considered in the experiment.

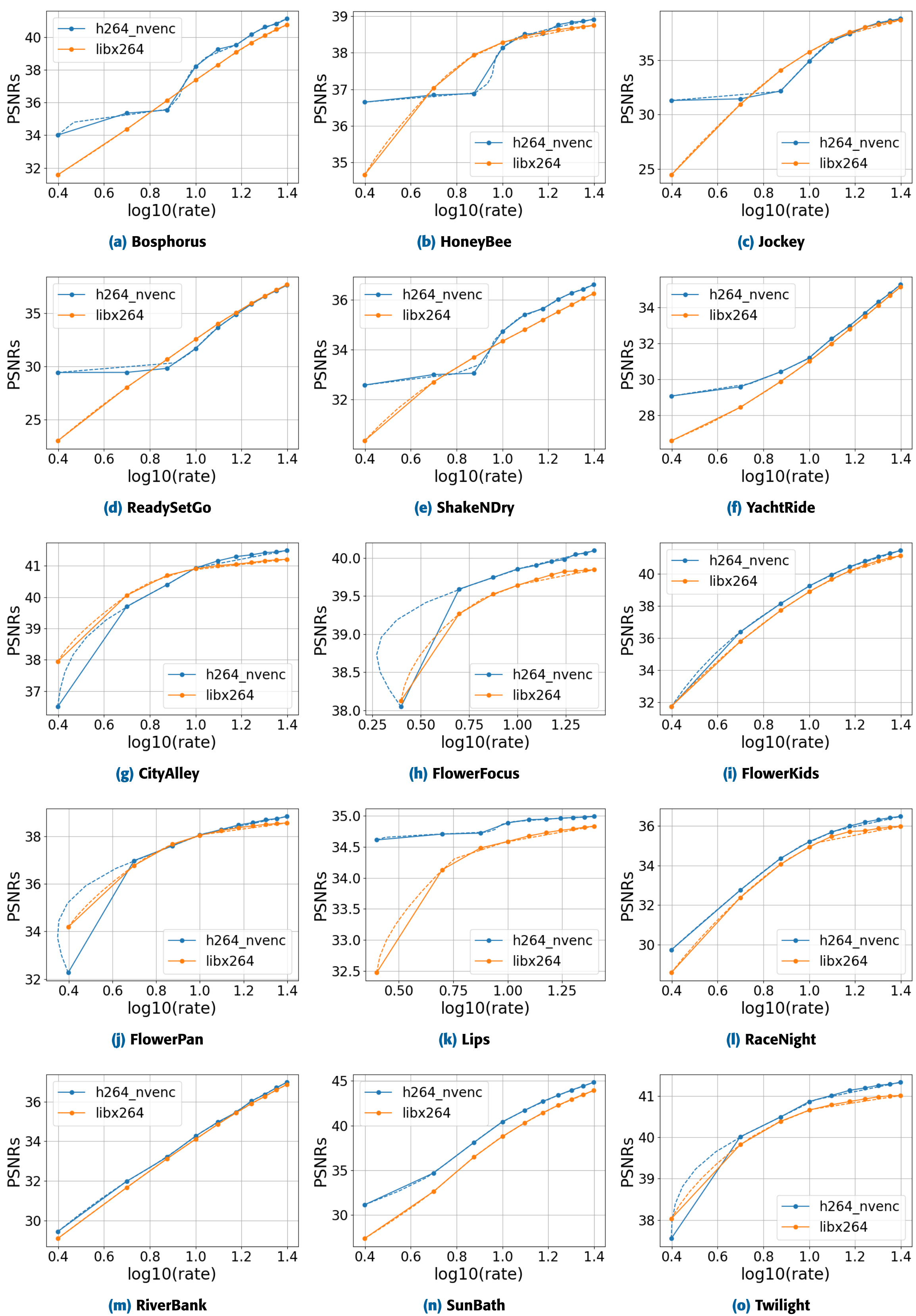


FIGURE 2. Rate-Distorsion points and Akima interpolation to compute DB-Rate and BD-Quality for h264_nvenc and libx264.

TABLE 4. Table comparing BD-Rate and BD-Quality for hevc_nvenc and libx265.

Video	PSNR		VMAF	
	BD-Rate (%)	BD-PSNR (dB)	BD-Rate (%)	BD-VMAF
Bosphorus	-16.21	0.47	-3.56	0.42
HoneyBee	-58.21	0.68	-60.2	4.43
Jockey	-3.79	0.04	5.81	-0.73
ReadySetGo	-20.28	0.63	-22.07	1.8
ShakeNDry	13.55	-0.25	20.93	-1.78
YachtRide	-16.79	0.44	-8.22	0.38
CityAlley	-41.24	0.57	-42.46	0.9
FlowerFocus	13.13	0.03	41.08	-0.09
FlowerKids	-6.37	0.20	-5.22	0.51
FlowerPan	-25.94	0.45	-19.38	1.15
Lips	77.02	-0.03	-10.56	-0.17
RaceNight	-4.95	0.09	-4.18	0.41
RiverBank	-14.72	0.45	-3.35	0.39
SunBath	13.23	-0.61	10.02	-1.24
Twilight	-22.00	0.29	-12.09	0.55

B. H265/HEVC CODECS: HEVC_NVENC VS LIBX265

Table 4 presents the Bjøntegaard Delta metrics obtained for two HEVC/H.265 encoder implementations: hevc_nvenc as the anchor codec and libx265 as the test codec.

From the perspective of the BD-Rate using the PSNR, the data in Table 4 indicates that h265_nvenc is better than libx265 in only 4 of the 15 videos. The mean for that column indicates that libx265 consumes 7.57% less bitrate than h265_nvenc to achieve the same quality. When considering VMAF, h265_nvenc outperforms libx265 in 6 out of the 15 videos. The mean in this case is -7.56, which is very similar to the one obtained using PSNR. Therefore, the differences between the codecs are not very pronounced. This conclusion is also supported by BD-PSNR and BD-VMAF, with mean differences of 0.23 and 0.462 respectively.

Figure 3 shows the rate-distortion graphs and the interpolation computed, using PSNR as quality measure, for these two implementations of H265/HEVC.

C. AV1 CODECS: AV1_NVENC VS. LIBRAV1E

Table 5 shows the Bjøntegaard Delta metrics obtained for two AV1 encoder implementations: av1_nvenc as the anchor and librav1e as the test.

Analizing the DB-Rate data, for PSNR av1_nvenc showed better performance than librav1e in 7 out of the 15 video samples, which represents about 46.67% of the cases. However, the mean value indicates that librav1e required 2.71% more bitrate which is not a substantial increment. For VMAF, av1_nvenc outperforms librav1e in 12 out of the 15 cases, accounting for approximately 80% of the videos. In this case, codec librav1e required, in mean, 24.43% more bitrate than av1_nvenc to achieve the same VMAF quality compared to av1_nvenc.

Figure 4 illustrates their rate-distortion relationship, highlighting the efficiency of each approach and providing a clear visual and numerical comparison through the BD metrics.

TABLE 5. Table comparing BD-Rate and BD-Quality for av1_nvenc and librav1e.

Video	PSNR		VMAF	
	BD-Rate (%)	BD-PSNR (dB)	BD-Rate (%)	BD-VMAF
Bosphorus	-1.9	0.03	9.9	-0.31
HoneyBee	-40	0.31	-32.16	0.52
Jockey	-4.84	0.01	1.57	-0.17
ReadySetGo	5.24	-0.29	12.57	-1.08
ShakeNDry	0.51	-0.01	18.7	-0.38
YachtRide	-1.61	0	14.63	-0.38
CityAlley	-36.16	0.53	-18.64	0.46
FlowerFocus	10.75	0.1	37.24	-0.15
FlowerKids	9.06	-0.20	25.69	-1.86
FlowerPan	-16.17	0.39	8.37	0.19
Lips	148	-0.05	260.88	-1.15
RaceNight	7.47	-0.1	21.43	-1.87
RiverBank	3.55	-0.09	18.7	-0.5
SunBath	-9.53	0.48	4.11	-0.54
Twilight	-33.76	0.38	-16.47	0.49

D. GPU CODECS COMPARISON: HEVC_NVENC VS AV1_NVENC

Table 6 provides a comprehensive comparison between two different Nvidia codecs implementations av1_nvenc as the anchor codec and hevc_nvenc as testing codec.

TABLE 6. Table comparing BD-Rate and BD-Quality for av1_nvenc and hevc_nvenc.

Video	PSNR		VMAF	
	BD-Rate (%)	BD-PSNR (dB)	BD-Rate (%)	BD-VMAF
Bosphorus	45.54	-0.86	54.34	-1.26
HoneyBee	46.29	-0.42	55.0	-1.45
Jockey	62.71	-0.86	63.4	-3.09
ReadySetGo	25.92	-0.82	21.07	-1.44
ShakeNDry	32.52	-0.46	36.59	-0.61
YachtRide	29.19	-0.71	29.1	-0.49
CityAlley	-2.04	-0.05	0.42	-0.09
FlowerFocus	51.05	-0.14	86.36	-1.29
FlowerKids	20.67	-0.52	17.83	-1.41
FlowerPan	-11.42	0.04	-11.79	0.26
Lips	148.60	0.07	504.68	-1.71
RaceNight	25.36	-0.37	28.39	-2.54
RiverBank	13.95	-0.37	17.56	-0.49
SunBath	25.75	-1.10	30.11	-3.63
Twilight	-5.6	-0.03	15.81	-0.75

Considering the BD-Rate data, from the PSNR perspective, the data in Table 6 shows that av1_nvenc outperforms hevc_nvenc in 12 out of the 15 video samples, translating to about 80% of the cases. The mean value for that column is 33.90 which implies that hevc_nvenc required in mean 33.90% more bitrate to achieve the same PSNR than av1_nvenc. The data for VMAF shows similar results: av1_nvenc outperforms hevc_nvenc in 13 videos. Taking the mean value of that column we observe that hevc_nvenc required in mean 63.26% more bitrate to achieve the same VMAF than av1_nvenc.

For BD-Quality with PSNR, av1_nvenc outperforms hevc_nvenc in 12 video, with a mean loss in quality of -0.44 dB for hevc_nvenc. If the quality is measured using VMAF av1_nvenc outperforms hevc_nvenc in 13 videos, with a mean loss in quality of -1.33 dB for hevc_nvenc. Figure 5 contrasts their performance.

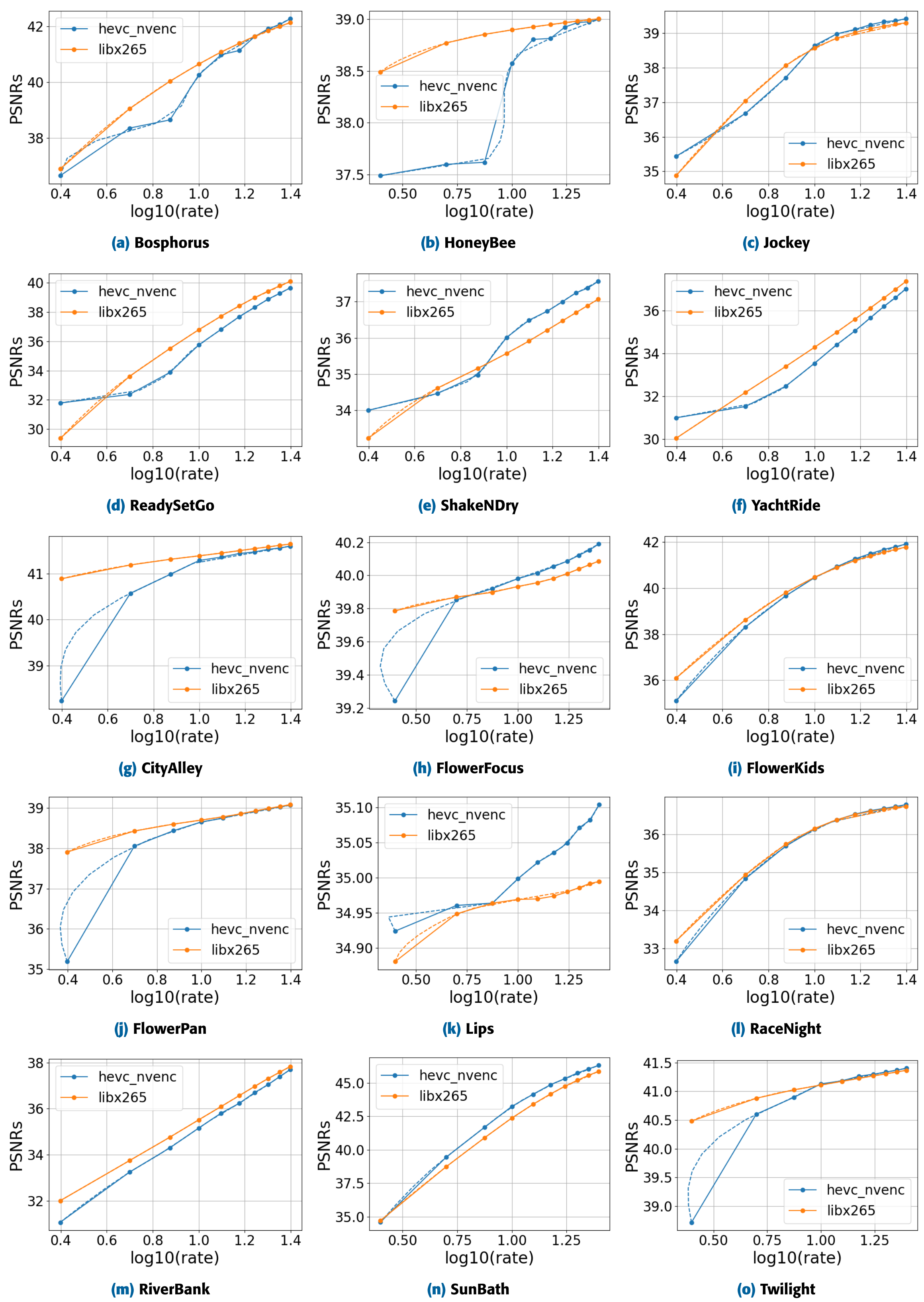


FIGURE 3. Rate-Distorsion points and Akima interpolation to compute DB-Rate and BD-Quality for hevc_nvenc and libx265.

(a) Bosphorus

(b) HoneyBee

(c) Jockey

(d) ReadySetGo

(e) ShakeNDry

(f) YachtRide

(g) CityAlley

(h) FlowerFocus

(i) FlowerKids

(j) FlowerPan

(k) Lips

(l) RaceNight

(m) RiverBank

(n) SunBath

(o) Twilight

FIGURE 4. Rate-Distorsion points and Akima interpolation to compute DB-Rate and BD-Quality for av1_nvenc and librav1e.

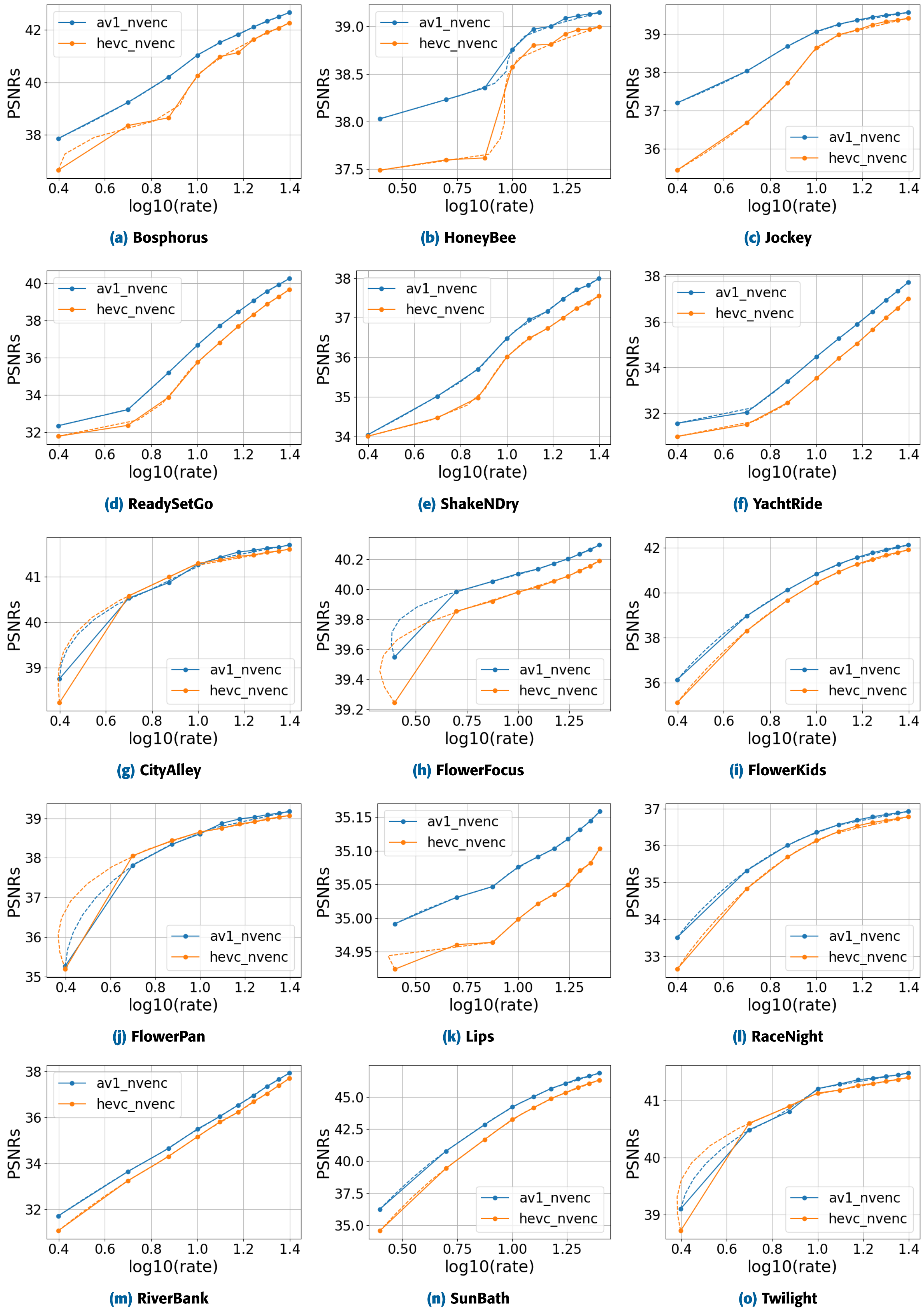


FIGURE 5. Rate-Distorsion points and Akima interpolation to compute DB-Rate and BD-Quality for av1_nvenc and hevc_nvenc.

E. BD-RATE DISCUSSION

The performance of video codecs, as reflected in BD-Rate measurements, varies significantly across different video sequences due to the inherent complexity of the content. Videos with high spatial detail, such as intricate textures and fine patterns, demand more bits to maintain quality, making them more challenging to encode efficiently. Similarly, sequences with high temporal complexity, characterized by fast motion or erratic changes, reduce the effectiveness of motion estimation and prediction, leading to higher bitrates. In contrast, homogeneous content, such as static or uniform scenes, can be compressed more effectively, resulting in better BD-Rate performance. Additionally, abrupt scene changes disrupt temporal prediction mechanisms, often causing encoding inefficiencies and increased bit usage.

The variability in BD-Rate results is also influenced by the design and features of the codecs. Modern codecs, such as AV1 and HEVC, utilize advanced adaptive mechanisms for quantization, transforms, and prediction, which can vary in efficiency depending on the characteristics of the input video. Furthermore, differences in rate-distortion optimization strategies across codecs affect how well they exploit spatial, temporal, and statistical redundancies in the content. Bitrate allocation strategies also play a crucial role, as the distribution of bits across frames impacts a codec's ability to maintain consistent quality, particularly for sequences with fluctuating complexity. These factors highlight the need for evaluating codecs across a diverse range of video sequences to ensure a comprehensive understanding of their performance.

Variations in frame rate (fps) and duration significantly impact BD-Rate results due to their influence on temporal redundancy, motion prediction, and bitrate allocation. High-FPS videos typically offer better temporal prediction efficiency due to smaller gaps between frames, but fast or erratic motion can still challenge codecs. Low-FPS videos, with larger temporal gaps, often reduce prediction accuracy, leading to higher bitrates. Similarly, short videos, especially those with abrupt changes, may exhibit higher BD-Rate variability due to limited opportunities for optimal rate allocation. In contrast, longer videos often include more diverse content and allow for smoother bit distribution, potentially stabilizing BD-Rate results.

In addition to content and codec characteristics, the sensitivity of the evaluation metrics used in BD-Rate calculations also contributes to performance variability. Metrics such as PSNR, commonly used to measure video quality, respond differently to specific video features, such as fine textures or motion artifacts. Moreover, these metrics may not always align with human perceptual quality, which can amplify differences in observed performance. This highlights the importance of complementing traditional objective metrics with perceptual measures, such as VMAF, to achieve a more comprehensive assessment of codec efficiency and quality preservation.

F. CODECS COMPARISON: PSNR

In light of the remarks made in the preceding sections, Figure 6 shows all the bitrate-quality curves for all the codecs and all the videos. In this case, the quality was measured using the PSNR.

AV1 codecs exhibits superior performance, particularly the av1_nvenc codec, which excels in the majority of video tests. HEVC codecs ranks next, offering comparable quality to the AV1 codecs according to the PSNR metric. However, a closer examination of VMAF metrics reveals a different story, as illustrated in Figure 7. In contrast, H264/AVC codecs deliver the least quality at equivalent bitrates, with the libx264 codec recording the lowest PSNR values across the tested bitrates.

G. CODECS COMPARISON: VMAF

Figure 7 shows all the bitrate-quality curves for all the codecs and all the videos. In this case, the quality was measured using the VMAF.

Overall, within each codec family (AV1, HEVC, and H.264/AVC), GPU-implemented codecs generally achieve higher VMAF scores than their CPU counterparts at the same target bitrate. AV1_nvenc demonstrates the highest performance, delivering superior quality due to optimized GPU encoding processes, followed closely by librav1e. However, in specific low-bitrate scenarios, HEVC implementations can attain comparable or even higher VMAF scores than AV1, suggesting greater efficiency at conserving quality under constrained data rates. On the other hand, libx264 consistently shows the least effective quality-to-bitrate ratio, indicating limitations in H.264/AVC efficiency compared to newer standards.

H. TIMES COMPARISON

This section will examine the encoding times of the codecs analyzed in this study. As previously noted, these times are contingent on the infrastructure where the encoding is performed. However, this analysis can provide insight into the relative efficiency of different codecs on the same platform.

Table 7 shows the times in milliseconds (ms) per frame required to encode each video at a bitrate of 25 Mbps, as well as the percentage improvement of the GPU codec over the CPU codec of the same family.

A comparison of the average values presented in the final row of Table 7 reveals that the libx264 codec requires an average of 139.07 milliseconds per frame, whereas the h264_nvenc reduces this average to 109.69 milliseconds. Consequently, the h264_nvenc codec is, on average, 1.30 times faster than the libx264 codec. In the case of the HEVC codec family, it can be observed that the libx265 codec requires an average of 240.4 ms per frame, while the hevc_nvenc codec requires an average time of 108.39 ms per frame to perform the same task. In this instance, the hevc_nvenc codec is 2.35 times faster on average than the libx265 codec. Upon analysis of the AV1 family

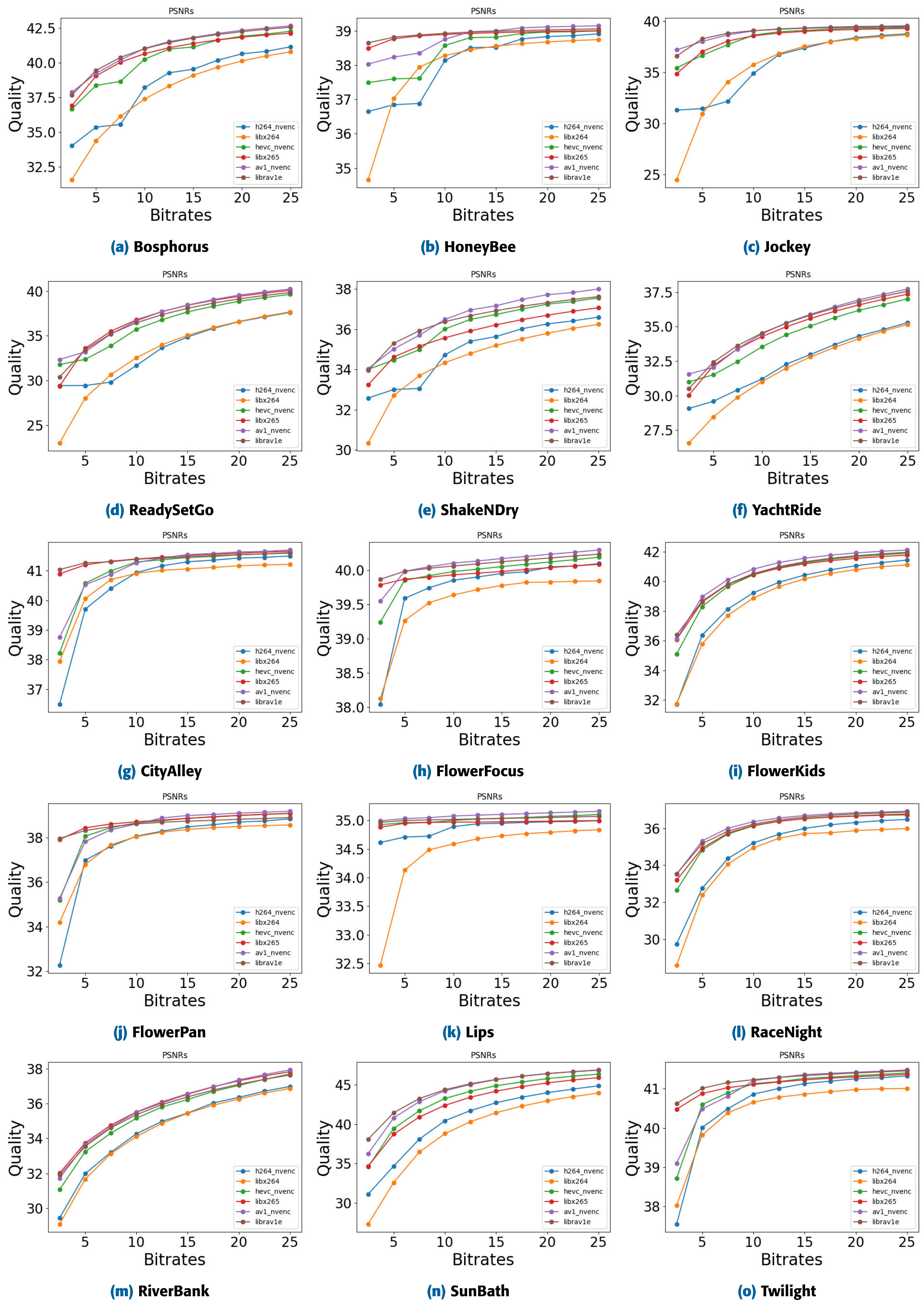


FIGURE 6. **Bitrate-PSNR curves for all the codecs and videos.**

(a) Bosphorus

(b) HoneyBee

(c) Jockey

(d) ReadySetGo

(e) ShakeNDry

(f) YachtRide

(g) CityAlley

(h) FlowerFocus

(i) FlowerKids

(j) FlowerPan

(k) Lips

(l) RaceNight

(m) RiverBank

(n) SunBath

(o) Twilight

FIGURE 7. Bitrate-VMAF curves for all the codecs and videos.

TABLE 7. Times and factor improvement between codecs of the same family.

Name	libx264 (ms/frame)	h264_nvenc (ms/frame)	Factor GPU vs. CPU	libx265 (ms/frame)	hevc_nvenc (ms/frame)	Factor GPU vs. CPU	librav1e (ms/frame)	av1_nvenc (ms/frame)	Factor GPU vs. CPU
Bosphorus	116.42	93.57	1.24	208.95	86.42	2.42	4361.20	102.44	42.57
HoneyBee	123.03	114.36	1.08	222.38	122.50	1.82	8091.18	144.47	56.01
Jockey	143.25	98.98	1.45	267.53	102.15	2.62	5148.62	118.36	43.50
ReadySetGo	178.06	149.30	1.19	202.34	100.59	2.01	5830.54	94.32	61.82
ShakeNDry	130.71	119.18	1.10	203.00	116.41	1.74	3883.01	119.99	32.36
YachtRide	118.74	98.35	1.21	234.55	109.04	2.15	4501.35	93.47	48.16
CityAlley	143.03	97.56	1.47	205.32	99.21	2.07	12351.46	121.93	101.30
FlowerFocus	137.17	116.43	1.18	284.53	120.13	2.37	10608.62	145.43	72.95
FlowerKids	122.95	93.98	1.31	217.22	113.27	1.92	9153.12	102.60	89.21
FlowerPan	142.73	117.12	1.22	219.61	113.63	1.93	10533.68	132.60	79.44
Lips	155.94	146.05	1.07	232.91	121.74	1.91	4715.39	124.39	37.91
RaceNight	153.82	121.02	1.27	279.10	135.40	2.06	8534.09	139.40	61.22
RiverBank	114.33	109.34	1.05	205.63	115.75	1.78	7857.04	142.67	55.07
SunBath	149.25	68.65	2.17	403.21	63.46	6.35	10356.18	65.38	158.40
Twilight	156.65	101.48	1.54	219.75	106.14	2.07	9796.44	119.09	82.26
Average	*139.07*	*109.69*	*1.30*	*240.40*	*108.39*	*2.35*	*7714.79*	*117.77*	*68.14*

codecs, it becomes evident that the GPU implementation (av1_nvenc) offers a substantial enhancement over the fast CPU implementation of AV1 (librav1e). In this instance, the librav1e codec requires an average of 7714.79 milliseconds to encode a 4K frame, whereas the av1_nvenc codec completes the same task in 117.77 milliseconds being 68.14 times faster than the librav1e implementation. Finally, the GPU implementation of the encoders do not show much variation, the h264_nvenc codec averages 109.69 ms per frame, closely followed by hevc_nvenc at 108.39 ms and av1_nvenc at 117.77 ms. However, the CPU implementations show big differences, libx264 takes an average of 139.07 ms per frame. This increases to 240.40 ms for libx265, which is 1.78 times slower, and skyrockets to 7714.79 ms for librav1e, which is 55.4 times slower than libx264 and 32.1 times slower than libx265.

V. CONCLUSION AND FUTURE WORK

Video encoding has a big impact in streaming, video conferencing, and multimedia storage, specially as the demand for high-quality and high-resolution content continues to grow. The goal is to reduce the bandwidth required for transmission and to lower storage costs, while also aiming to maintain or improve the viewing experience. The demand for high resolutions require the use of the latest generation of codecs. Furthermore, if these contents are to be broadcast in live events, it is necessary to use fast and efficient codecs which can only be achieved with the use of GPUs.

This work has compared the coding efficiency of video encoders (AVC/H264, HEVC/H265, and AV1) on a set of 4K videos. For each encoder, two implementations were selected, one that runs on CPUs and another that uses GPUs, and the Bjøntegaard rate and quality (using PSNR and VMAF) was computed. Besides, the mean encoding time for each encoder was obtained.

The BD rate comparison shows that the CPU version of the encoders require more than 20% of bitrate than the GPU implementation (except for H265 where the GPU version require a 7% more bitrate). From the rate-distortion curves we can see that the AV1 family is better although HEVC implementations can sometimes match or exceed AV1 in VMAF at low bitrates and the libx264 codec consistently performs the worst. From the encoding time results, we observe that the encoder that achieves a higher gain when using the GPU version is AV1, which is 68.14 times faster than the CPU implementation. For, H264 the gain was 1.34 times and for H265 the gain was 2.35 times.

CPU-specific codecs show considerable variance in encoding times, with libx265 being 1.78 times slower than libx264 and librav1e being drastically slower. In contrast, GPU-based codecs exhibit minimal variance, with differences of less than 10 milliseconds. This demonstrates the superior efficiency and consistency of GPU implementations for video encoding.

In the future, we intend to integrate GPU codecs in order to enable live multiresolution encoding at the cloud edge for HTTP Adaptive Streaming (HAS) applications. Additionally, we are developing a serverless platform for field-of-view (FoV) based encoding of 360-degree VR videos. This initiative aims to leverage GPU codecs in order to improve the efficiency and flexibility of video encoding processes, particularly for immersive 360-degree virtual reality content.

ACKNOWLEDGMENT

The authors would like to express their gratitude to Prof. Miguel Arevalillo-Herráez and his research team for kindly lending them the Nvidia 4090 RTX graphics card for the experiments.

REFERENCES

[1] *2024 Global Internet Phenomena Report*, Sandvine, Waterloo, ON, USA, 2024.

[2] L. Yu, S. Chen, and J. Wang, "Overview of AVS-video coding standards," *Signal Process., Image Commun.*, vol. 24, no. 4, pp. 247–262, Apr. 2009.

[3] C. Bonnineau, W. Hamidouche, J. Fournier, N. Sidaty, J.-F. Travers, and O. Déforges, "Perceptual quality assessment of HEVC and VVC standards for 8K video," *IEEE Trans. Broadcast.*, vol. 68, no. 1, pp. 246–253, Mar. 2022.

[4] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 560–576, Jul. 2003.

[5] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1649–1668, Dec. 2012.

[6] Y. Chen et al., "An overview of coding tools in AV1: The first video codec from the alliance for open media," *APSIPA Trans. Signal Inf. Process.*, vol. 9, no. 1, pp. 1–15, 2020.

[7] S. Katsigiannis, V. Dimitsas, and D. Maroulis, "A GPU vs CPU performance evaluation of an experimental video compression algorithm," in *Proc. 7th Int. Workshop Quality Multimedia Exper. (QoMEX)*, May 2015, pp. 1–6.

[8] J. Žádník, M. Mäkitalo, J. Vanne, and P. Jääskeläinen, "Image and video coding techniques for ultra-low latency," *ACM Comput. Surveys*, vol. 54, no. 11s, pp. 1–35, Sep. 2022.

[9] W. Moina-Rivera, M. Garcia-Pineda, J. Gutiérrez-Aguado, and J. M. Alcaraz-Calero, "Cloud media video encoding: Review and challenges," *Multimedia Tools Appl.*, vol. 83, no. 34, pp. 81231–81278, Mar. 2024.

[10] M. Vranješ, S. Rimac-Drlje, and K. Grgić, "Review of objective video quality metrics and performance comparison using different databases," *Signal Process., Image Commun.*, vol. 28, no. 1, pp. 1–19, Jan. 2013.

[11] N. Barman, M. G. Martini, and Y. Reznik, "Bjøntegaard delta (BD): A tutorial overview of the metric, evolution, challenges, and recommendations," 2024, *arXiv:2401.04039*.

[12] C. Herglotz, H. Och, A. Meyer, G. Ramasubbu, L. Eicherüller, M. Kränzler, F. Brand, K. Fischer, D. T. Nguyen, A. Regensky, and A. Kaup, "The bjøntegaard bible why your way of comparing video codecs may be wrong," *IEEE Trans. Image Process.* vol. 33, pp. 987–1001, 2024.

[13] I. E. Richardson, *The H.264 Advanced Video Compression Standard*. Hoboken, NJ, USA: Wiley, 2003.

[14] *AV1 Beats X264 and Libvpx-vp9 in Practical Use Case*, Facebook Eng., Menlo Park, CA, USA, 2018.

[15] Alliance Open Media. (2019). *A Technical Overview of AV1*. [Online]. Available: https://aomedia.org/

[16] M. Uhrina, L. Sevcik, J. Bienik, and L. Smatanova, "Performance comparison of VVC, AV1, HEVC, and AVC for high resolutions," *Electronics*, vol. 13, no. 5, p. 953, Mar. 2024.

[17] O. Nawaz, M. Fiedler, and S. Khatibi, "QoE-based performance comparison of AVC, HEVC, and VP9 on mobile devices with additional influencing factors," *Electronics*, vol. 13, no. 2, p. 329, Jan. 2024.

[18] T. Amestoy, N. Sidaty, W. Hamidouche, P. Philippe, and D. Menard, "Video quality assessment and coding complexity of the versatile video coding standard," 2023, *arXiv:2310.13093*.

[19] C. Herglotz, M. Kranzler, R. Mons, and A. Kaup, "Beyond Bjøntegaard: Limits of video compression performance comparisons," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Oct. 2022, pp. 46–50.

[20] W. Xiao, B. Li, J. Xu, G. Shi, and F. Wu, "HEVC encoding optimization using multicore CPUs and GPUs," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 11, pp. 1830–1843, Nov. 2015.

[21] B. Su, B. Cheng, and J. Chen, "GPU based high definition parallel video codec optimization in mobile device," *IEEE Trans. Mobile Comput.*, vol. 22, no. 6, pp. 3333–3349, Jun. 2023.

[22] A. Salcedo-Navarro, R. Peña-Ortiz, J. M. Claver, M. Garcia-Pineda, and J. Gutiérrez-Aguado, "Cloud-native GPU-enabled architecture for parallel video encoding," in *Proc. Eur. Conf. Parallel Process.* Cham, Switzerland: Springer, 2024, pp. 327–341.

[23] A. Mercat, M. Viitanen, and J. Vanne, "UVG dataset: 50/120fps 4K sequences for video codec analysis and development," in *Proc. 11th ACM Multimedia Syst. Conf.*, May 2020, pp. 297–302.

ANDONI SALCEDO-NAVARRO received the B.Sc. degree in computer engineering with a specialization in computing from the University of Zaragoza, and the M.Sc. degree in web technologies, cloud computing, and mobile applications (TWCAM) from the University of Valencia. He is currently pursuing the Ph.D. degree, with a research focus on multimedia content distribution in cloud-edge environments. His research interests include cloud-based video streaming and he has published several articles on this topic.

JUAN GUTIÉRREZ-AGUADO received the Ph.D. degree in computer science from the Universitat de València. He is currently an Associate Professor with the Universitat de València. He has taught undergraduate and graduate courses on image processing, programming, mobile devices, server-side programming, and cloud computing. He is the Director of the Master in Web Technologies, Cloud Computing, and Mobile Applications. He has authored or co-authored of journal papers in computer vision and image processing and recently in cloud monitoring and server less platforms for video encoding. His current research interests include distributed and cloud computing.

MIGUEL GARCIA-PINEDA received the M.Sc. and Ph.D. degrees in telecommunication engineering from the Universitat Politècnica de València, in 2008 and 2013, respectively. He is currently the Vice Dean and an Associate Professor with the Department of Computer Science, School of Engineering (ETSE-UV), Universitat de València. He has been involved in several research projects and has authored more than 110 papers in international conference proceedings and over 50 peer-reviewed articles in high-impact international journals. His research interests include multimedia encoding and streaming, quality of service and quality of experience, and cloud computing. In addition, he is an active associate editor and a reviewer of various international journals.

Capítulo 8

Towards GPU-enabled serverless cloud edge platforms for accelerating HEVC video coding

Cluster Computing (2025) 28:68
https://doi.org/10.1007/s10586-024-04692-0

Towards GPU-enabled serverless cloud edge platforms for accelerating HEVC video coding

Andoni Salcedo-Navarro[1] · Raúl Peña-Ortiz[1] · Jose M. Claver[1] · Miguel Garcia-Pineda[1] · Juan Gutiérrez-Aguado[1]

Received: 16 April 2024 / Revised: 12 June 2024 / Accepted: 28 September 2024 / Published online: 5 November 2024
© The Author(s) 2024

Abstract

Multimedia streaming has become integral to modern living, reshaping entertainment consumption, information access, and global engagement. The ascent of cloud computing, particularly serverless architectures, plays a pivotal role in this transformation, offering dynamic resource allocation, parallel execution, and automatic scaling-especially beneficial in HTTP Adaptive Streaming (HAS) applications. This study presents an event-driven serverless cloud edge platform with graphics processing units (GPUs), managed by Knative, tailored for video encoding. Two implementations of the High Efficiency Video Coding (HEVC) codec have been encapsulated in the functions: HEVC NVENC (Nvidia Encoder), that uses GPU acceleration, and x265 that only uses CPUs. Experiments focused on measuring the impact of replica requested resources on cold start, scalability and resource consumption with different allocated resources on slim and fat virtual machines (VMs). The best results are obtained when four slim replicas of the functions are deployed on a fat VM with a 8.4% reduction in encoding time for x265 and a 15.2% improvement for HEVC NVENC compared with other deployment scenarios. Comparatively, HEVC NVENC encoding is 8.3 times faster than x265. In multiresolution scenarios, GPU encoding drastically reduces segment encoding time by a factor of 12.4 between non-GPU and GPU-accelerated. These findings are important for live streaming applications where low latency is critical at all stages of the streaming process.

Keywords Serverless · FaaS · GPU · Video encoding · Cloud edge · HEVC · NVENC

1 Introduction

In the digital age, multimedia streaming has become an integral part of our daily lives, fundamentally transforming the way we consume and share content. This technology allows us to access a wide range of audio and video content, from music and movies to live sports events and educational materials, either on-demand or live. The importance of multimedia streaming in today's world cannot be overstated, as it has revolutionised the entertainment industry, education, communication, and business in countless ways [1]. For example, one of the most popular forms of multimedia entertainment today is through online streaming platforms becoming serious rivals to the broadcasting and television industry. The demand for regular television broadcasting is declining as these platforms provide high-quality original productions at the convenience of the customer's home. The shift from linear TV to on-demand viewing, the rise in binge-watching, and the increase in diversity of content due to original programming are some of the changes brought about by streaming services.

In terms of streaming technology, HTTP Adaptive Streaming (HAS) has become the dominant video delivery technology over the Internet. In HAS, videos are split into

✉ Miguel Garcia-Pineda
miguel.garcia-pineda@uv.es

✉ Juan Gutiérrez-Aguado
juan.gutierrez@uv.es

Andoni Salcedo-Navarro
andoni.salcedo@uv.es

Raúl Peña-Ortiz
raul.penya@uv.es

Jose M. Claver
jose.claver@uv.es

[1] Departament d'Informàtica, Universitat de València, Avda. de la Universitat, s/n, 46100 Burjassot, València, Spain

short intervals called segments, and each segment is encoded at various qualities/bitrates to adapt to the available bandwidth. These streaming systems necessitate efficient video encoding systems capable of generating content in a multitude of resolutions, bit rates, and codecs, with the aim of providing an optimal Quality of Experience (QoE) to end-users [2].

One of the key factors that has facilitated this transformation is the seamless integration of multimedia streaming with cloud computing [3]. Cloud computing has emerged as a game-changer for multimedia streaming, serving as a robust and scalable infrastructure for delivering content to users across the globe [4]. This technology has allowed individuals and organisations to exploit the power of the cloud to store, process, and deliver multimedia content efficiently and cost-effectively by harnessing:

- Scalability: The demand for multimedia content can vary dramatically, with spikes during popular events or viral content releases. Cloud computing enables content providers to dynamically scale their infrastructure up or down to accommodate these fluctuations in demand.
- Global Reach: Cloud-based multimedia streaming services can easily reach a worldwide audience by leveraging the distributed nature of cloud data centers. This allows content providers to minimise latency and deliver high-quality streaming experiences to users in various regions.
- Cost Efficiency: The cloud computing model offers cost benefits to multimedia streaming providers. Instead of investing in and maintaining their own hardware, they can rent resources on a pay-as-you-go basis, reducing capital expenditures and optimising operational costs.
- Flexibility: Cloud-based multimedia streaming services provide the flexibility to adapt to evolving industry standards and user preferences. Providers can quickly update their services, add new features, or support various devices without significant disruptions.
- Data Security: Cloud computing providers often invest heavily in security measures to protect data, making them a reliable choice for storing and transmitting multimedia content securely. This is particularly important in the context of copyrighted materials and user data protection.
- Content Delivery Networks (CDNs): Many multimedia streaming services leverage CDNs, which are a distributed network of servers in various data centers. These CDNs are often managed by cloud providers and are strategically positioned to cache and deliver content as close to the end-users as possible, further enhancing the streaming experience.
- Big Data Analytics: Cloud computing enables content providers to analyse user data and preferences, which can be used to personalise content recommendations and improve the overall user experience.

Within the various paradigms encompassed by the Cloud Computing concept, like Infrastructure-as-a- Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS), etc., serverless computing is a cloud computing paradigm that shifts the responsibility of managing servers from the user to the cloud provider. In Function-as-a-Service (FaaS), one subset in the serverless ecosystem, users only need to focus on writing and deploying individual functions or units of code. The cloud provider dynamically manages the execution environment and automatically scales resources up or down based on the actual demand [5]. The event-driven model of FaaS, coupled with its statelessness and automatic scaling, distinguishes it from traditional cloud computing models. Each function execution is isolated and stateless, allowing for efficient resource management.

Cost efficiency stands out as a primary benefit of serverless computing, thanks to its pay-as-you-go model. Users are charged solely for the resources consumed during function execution, eliminating costs associated with idle time [6]. Automatic scaling ensures optimal resource utilization, making serverless computing a cost-effective choice, especially for unpredictable workloads. Reduced operational complexity is another significant advantage, as serverless computing abstracts infrastructure management tasks. Cloud providers handle server provisioning, maintenance, and scaling, enabling developers to concentrate on code development rather than infrastructure upkeep. Developers can efficiently deploy and update functions without managing the intricacies of server infrastructure, leading to accelerated development cycles and quicker time-to-market for applications and features.

Moreover, serverless computing supports microservices architecture, enabling developers to build and deploy independent functions. This aligns seamlessly with a modular and scalable application design, contributing to the overall flexibility and adaptability of the architecture.

A promising application of serverless architectures is found in HAS-based video coding systems designed for video streaming (VoD and live). In these scenarios, the original video is segmented typically into short chunks lasting 1, 2, 4 s that are encoded for various bitrates and/or resolutions. For every encoding or set of encodings per chunk, a serverless platform can execute a code fragment encapsulated in a function, facilitating dynamic resource allocation. The serverless infrastructure enables parallel execution of encoding for each chunk. Moreover, it offers automatic scaling of replicas running the function to meet increased requirements, thereby potentially reducing the overall encoding time.

GPUs have been used in the video encoding process to improve performance and efficiency [7]. GPUs excel in handling intensive video tasks like encoding, thanks to their parallel processing. This means faster video processing, crucial for real-time streaming and broadcasting. GPUs' adaptability, support for multiple streams, and hardware-accelerated codecs make them vital for top-notch video encoding results.

For these reasons, our work presents and analyses a GPU video coding system built upon the serverless architecture paradigm characterised by streamlined automated or semi-automated deployment processes. Our proposal leverages the Knative open-source serverless platform [8] deployed on a resource-constrained infrastructure.

One of the key differences between the cloud edge and the cloud core is that in edge scenarios the nodes possess significantly fewer capabilities compared to powerful servers located in distant data centers. In addition, the number of available nodes is much lower in the edge. Therefore, with the use of the FaaS model in the cloud edge we are not taking advantage of the scaling capabilities but we are using the event management capabilities and concurrency control offered by Knative. The implemented system underwent testing across various scenarios to analyse the performance and resource consumption when encoding videos under real-world conditions, such as the dynamic adaptive streaming over HTTP (DASH) environment. To the best of our knowledge, there is a lack of studies that analyse the behaviour of encoders in serverless platforms with replicas of the function that make use of GPUs.

The main contributions of this work are:

- Proposal of a GPU serverless deployment for video encoding: architectural needs, configuration required, and development of serverless functions.
- Evaluation of different virtualisation scenarios to deploy serverless video encoding with and without the use of GPUs.
- Comparison of cold starts based on the computational resources requested by the function's replicas.
- Analysis of the behaviour of replicas under different scenarios and contrasting resource consumption and time taken for encoding a video.
- Analysis of the performance when multiple representations of each video chunk are generated in a single call.

This paper is organised as follows. Section 2 presents previous approaches related to the topics of this work. Section 3 presents the main components of the deployed serverless infrastructure. Section 4 describes the testbed, the six scenarios considered (with and without GPU), and the results obtained for cold start, encoding times and resource consumption. Then the best scenario is selected to analyse multiresolution encoding with a single event. Section 5 performs a comparison of this proposal with other previous works considering 12 aspects. Finally, Section 6 presents the conclusions and further work.

2 Related work

Proposed cloud-based solutions for distributed video coding and transcoding services have recently evolved to be adapted to the more demanding requirements and improve their performance [9].

The need of dynamic instantiation of the number of encoders has been crucial for the new video coding services, characterized by high demand and adaptive streaming applications. For this purpose, new cloud-based solutions adapting the number of encoders to the needs of video streaming, such as high video resolutions with bandwidth constraints and quality requirements have been proposed. Thus, Gutiérrez-Aguado et al. [10] proposed a cloud-based distributed architecture tuned for video encoding (HEVC, VP9 and AV1 encoders). Based on an elastic pool of workers and media servers that provide fault tolerance, this solution allows the adaptation of the resources to the workload, offers high availability, and provides ubiquitous access.

Enabling GPUs in cloud-based architectures have been proposed in different works. Some examples of using GPUs in cloud environments can be seen in Yang et al. [11], which use PCI pass-through technology to access the NVIDIA graphics card and show that the GPU performance is similar in native and virtual machines. The next advancement in the use of GPUs in cloud environments can be seen in Gutiérrez-Aguado et al. [12], which introduce a new architecture to cloudify existing GPUs, which do not support native virtualization.

Video coding has also benefited from the use of GPUs to speed up the increasing complexity and high time cost of new encoders. Thus, the use of GPUs has been proposed to accelerate video coding in standards such as H.264/AVC [13], HEVC [14], VP9 [15] and others. Moreover, Comi et al. [16] proposed using GPUs for video coding in cloud environments in the form of Network Function Virtualization (NFV), which facilitate the management and orchestration of network services. Thus, the GPU acceleration combined with NFV enhance the computing performance of commodity servers with the advantages offered by NFV in service management.

Nevertheless, these cloud-based solutions use virtual machines (VMs) to deploy and destroy the dynamic encoding workers used for the distributed video encoding, with a non-negligible time for this purpose. Linux containers, and specially Docker containers, as a lightweight

process, have been an alternative to VMs, reducing the deploying, and obtaining higher efficiency than with traditional VMs [17].

Following this approach, several architectures of containerized multimedia processing system for video transcoding based on Docker have been proposed (see subsection 4.3. in [9]). But the use of containers adds complexity due to orchestration and programming, hence container orchestration platforms (COPs) such as Docker Swarm, Nomad or Kubernetes [18] are used in cloud computing environments, being the latter the currently standard COP.

In the last decade, serverless computing has witnessed a significant upsurge primarily due to the cloud provider's management of dynamically allocated computing resources, coupled with a granular pay-per-use billing approach. It makes easier to write robust, large-scale web services [19], opening a new era in cloud computing. Amazon Web Services introduced AWS Lambda, the first serverless platform, on 2014, and similar abstractions are now available on most cloud computing platforms [20]. Moreover, several frameworks adopting the serverless approach have emerged to present open-source options as alternatives to what public cloud providers offer. Among these, notable platforms include OpenFaaS [21], Knative [22], Apache OpenWhisk [23], IronFunctions [24], among others. Furthermore, using on-premise serverless infrastructures as OSCAR [25], which is based on OpenFaaS, combined with the use of public cloud serverless as AWS Lambda, has advantages. But in some cases, as in edge computing, the on-premises solution can be better when a minimalist Kubernetes distribution and a fast even-driven functionality are required.

There are few, but interesting works, where video coding systems have been developed adopting the serverless approach. Fouladi et al. [26] present a video encoder (for VP8) aimed at fine-grained parallelism using a functional programming style that achieves fine-grained parallelism based on AWS Lambda and the Mu framework [27]. Sprocket [28] is a serverless-based video processing framework that extends the Mu framework to support dynamic creation of tasks during processing. It enables developers to program a series of operations over video content in a modular and extensible manner. More recently, Moina-Rivera et al. [29] presents an event-driven serverless functions to encode video (for H.264 and AV1). The authors deployed event-driven serverless pipelines for video coding and quality metrics (VMAF) on a local serverless platform based on Knative.

As we have seen in this section, and to the best of our knowledge, there are few works that use GPUs in serverless frameworks [30] and there is a lack of works that provide architectures and analysis of GPU integration in serverless environments and none for video encoding. For this reason, in the following sections, we will present our proposal and evaluate it to validate its correct operation. In section 5, our proposal is compared with previous works related to video encoding on serverless platforms to evaluate their differences and similarities.

3 Proposed architecture and serverless functions

This section presents the essential features of the Knative serverless platform, a pivotal element in our design strategy. We describe the platform's architecture and core components and the implementation of the event-driven serverless functions.

3.1 Serverless platform: Knative

Among the various open-source serverless platforms, Knative stands out for its unique capabilities, as highlighted in [31]. A standout feature of Knative is its ability to dynamically scale replicas based on concurrency, which aligns perfectly with our application's requirement of limiting each function replica to a single concurrent execution due to resource constraints. Knative further excels with its support for cold/warm starts, zero-scaling, and compatibility with CloudEvents [32], making it the ideal choice for our needs.

CloudEvents is a framework for streamlining event description and delivery across different services and platforms. This specification, under the Cloud Native Computing Foundation, is widely adopted by major cloud and SaaS providers. It details how to encapsulate CloudEvents in various protocols like AMQP, HTTP, Kafka, MQTT, websockets, Protobuf, etc. In our setup, CloudEvents are embedded in HTTP messages, with metadata in HTTP headers and event data in JSON format in the HTTP message body.

To invoke event-driven functions, Knative uses CloudEvents. Knative offer two elements that allow the routing of a CloudEvent from a source of events to the consumers of the events. The first element is the broker that receives the CloudEvents from the source. The second element is the trigger that is bound to a broker and specifies filters that the CloudEvent must satisfy in order to deliver the event to a consumer. This process is shown in Fig. 1. These consumers can be standard Kubernetes resources or Knative services. The latter option offers automatic scalability and concurrency control.

A network layer is necessary for installing Knative, with options like Kourier, Istio, or Contour. We chose Contour, as it recognizes Knative properties governing function

Fig. 1 A source sends a CloudEvent to the broker, then the trigger sends the event to a sink depending on the attributes of the event

timeouts, set in the ConfigMap config-defaults in the knative-serving namespace. We adjusted these default values for longer function execution as shown in Listing 1.

The deployed infrastructure, shown in Fig. 2, consists of Knative deployed on a Kubernetes cluster and various serverless event-driven functions. Video chunks are stored on a download server (Nginx container), and a private registry is used for container images, accessible to Kubernetes through credentials. Processed video chunks

Listing 1 Modified timeout properties used to set timeouts of functions

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-defaults
  namespace: knative-serving
data:
  revision-timeout-seconds: "3600"
  max-revision-timeout-seconds: "7200"
  scale-to-zero-grace-period: "15s"
```

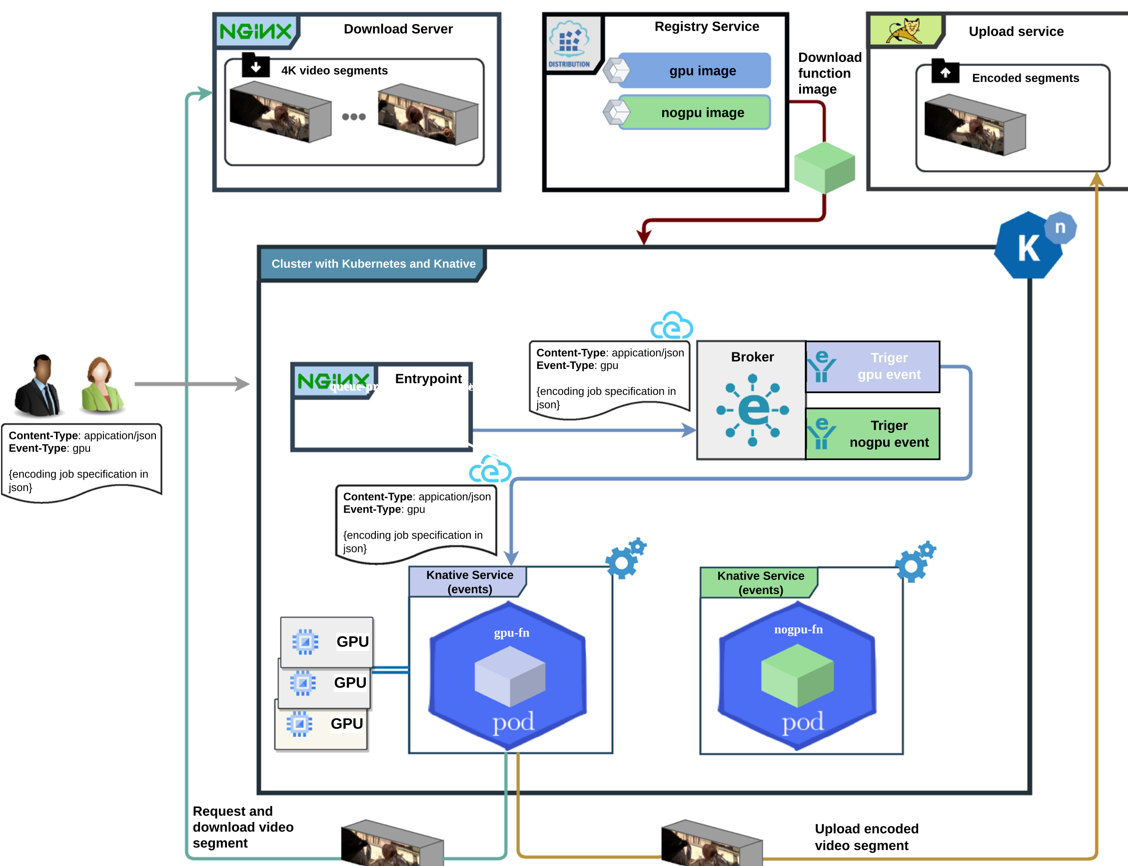


Fig. 2 A cluster with Kubernetes and Knative where functions are deployed. A download server with the videos, an upload server to upload the results, and a container image registry are connected to the cluster. The kubernetes cluster is configured to have access to GPU resources and the function replicas can request GPU acceleration

are uploaded to a server running an upload service (developed in Tomcat). To encode a video, a CloudEvent is generated for each chunk and sent to the network layer gateway, then delivered to a function replica.

3.2 Function implementation

Knative imposes no restrictions on function implementation. We used Java, based on our prior work, to develop the function with frameworks like Tomcat, Quarkus, Spring, or Micronaut, each capable of exposing an HTTP endpoint.

The functions are implemented using a modified version of embedded Tomcat, which offers CloudEvents abstractions [33]. This leads to smaller images due to fewer dependencies. To develop a event-driven function, the `CloudEventListener` abstract class must be extended to implement the abstract method `consumeEvent(...)` that receives the unmarshalled `CloudEvent` object and the two standard objects `HttpServletRequest` and `HttpServletResponse`. Using this application runtime we developed two serverless functions:

- Encoding video with GPU supporting HEVC using Nvidia NVENC codec, and
- Encoding video without GPU supporting libx265 codec.

These functions, being ephemeral, download, process, and upload video chunks for each invocation. The functions are deployed to respond to HTTP requests that encapsulate CloudEvents. For instance, a sample call to activate the functions via the broker through the Envoy gateway is detailed in Listing 2. In this call we can see the parameters passed to the `ffmpeg` process and the information about the data being downloaded and uploaded.

4 Experiments and results

This section describes the testbed used for the experiments and the different scenarios considered: three to be used with the codecs that do no utilise the GPU and other three with the codecs that do use the GPU. Then we analyse the impact of the requested replica resources in the cold start time. The resource consumption in terms of RAM, CPU and GPU usage is measured in all scenarios and codecs. Finally, the behaviour when encoding multiple representations in a single call is analysed.

4.1 Testbed and videos

The testbed used for the experiment consists of five physical machines interconnected via a 10 Gb/s switch. Three of them are commodity servers: one of the servers acts as a private registry that stores all the function images; other server has a Nginx service that allows to download all the video segments via HTTP; the third one is an upload service over HTTP, designed to store the encoded videos.

The Kubernetes cluster is deployed on the other two servers. One of them allocates the Kubernetes master node in a virtual machine with 16 GB RAM and 10 vCPUs. The other server has 128 GB RAM, 12 cores and 4 Nvidia Quadro M4000 GPUs, with 8 GB RAM each, that will allocate the Kubernetes worker VMs.

The virtual infrastructure has been designed carefully taking into account the performance. The SR-IOV NICs have been configured to expose virtual functions that are attached to the virtual machines. Besides, the GPUs will be assigned to the VMs using PCI passthrough. Finally, each VM will have a dedicated SSD to ensure fast data access. For Kubernetes, Docker was selected as the Container Runtime Interface (CRI) and Flannel as the Container Network Interface (CNI). Besides, we have deployed the Nvidia Operator (version v23.3.1) which facilitates a

Listing 2

```
curl -s -X POST KNATIVE_GATEWAY_URL \
    -H "Ce-createdtime: 1699607955259" \
    -H 'Content-Type: application/json' \
    -H 'Ce-Type: encoder' \
    -H 'Ce-Specversion: 1.0' \
    -H 'Ce-Source: /HttpEventSource' \
    -H "Ce-Id: 1" \
    -d "{
      "ffmpegParams": "-i chunk1.mp4 -c:v libx265 -b:v 11M
        -maxrate 11M -minrate 11M -preset medium echunk1.mp4",
      "datamesh": {
        "downloadUrl": "DOWNLOAD_SERVER_URL",
        "downloadFiles": [chunk1.mp4],
        "uploadFiles": [echunk1.mp4],
        "uploadUrl": "UPLOAD_SERVER_URL",
        "timesFile": "times.json"
    }
    }"
```

Table 1 Videos used to perform the experiments

Video name	Acronym	Resolution	Duration (seconds)	Chunks	Size (GB)
Ancient thought	ANC	4K	188	94	2.9
Eldorado	ELD	4K	184	92	4.7
Indoor soccer	IND	4K	252	126	8.0
Skateboarding	SKA	4K	276	138	7.7

dynamic reconfiguration of the assignation of GPUs to VMs. The serverless platform used in this work is Knative (version 1.4.0).

The experimental phase employs two specific images. The `ffmpeg-fn-acc` image, with 604 MB of size, is built with Java, FFmpeg, CUDA GPU support, and incorporates all necessary Nvidia drivers. This image supports GPU acceleration and the codec `hevc_nvenc`. In contrast, the `ffmpeg-fn-no-acc` image, with 120 MB of size, is built with Java and FFmpeg, does not have GPU support but can handle codec `libx265`.

The 4K videos[1] used in the experiments are listed in Table 1. The videos have been downloaded, transcoded using x264 at very high quality (10 CRF), and finally split to obtain segments of 2 s. Table 1 also shows the final sizes after that process of transcoding and splitting.

4.2 Scenarios without and with GPUs

We have tested the performance in six different scenarios. In three of them, the replicas do not require GPU and use `libx265` to encode the segments. In the other three, the replicas request GPU and the encoder used is `hevc_nvenc`. Both encoders are different implementations of the same standard H.265/HEVC.[2] Meanwhile, `hevc_nvenc` leverage GPU integration `libx265` only use CPUs

In all the scenarios, the computational resources assigned to each experiment are the same: 8 vCPUs and 96 GiB RAM. When GPUs are considered, the total number of GPUs assigned will be the same. However, these computational resources will be distributed differently in each scenario. The goal is to determine what scenario is the best (given these computational resources) in terms of video encoding time. The constraints encountered stem from a deficiency in resources, attributed to the scarcity of financial allocations.

4.2.1 Scenarios without GPU

Figure 3 depicts scenarios where GPU acceleration is not used for video encoding tasks. The three scenarios are:

Scenario 1: four slim VMs with four slim replicas. This configuration has four slim worker nodes with 2 vCPUs and 24 GiB RAM. On each VM, a slim replica, that requests 2 vCPUs and 24 GiB RAM, has been provisioned, see Fig. 3a. In this scenario, the load is evenly distributed among the lightweight VMs and their corresponding lightweight replicas. Each replica is allowed to run only one concurrent encoding job.

Scenario 2: one fat VM with four slim replicas. This scenario employs a single fat VM with 8 vCPUs and 96 GiB RAM. This single VM allocates four slim replicas that are limited to 2 vCPUs and 24 GiB RAM, see Fig. 3b. Comparing this scenario with the previous one, the total amount of computational resources are the same, but their distribution is different. Each replica is allowed to run only one concurrent encoding job.

Scenario 3: one fat VM with one fat replica. This setup also uses a single fat VM with 8 vCPUs and 96 GiB of memory. However, instead of deploying four slim functions, we have a single fat function deployed on it, see Fig. 3c. The replica is allowed to run up to four concurrent encoding jobs.

4.2.2 Scenarios with GPU

In this subsection, the scenarios described have access to GPUs to speedup the video encoding process. The four GPUs will be assigned in all the scenarios depicted in Fig. 4 and described below.

Scenario 4: four slim VMs with four GPU-assisted slim replicas. This scenario uses four slim VMs, with 2 vCPUs and 24 GiB RAM each. Besides, each VM has a dedicated GPU. The replicas in this scenario will request 2 vCPUs and 24 GiB RAM and will have access to the GPU of the underlying VM, see Fig. 4a. Each replica is allowed to run only one concurrent encoding job.

Scenario 5: one fat VM with four GPU-assisted slim replicas. A single fat VM with 8 vCPUs, 96 GiB of memory, and the four GPUs allocates four replicas of the function. Each replica is limited to use 2 vCPUs, 24 GiB RAM, and one GPU, see Fig. 4b. Each replica is allowed to run only one concurrent encoding job.

[1] 4K videos available at: https://www.cablelabs.com/4k

[2] ITU Recommendation, available at: https://www.itu.int/rec/T-REC-H.265-202309-I/en

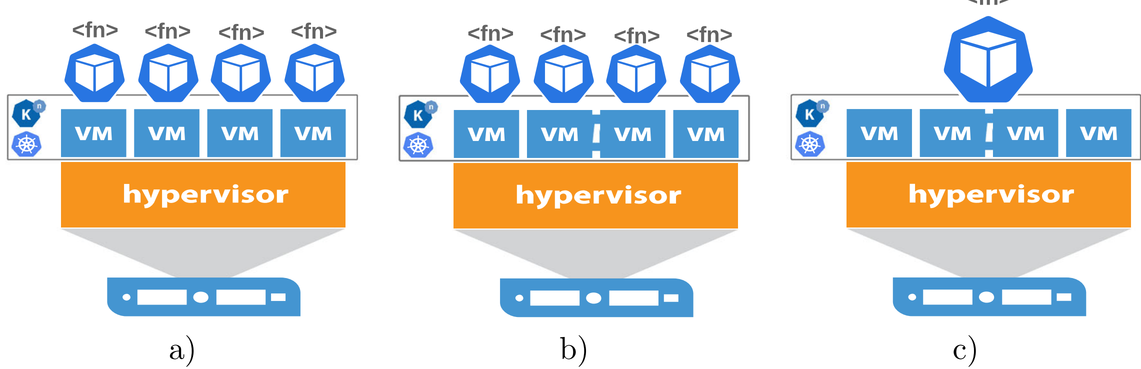


Fig. 3 Scenarios of replicas without GPUs: **a** slim VMs with slim replicas, **b** fat VM with slim replicas, and **c** fat VM with fat replica

Scenario 6: one fat VM with one GPU-pool fat replica.
In this setup a single fat VM, with 8 vCPUs, 96 GiB of memory, and the four GPUs allocate a single replica of the function. The replica requests 8 vCPUs, 96 GiB RAM, and the four GPUs, see Fig. 4c. This replica is allowed to run up to four concurrent encoding jobs.

4.3 Impact of requested function resources on cold start

One of the main features that defines serverless platforms is the ability to scale to zero the number of instances running the function if they are idle for a period of time. This is a convenient feature for users as entails a reduction in costs. However, this convenience has an impact on latency. If there are no running instances and the platform receives a request, one instance must be started to deal with the request. The container runtime must download the layers of the images from the registry if they are not in the file system of the node. Then the environment for the pod has to be created, the function must be started and, finally, the request can be delivered to the function. This elapsed time is named **cold start**. In this work, this time will be measured as the time elapsed from the creation of the event to the time when the event is received by the function.

The experiments described in this section are designed to evaluate the impact of the resources requested by the replica and the underlying VM configuration on the cold-start. We have performed 400 invocations in each scenario ensuring that between each invocation the platform has scaled to zero the number of instances.

Figure 5 shows the boxplots of the data obtained for the six scenarios. Each subfigure shows the boxplot of the cold start times when no GPU is requested (left) and the boxplot of the cold start times when the replicas request GPU (right). The pattern observed is that when the replicas request GPUs the required time to have the function running increases when compared with the setup where no GPU are requested. Specifically, the mean overhead is 10.47% in the first scenario, 17.44% in the second, and 36.50% in the third one. Looking at the results of the scenarios with GPUs, we can conclude that there are no big differences among them (the biggest difference in mean is just a 3.92%). On the contrary, for the scenarios without GPU we can observe more differences (up to 20.15%) being the scenario 3 (with only one fat replica) the fastest one.

4.4 Video encoding times and replica resource consumption

The video used for all the experiments in this section is SKA with 138 2-second segments, 7.7 GB of total size and a mean size per segment of 55.8 MB.

For each scenario, the total time required to encode all the segments was computed. When collecting this data, no monitoring of resources was performed (as the monitoring itself consumes part of the resources).

The experiment was repeated to collect resource consumption data. In this setup, PodInsights[34], a Kubernetes daemonset resource (that ensures that it will run one copy at each worker node) has been deployed. This daemonset makes use of the `docker stats` command to recover information of every container running in the worker node, and stores the information in a MongoDB instance running inside Kubernetes.

After each experiment, the collected data is queried and stored in files for an ulterior analysis. It is important to highlight that the `docker stats` command can only be executed each second. For those scenarios where there are GPUs, the command `nvidia-smi` was executed every 200 milliseconds inside each replica to collect GPU usage.

The goal of this section is twofold. On the one hand, we want to compare the gain when using the GPUs to encode in a serverless platform compared with a baseline case, where GPUs are not available. On the other hand, we want to determine what is the best scenario, given the available resources, in terms of total time required to encode.

Fig. 4 Scenarios with GPUs: **a** slim VMs with slim replicas, **b** fat VM with slim replicas, and **c** fat VM with fat replica

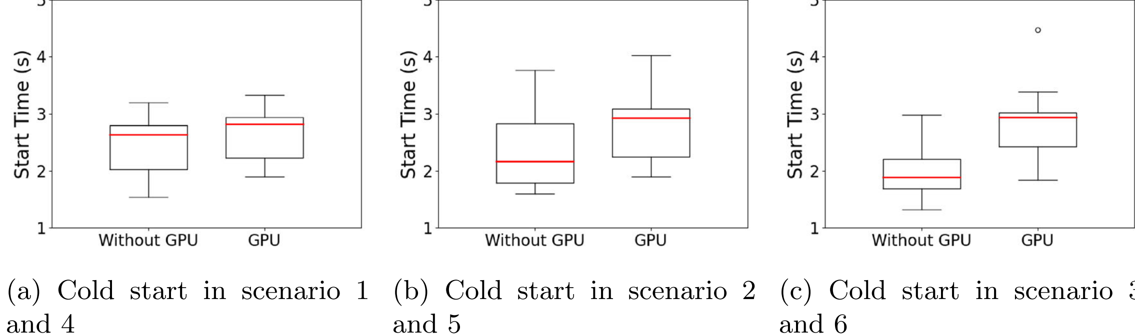


Fig. 5 Cold start times for the scenarios. **a** Scenario 1 and 4, **b** scenario 2 and 5, and **c** scenario 3 and 6

Listing 3 Sample ffmpeg command executed inside the replicas to encode a video chunk using x265.

```
ffmpeg -i chunk1.mp4 -c:v libx265 -b:v 8M -maxrate 8M -minrate 8M
    -preset medium echunk1.mp4
```

4.4.1 Encoding times and resource consumption in the scenarios without GPUs

As a baseline case, we present the data obtained when the encoding is performed in the scenarios that do not utilise GPUs. Table 2 describes the different scenarios studied regarding the resources allocated to the replicas, the number of worker nodes, and the concurrency (number of simultaneous encoding jobs) allowed in the replicas.

Listing 3 presents an illustrative example of the ffmpeg commands executed inside the replicas to encode using `libx265` The output resolution is 4K and the bitrates is 8 Mb/s for `libx265`, which are the minimum recommended bitrates for these codecs.[3]

Table 3 summarises the times obtained for the encoder `libx265` and the three scenarios. Download, encoding, and upload columns show the mean time (in milliseconds) required to download, encode, and upload the 138 segments of the video. The best scenario is scenario 2 (with one fat VM and four replicas of the function) and the worst setup is scenario 1. The encoding time for x265 in the scenario 2 improves the encoding time by 8.42%.

[3] See for instance https://support.google.com/youtube/answer/2853702

Table 2 Different scenarios where the resources allocated to the replicas, the number of worker nodes, the encoder and the concurrency allowed in the replicas are varied

Name	VMs	Instances	CPUs	RAM	Concurrency
$< x265 >^{e_1}_{slim}$	4	4	2	24	1
$< x265 >^{e_2}_{slim}$	1	4	2	24	1
$< x265 >^{e_3}_{fat}$	1	1	8	96	4

The mean upload times are sort for x265 due to the size of the videos are smaller (the bitrate is 8 Mb). The loss in performance in scenario 1 can be attributed to the consumption of resources by the Knative service mesh in each worker node, which is around 480 milicores per node. In this setup, each VM has 2 vCPUs (or 2000 millicores), so the service mesh consumes a 24% of the available resources. Therefore, with 4 worker nodes the service mesh consumes almost 2 vCPUs. In the other two scenarios, with one worker node, the service mesh resource consumption represents only the 6% (480 millicores of the 8000 millicores available).

The total time in seconds required to encode all the segments of the video is shown in the last column of Table 3. This is in line with the known efficiency of `libx265` in achieving greater compression at the cost of more complexity.

Figure 6 shows the job distribution per replica during the encoding process. It is important to highlight that the computational burden of the encoding depends on the contents of the segments, therefore it is possible for a replica to execute some more jobs. However, as a new job is delivered to a replica only when it notifies the completion of the task, all the replicas spend a similar total amount of time encoding (in this experiment the maximum difference among all replicas was only 0.6%). In scenario 3 with a fat VM with one fat function, all encoding tasks are handled by the only available replica, but in this case up to four jobs can be executed concurrently.

Turning the attention to resource consumption, Fig. 7 shows the boxplot of the CPU consumption, obtained from the `percentageCPU` value from `docker stats`, of each replica involved in the encoding process. Scenario 1 with four VMs and four slim replicas each encoding consumed nearly all the CPU resources of the underlying VM. In scenario 3 with one fat VM that allocates one fat replica, the same behaviour is obtained. In scenario 2, with one fat VM and four slim replicas, each replica consumes one fourth of the available CPU resources, therefore the aggregated effect of the four replicas also saturates the CPU resources.

Figure 8 shows the memory consumption of each replica during the encoding process, offering insight of the different memory requirements for replicas. The top row shows the data for scenario 1 (four each slim replicas in its own slim VM) where the maximum memory consumption is 1.7 GB and this maximum is fairly stable among jobs. The central row shows the data for the scenario 2 (four slim replicas in a fat VM) where we can observe peaks at 2.2 GB, but most jobs consume 2.1 GB.

In scenario 2 there is an increase of 0.4 GB of RAM per replica compared to the RAM consumption in scenario 1. In scenario 1 the replica only sees the 2 vCPUs of the VM, but in scenario 2 the replica sees the 8 vCPUs, even though the function is limited to use 2000 millicores. The `ffmpeg` call used starts as many threads as vCPUs are available. Therefore, when one replica runs `ffmpeg` in a slim VM, that has 2 vCPUS, two threads are started. When four replicas running `ffmpeg` are executed in a fat VM, with 8 vCPUS, each `ffmpeg` process launches eight threads. This increase in the number of threads accounts for the increase of 0,4 GB of RAM consumption per replica.

Finally, the bottom row is devoted to scenario 3 (one fat replica in a fat VM with up to four concurrent jobs) the maximum memory is four-fold those of scenario 2. In scenario 3 we observe that, due to the concurrency of tasks, the memory usage does not drop as there is always at least one `ffmpeg` process actively processing a video.

4.4.2 Encoding times and resource consumption in the scenarios with GPUs

This subsection presents the data obtained when the encoding is performed in the three scenarios that utilise GPUs and compares these results with those presented in the previous section. Table 4 describes the different scenarios studied, the resources allocated to the replicas, the number of worker nodes, the number of GPUs used per replica, and the concurrency (number of simultaneous encoding jobs) allowed in the replicas.

Listing 4 presents an illustrative example of the ffmpeg commands executed inside the replicas to encode one video chunk with `hevc_nvenc` using the GPU.

Listing 4 Sample ffmpeg commands executed inside the replicas to encode a video chunk with `hevc_nvenc` using the GPU.

```
ffmpeg -i chunk1.mp4 -c:v hevc_nvenc -b:v 8M -maxrate 8M -minrate 8M
    -rc vbr -preset medium echunk1.mp4
```

Table 5 summarises the results for the three scenarios that entail the use of GPUs. Download, encoding, and upload columns shows the mean time (in milliseconds) required to download, encode, and upload the 138 segments of the video respectively.

We can draw similar conclusions to those obtained in the previous section. The scenario does not have a noticeable impact in the download nor in the upload times, but it has an impact on the encoding times. Similarly to the scenarios without GPUs, the best scenario for both codecs is scenario 5 (with one fat VM and four slim replicas of the function) and the worst setup is scenario 4. The mean encoding time per segment for `hevc_nvenc` in the scenario 5 improves a 15.21%. In contrast with the scenarios that do not require GPU, the implementation that uses GPUs produces similar encoding times.

The total time (in seconds) required to encode all the segments of the video is shown in the last column of Table 5. Comparing the data for scenario 5 with the last column of scenario 2 in Table 3, we can see the benefits of using the GPU. Specifically, `hevc_nvenc` is 8.33 times faster than `libx265`.

As illustrated in Fig. 9, during the encoding process utilising GPUs, the workload is distributed among all the replicas in scenarios 4 and 5. Even though the number of jobs completed at each replica is slightly different, the maximum difference between completion times is less than 1.12% in all cases. The single replica in scenario 6 deals with all the jobs, running up to four encoding jobs concurrently.

Finally, we examine the resource utilization of each GPU replica, considering resources from the VM (CPU and

Table 3 Mean times in milliseconds (per segment) to download, encode and upload

Name	Dowload(ms)	Encoding(ms)	Upload(ms)	Total(s)
$<x265>^{e_1}_{slim}$	175	71938	50	2518.78
$<x265>^{e_2}_{slim}$	186	66351	57	2320.10
$<x265>^{e_3}_{fat}$	160	67251	52	2332.38

Last column shows the total seconds to encode all the chunks of the video

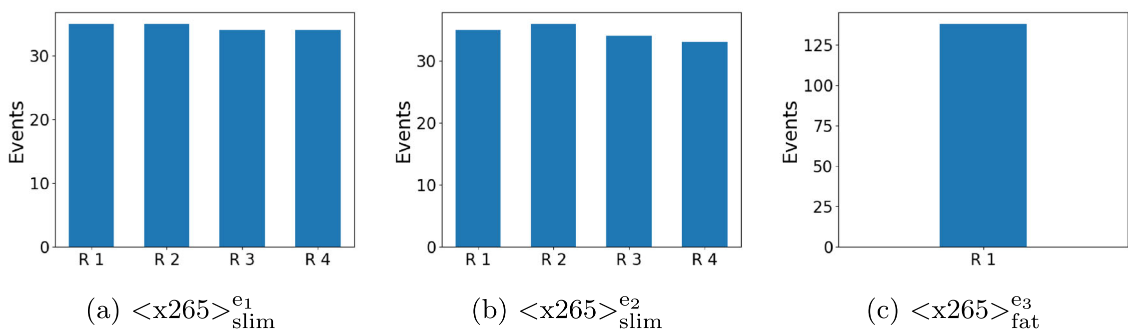


Fig. 6 Number of events (encoding jobs) managed by each replica when encoding the 138 chunks of SKA video using `libx265` in the considered scenarios. **a** Scenario 1 with 4 slim replicas, **b** scenario 2 with 4 slim replicas, and **c** scenario 3 with 1 fat replica

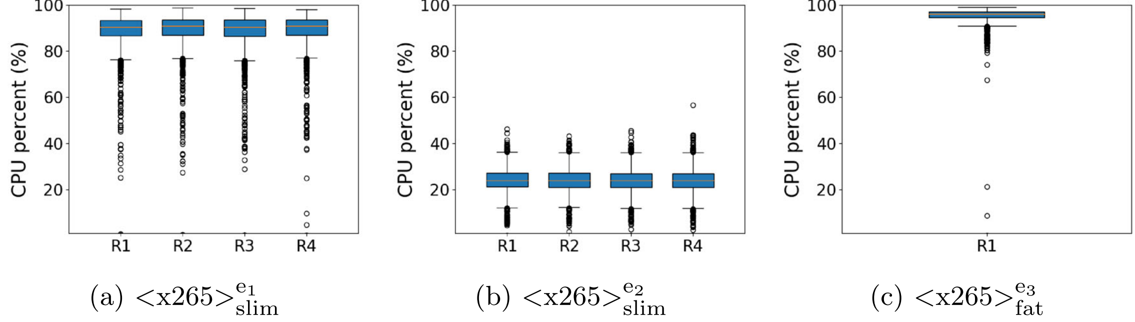


Fig. 7 CPU consumption (in %) per scenario for each replica of the `libx265` encoder function. **a** Scenario 1 with 4 slim replicas, **b** scenario 2 with 4 slim replicas, and **c** scenario 3 with 1 fat replica

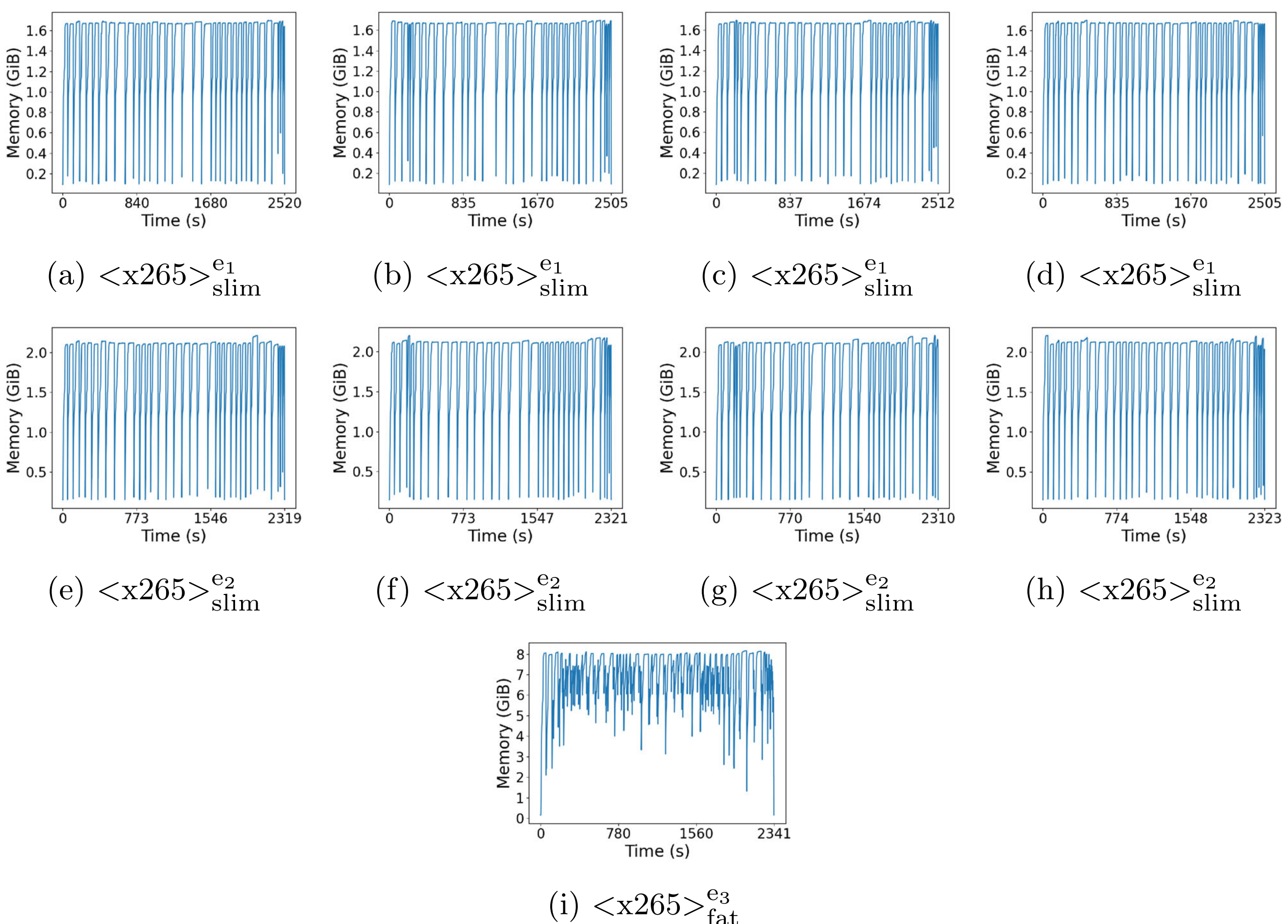


Fig. 8 Memory consumption (in GiB) vs time (in seconds) per replica of the encoder function `libx265`. Top row (**a**, **b**, **c**, and **d**) shows results for scenario 1, central row (**e**, **f**, **g**, and **h**) for scenario 2, and bottom row (**i**) for scenario 3

Table 4 Different scenarios where the resources allocated to the replicas, the number of worker nodes, the number of GPUs per replica and the concurrency allowed in the replicas are varied

Name	VMs	Replicas	GPUs per replica	CPUs	RAM	Concurrency
$<hevc_nvenc>^{e_4}_{slim}$	4	4	1	2	24	1
$<hevc_nvenc>^{e_5}_{slim}$	1	4	1	2	24	1
$<hevc_nvenc>^{e_6}_{fat}$	1	1	4	8	96	4

Table 5 Mean times (per segment) to download, encode and upload

Name	Download(ms)	Encode(ms)	Upload(ms)	Total(s)
$<hevc_nvenc>^{e_4}_{slim}$	153	8771	50	315.77
$<hevc_nvenc>^{e_5}_{slim}$	164	7613	59	278.37
$<hevc_nvenc>^{e_6}_{fat}$	141	7948	52	286.23

Last column shows the total time to encode all the chunks of the video

RAM) and resources from the GPU (GPU utilization and GPU memory usage) during the encoding process. Figures 10, 11 show how `hevc_nvenc` influence CPU and GPU utilization respectively across different scenarios.

Comparing Fig. 7 with Fig. 10 we can observe a reduction in CPU consumption when GPUs are used. As an example, in scenario 4 using `hevc_nvenc` the mean CPU

consumption is around 80% while in scenario 1 using hevc_nvenc the mean CPU consumption is around 85%.

From Fig. 11 we can see that in scenario 4 using hevc_nvenc the mean GPU utilization is 10.5%. The last column in Fig. 11 shows the aggregated GPU utilization of the four GPUs assigned to the fat replica.

The RAM consumption for hevc_nvenc is shown in Fig. 12. Comparing this data with Fig. 8 we observe the following reductions in RAM usage comparing hevc_nvenc with libx265 : 77.7% (scenarios 4 and 1); 71.2% (scenarios 5 and 2); and 76.7% (scenarios 6 and 3).

Finally, Fig. 13 show the GPU memory usage for hevc_nvenc. The hevc_nvenc encoder consumes low memory in the GPU: 0.17 GB for scenario 4, 0.17 GB for scenario 5, and 0.67 GB for scenario 6. Besides, adding the RAM consumption and the memory usage in the GPU, the total memory is lower than the RAM required for the encoder that do not use GPU.

Summarising, there is a noticeable reduction in both CPU and RAM usage when GPUs are employed. This reduction is expected, as the GPU takes on the bulk of the computational load, allowing the machine's other resources to operate with less strain.

4.5 Function performance in multiresolution video encoding

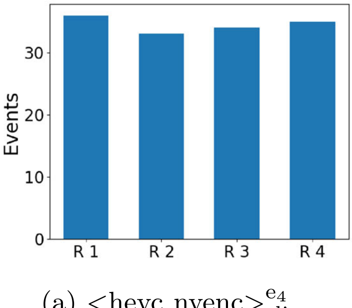
From the data obtained in the previous subsections, scenario 2 (without GPUs) and scenario 5 (with GPUs) provided the best results in terms of coding time. In this section, we will select those scenarios to perform the encoding of all the segments of the four videos detailed in Table 1. Besides, each segment will be encoded in different representations in a single call so that each segment will be transferred to the GPU only once. The output of each function will be five representations with different resolutions and bitrates.

4.5.1 Multiresolution encoding without GPU

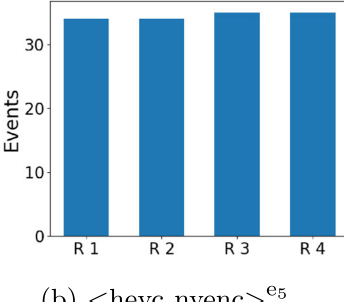
Listing 5 shows a sample command used to encode a segment, in a single ffmpeg call, at five different resolutions.

Listing 5 Sample calls to perform multiresolution encoding: using libx265 .

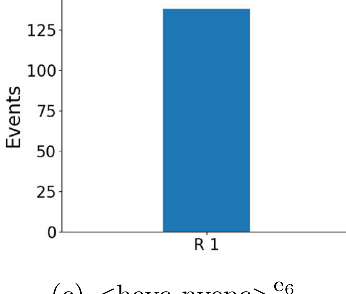
```
#x265 without gpu
ffmpeg -i chunk1.mp4
    -filter_complex '[0:v]yadif,split=5[out1][out2][out3][out4][out5]'
    -map '[out1]' -c:v libx265 -s 854x480 -b:v 1350k -maxrate 1500k
        -bufsize 3M -preset medium echunk1_480p.mp4
    -map '[out2]' -c:v libx265 -s 1280x720 -b:v 2750k -maxrate 4M
        -bufsize 8M -preset medium echunk1_720p.mp4
    -map '[out3]' -c:v libx265 -s 1920x1080 -b:v 6M -maxrate 8M
        -bufsize 16M -preset medium echunk1_1080p.mp4
    -map '[out4]' -c:v libx265 -s 2560x1440 -b:v 8M -maxrate 11M
        -bufsize 22M -preset medium echunk1_1440p.mp4
    -map '[out5]' -c:v libx265 -s 3840x2160 -b:v 11M -maxrate 14M
        -bufsize 28M -preset medium echunk1_2160p.mp4
```



(a) <hevc_nvenc>$^{e_4}_{slim}$

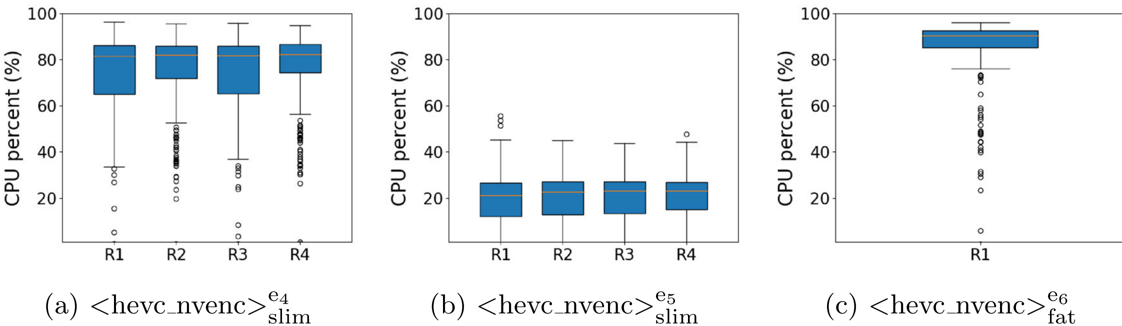


(b) <hevc_nvenc>$^{e_5}_{slim}$



(c) <hevc_nvenc>$^{e_6}_{fat}$

Fig. 9 Number of events (encoding jobs) managed per replica to encode the 138 chunks of SKA video using hevc_nvenc in the three scenarios with GPU. **a** Scenario 4 with 4 slim replicas, **b** scenario 5 with 4 slim replicas, and **c** scenario 6 with 1 fat replica



(a) <hevc_nvenc>$^{e_4}_{slim}$ (b) <hevc_nvenc>$^{e_5}_{slim}$ (c) <hevc_nvenc>$^{e_6}_{fat}$

Fig. 10 CPU consumption (in %) per scenario for each replica of the encoder functions using `hevc_nvenc`. **a** Scenario 4 with 4 slim replicas, **b** scenario 5 with 4 slim replicas, and **c** scenario 6 with 1 fat replica

Detailed times during the encoding of all the segments of the four videos are provided in Table 6. The first column represents the video, the second one the mean time to download a segment of the video, the third one the mean time to encode a video chunk, the fourth is the mean time to upload the encoded video chunk, the fifth is the total size of the video, and finally, the last column shows the total time to encode all the chunks of the video.

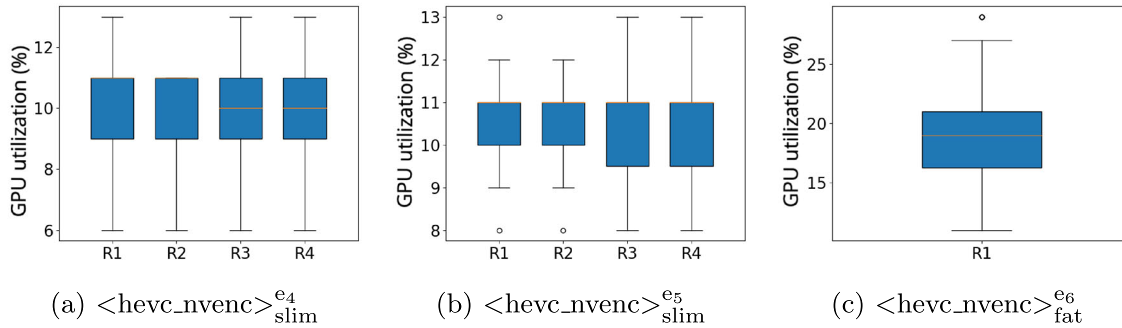
If we compare the mean time to encode each segment of SKA with `libx265` in Table 6 and that of the scenario 2 in Table 3, we see that it takes around 128 s to encode the five representations in a single ffmpeg call while it took around 72 s to encode a single representation. The mean encoding time, when performing multiresolution, with this encoder increments by a 1.78x factor.

4.5.2 Multiresolution encoding with GPU

Listing 6 shows a sample commands executed inside the replicas with GPUs to encode a segment, in a single ffmpeg call, at five different resolutions.

Listing 6 Sample call to perform multiresolution encoding with GPUs using `hevc_nvenc` .

```
#h265/HEVC with GPU
ffmpeg -i chunk1.mp4
    -filter_complex '[0:v]yadif,split=5[out1][out2][out3][out4][out5]'
    -map '[out1]' -c:v hevc_nvenc -s 854x480 -b:v 1350k -maxrate 1500k
        -bufsize 3M -b_ref_mode 0 -rc vbr -preset medium echunk1_480p.mp4
    -map '[out2]' -c:v hevc_nvenc -s 1280x720 -b:v 2750k -maxrate 4M
        -bufsize 8M -b_ref_mode 0 -rc vbr -preset medium echunk1_720p.mp4
    -map '[out3]' -c:v hevc_nvenc -s 1920x1080 -b:v 6M -maxrate 8M
        -bufsize 16M -b_ref_mode 0 -rc vbr -preset medium echunk1_1080p.mp4
    -map '[out4]' -c:v hevc_nvenc -s 2560x1440 -b:v 8M -maxrate 11M
        -bufsize 22M -b_ref_mode 0 -rc vbr -preset medium echunk1_1440p.mp4
    -map '[out5]' -c:v hevc_nvenc -s 3840x2160 -b:v 11M -maxrate 14M
        -bufsize 28M -b_ref_mode 0 -rc vbr -preset medium echunk1_2160p.mp4
```



(a) <hevc_nvenc>$^{e_4}_{slim}$ (b) <hevc_nvenc>$^{e_5}_{slim}$ (c) <hevc_nvenc>$^{e_6}_{fat}$

Fig. 11 GPU utilization (in %) per scenario for each replica of the `hevc_nvenc` encoder function. **a** Scenario 4 with 4 slim replicas, **b** scenario 5 with 4 slim replicas, and **c** scenario 6 with 1 fat replica

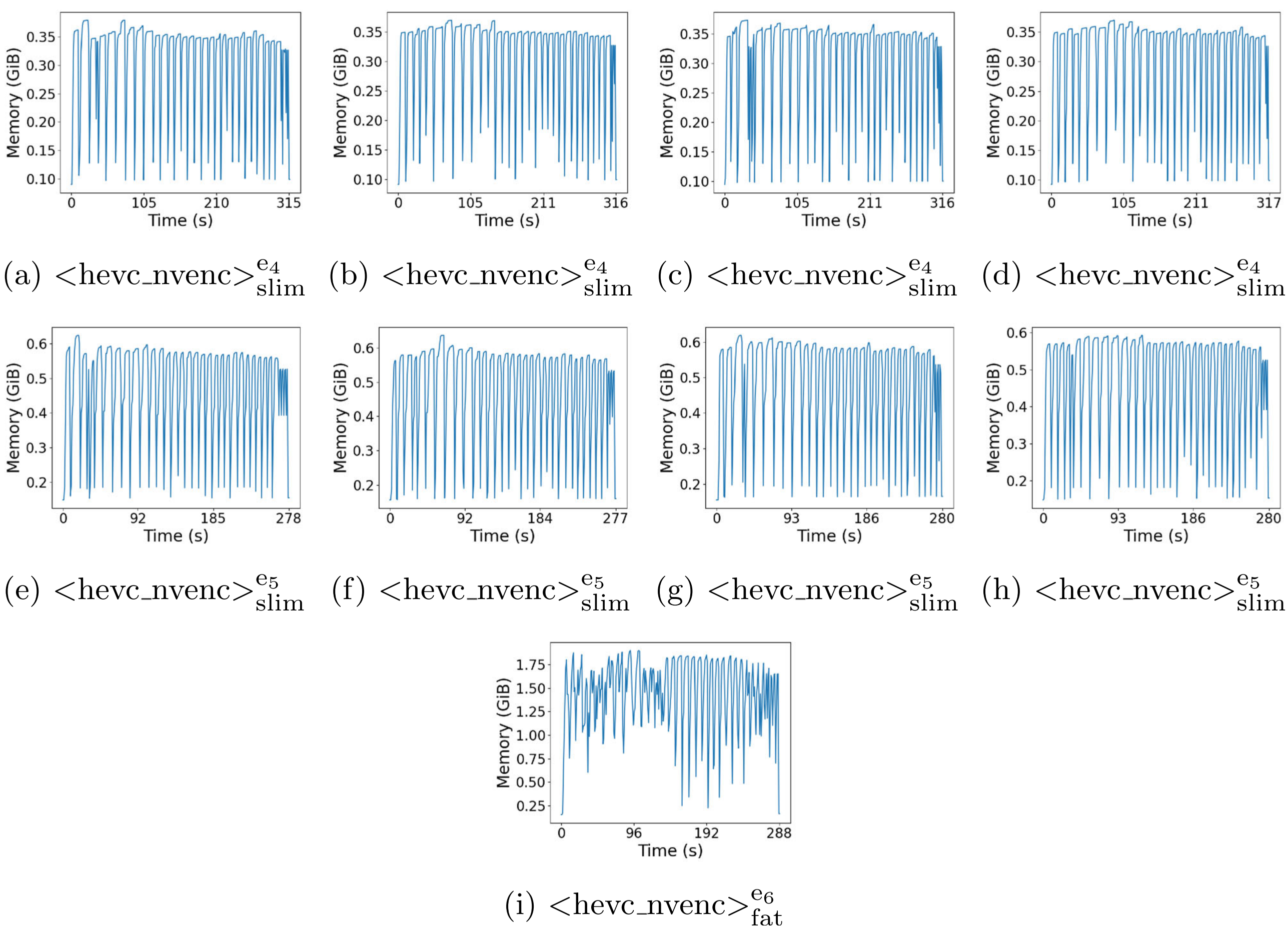


Fig. 12 Memory consumption (in GiB) vs time (in seconds) per replica of the encoder function `hevc_nvenc`. Top row (**a**, **b**, **c**, and **d**) shows results for scenario 4, central row (**e**, **f**, **g**, and **h**) for scenario 5, and bottom row (**i**) for scenario 6

Table 7 presents the times obtained when encoding all the segments of the four videos detailed in Table 1. The first column represents the video, the second one the mean time to download a segment of the video, the third one the mean time to encode a segment, the fourth is the mean time to upload a segment, the fifth is the total size of the video and finally, the last column shows the total time to encode the video.

If we compare the mean time to encode each segment of SKA with `libx265` in Table 6 and that of the scenario 2 in Table 3, we see that it takes around 11.76 s to encode the five representations in a single ffmpeg call while it took around 7.6 s to encode a single representation. The mean encoding time, when performing multiresolution, with this encoder increments by a 1.54x factor.

Comparing the data for `libx265` without GPU and `hevc_nvenc` with GPU in Tables 6, 7, taking the mean across all videos, the mean time to encode a segment using `libx265` without GPU is 141.37 s while the mean encoding time using `hevc_nvenc` with GPU is 11.37 s. Therefore, when encoding with the GPU, the mean time to encode a segment has been reduced drastically by a factor of 12.43.

5 Comparative analysis with related work

In this section, we present a comparative analysis between our proposed architecture and the existing works. As described in Sect. 2, the antecedent studies pertinent to the serverless video encoding paradigm markedly differ from the analysis performed in this work. To perform this comparison we have extracted 11 features from the most close papers and from the proposal as shown in Table 8. The cells marked as N/A denote that we were not able to obtain the information from the respective source papers.

As we have discussed in Sect. 2, we have not found studies that perform a detailed study of the behaviour of encoders in serverless platforms using GPUs. For this reasons, we want to make a comparison with [26, 28], and [29]. Both [26] and [28] leverage Amazon's serverless platform, AWS Lambda, for the execution of their

(a) <hevc_nvenc>$^{e4}_{slim}$

(b) <hevc_nvenc>$^{e4}_{slim}$

(c) <hevc_nvenc>$^{e4}_{slim}$

(d) <hevc_nvenc>$^{e6}_{fat}$

(e) <hevc_nvenc>$^{e5}_{slim}$

(f) <hevc_nvenc>$^{e5}_{slim}$

(g) <hevc_nvenc>$^{e5}_{slim}$

(h) <hevc_nvenc>$^{e5}_{slim}$

(i) <hevc_nvenc>$^{e6}_{fat}$

Fig. 13 GPU memory usage (in MiB) vs time (in seconds) per replica of the encoder function `hevc_nvenc`. Top row (**a**, **b**, **c**, and **d**) shows results for scenario 4, central row (**e**, **f**, **g**, and **h**) for scenario 5, and bottom row (**i**) for scenario 6

Table 6 Mean times (per segment) to download, encode and upload

Video	Download(ms)	Encode(ms)	Upload(ms)	Size(MB)	Total(s)
Results for libx265					
ANC	102	136536	197	496	3264.97
ELD	166	134347	201	545	3139.70
IND	207	166397	221	775	5307.64
SKA	183	128225	190	748	4480.60

Last column shows the total time to encode all the chunks of the video. The two encoders have been evaluated without GPU for each video

Table 7 Mean times (per segment) to download, encode and upload

Video	Download(ms)	Encode(ms)	Upload(ms)	Size(MB)	Total(s)
Results for hevc_nvenc					
ANC	82	10320	177	525	255.11
ELD	146	11165	236	648	272.20
IND	179	12251	232	904	405.71
SKA	158	11757	210	822	424.78

Last column shows the total time to encode all the chunks of the video. The two encoders have been evaluated with GPU for each video

respective frameworks. This choice provides them with a virtually boundless pool of resources. In contrast, both [29] and our proposed solution adopt Knative as the serverless platform deployed on premises. This approach, while constrained in terms of resources, allows us comprehensive control over the system, emphasising a trade-off between resource limitations and enhanced system governance.

Mu framework serves as a fundamental component in the orchestration of parallel executions, a role shared by [26] and [28]. In contrast, both [29] and our proposed solution leverage CloudEvents in conjunction with the Knative open-source serverless platform. It is noteworthy to mention that the application of the Mu framework is now considered quite limited, given its abandonment in 2018 and the subsequent absence of support. This renders our choice of CloudEvents and Knative more contemporary and applicable.

An examination of the video encoding systems presented reveals substantial disparities between the earlier proposals and our system. Notably, the solution proposed in [26] exclusively supports the VP8 codec, which has fallen out of favor in contemporary contexts due to the widespread compatibility offered by the H.264 (`libx264`) and VP9 (an evolution of VP8) codec family. Moreover, VP8 struggles to deliver optimal performance for ultra-high resolutions such as 4K, where alternatives like H.265 (`libx265`) and AV1 showcase superior efficiency.

In contrast, our proposal takes a comprehensive approach by considering two distinct codecs: `libx265`, and `hevc_nvenc`. The encoder `libx265` is preferred for ultra-high resolutions (4K and 8K).

In [29], the authors experimented with the AV1 codec. However, our proposal excluded AV1 due to the absence of GPU support for this codec in the available hardware used for our experiments. Consequently, a direct comparison with CPU-based implementations was not feasible.

Our proposal, along with [26] and [29], underwent testing using 4K videos, while [28] employed videos with resolutions of 1080p and 720p. This divergence in testing resolutions underscores the varying experimental setups and objectives across these studies.

Regarding chunk sizes, our analysis reveals a consistent pattern among [28, 29], and our proposed solution, wherein a 2-second chunking strategy is employed. Conversely, [26] opts for larger 4-second chunks, further subdivided into micro-chunks, each comprising six frames. The selection of chunk sizes, typically ranging between 2 to 4 s, represents a well-balanced compromise between encoding efficiency and adaptability to varying bandwidth conditions [35].

Table 8 Comparison of video encoding systems based on serverless architectures

Features	Fouladi, S. et al. (2017) [26]	Ao, L. et al. (2018) [28]	Moina-Rivera, W. et al. (2022) [29]	Our proposal (2023)
Platform ownership	Amazon	Amazon	Own Platform	Own Platform
Serverless platform	AWS lambda	AWS lambda	Knative Open-source solution	Knative Open-source solution
Framework for managing parallel execution	Mu	Mu	CloudEvents	CloudEvents
Video Codecs Testing	VP8	N/A	x264 and AV1	x265 and HEVC NVENC
Video resolutions tested	4K	1080p and 720p	4K	4K, 1440p, 1080p, 720p, 480p
Other serverless functions	No	Yes, it includes facial recognition	Yes, it calculates VMAF quality metric	No
Video chunk size	4 seconds, divided into 16 microchunks of 6 frames each	2 seconds	2 seconds	2 seconds
Modified video codec	Yes	No, it is based on ffmpeg tool	No, it is based on ffmpeg tool	No, it is based on ffmpeg tool
Resource constrained serverless platform	No	No	Yes	Yes
Scalability and performance study	Low	Medium	High	High
GPU video encoding support	No	No	No	Yes

Some proposals require the adaptation of video codecs to exploit a finer parallelism. In the case of [26], adjustments to the VP8 codec were imperative to ensure the proper functioning of their proposal. In contrast, [28, 29], and our proposal relied on standard implementations of video codecs embedded within the versatile FFmpeg tool. This discrepancy underscores the diverse approaches taken by different frameworks, with each solution tailoring its codec strategy to align with the unique requirements of the serverless architecture.

Another feature presented in Table 8 is the incorporation of functions beyond video coding. In the case of [28], supplementary functions, such as face detection, augment their system's capabilities. Conversely, [29] integrates a video quality function, calculating VMAF per chunk by comparing a reference video with an encoded counterpart. In this work, we focussed on video encoding. However, it is worth noting that the architecture is inherently extensible, allowing for the straightforward integration of additional functions if required. This intentional modularity ensures adaptability to diverse use cases and evolving requirements without compromising the core video encoding functionality.

Notably, [29] and our proposed solution are tailored for environments with constrained resources, presenting different scenarios and analysing the behaviour in these limited environments. Conversely, other proposals such as [26] and [28] were conceptualized within environments where the primary constraint is monetary cost rather than computational resources.

Furthermore, our proposal offers an in-depth exploration of system scalability and performance, specifically when GPUs are employed for the encoding process comparing the results with equivalent encoders that do not use the GPUs. While [29] conducted a similar study, however it did not account for GPU usage. In [28], though a performance study is conducted, the level of depth is not as extensive. On the other hand, [26] presents a more lightweight study regarding the performance and scalability of their proposal, lacking the depth found in subsequent works.

This comparative analysis illuminates the progressive evolution in the depth and breadth of performance studies across these works, with our proposal placing particular emphasis on performance and the impact of GPU utilization.

6 Conclusions and future work

Multimedia streaming has become an indispensable part of modern life, transforming how we consume entertainment, access information, and engage with the world around us. The rise of cloud computing has played a pivotal role in this evolution, providing the infrastructure and scalability necessary to deliver high-quality streaming experiences to a global audience. In particular, serverless architectures are well-suited for video coding systems in HAS applications, where video is divided into segments of 2 or 4 s typically, due to their dynamic resource allocation, parallel execution capabilities, and automatic scaling features. This approach can significantly reduce encoding time and improve overall streaming efficiency.

This study has developed and benchmarked two distinct event-driven serverless functions tailored for video encoding in the cloud edge, evaluating their performance both with and without GPU acceleration. These functions have been encapsulated within two containers, and their operational characteristics were analysed through deployment on an on-premises resource-constrained serverless platform managed by Knative. Our experiments focused on assessing resource consumption under different configurations. The investigation involved varying the allocated resources, considering both fat and slim function replicas, deployed on both slim and fat VMs. This approach allowed for a thorough examination of the serverless functions' performance characteristics, offering insight on their behaviour in real-world deployment scenarios and contributing valuable data to the ongoing development on serverless computing and resource management.

The experiments demonstrate a balanced distribution of encoding tasks across all replicas. For both CPU and GPU codecs, the optimal setup was found to be a large VM hosting four smaller replicas of the function. In contrast, the least efficient configuration involved four small VMs, each running a single replica of the function.The GPU-accelerated version (`hevc_nvenc`) outperforms its CPU-only counterpart (`libx265`) significantly, being 8.33 times faster. This advantage is particularly valuable in live streaming applications, where reducing latency is crucial for achieving optimal performance. The benefits of using `hevc_nvenc` are even more evident in multiresolution encoding scenarios, which are common in HAS, where it reduces the mean time to encode a segment in five resolutions by a factor of 12.43.

As future work, we plan to adapt this platform to perform live multiresolution encoding in the cloud edge for HAS applications. Besides, we will explore new event-driven functions to support the execution of the encoder AV1 in the GPU. Additionally, our ongoing efforts are directed towards the development of a serverless platform designed for the field-of-view (FoV) based encoding of 360VR videos. This initiative aims to leverage serverless architecture to enhance the efficiency and adaptability of video encoding processes specifically targeted at immersive 360-degree virtual reality content. We seek to further

advance the capabilities and versatility of our system, ensuring its relevance and applicability in emerging domains.

Acknowledgements Not applicable.

Author Contributions All authors contributed to the conception and design of the study. The experimental setup was carried out by Raúl Peña-Ortiz, Juan Gutiérrez-Aguado, and Andoni Salcedo-Navarro. Experimental design was performed by all authors. Software implementation was performed by Andoni Salcedo-Navarro. Data collection and analysis were performed by Andoni Salcedo-Navarro and Juan Gutiérrez-Aguado. The first draft of the manuscript was written by all authors, and all authors read and approved the final manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. Grant by PID2021-126209OB-I00 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe", by the "European Union".

Data availability Data sharing not applicable to this article as no datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare that they have no Conflict of interest.

Ethics approval and consent to participate Not applicable.

Consent for publication Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Vijay, S., Mann, P., Chaudhary, R., Rana, A.: Emerging Trends in Multimedia. In: Emerging Technologies in Data Mining and Information Security. Lecture Notes in Networks and Systems, vol. 491, pp. 301–311. Springer, Singapore (2023). https://doi.org/10.1007/978-981-19-4193-1_29
2. Seufert, M., Egger, S., Slanina, M., Zinner, T., Hoßfeld, T., Tran-Gia, P.: A survey on quality of experience of HTTP adaptive streaming. IEEE Commun. Surv. Tutor. **17**(1), 469–492 (2015). https://doi.org/10.1109/COMST.2014.2360940
3. Amini Salehi, M., Li, X.: Future of Multimedia Streaming and Cloud Technology, pp. 179–187. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88451-2_10
4. Zhu, W., Luo, C., Wang, J., Li, S.: Multimedia cloud computing. IEEE Sig. Process. Mag. **28**(3), 59–69 (2011). https://doi.org/10.1109/MSP.2011.940269
5. Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N.J., Popa, R.A., Gonzalez, J.E., Stoica, I., Patterson, D.A.: What serverless computing is and should become: the next phase of cloud computing. Commun. ACM **64**(5), 76–84 (2021). https://doi.org/10.1145/3406011
6. Taibi, D., Spillner, J., Wawruch, K.: Serverless computing-where are we now, and where are we heading? IEEE Softw. **38**(1), 25–31 (2021). https://doi.org/10.1109/MS.2020.3028708
7. Katsigiannis, S., Dimitsas, V., Maroulis, D.: A GPU vs CPU performance evaluation of an experimental video compression algorithm. In: Seventh International Workshop on Quality of Multimedia Experience. QoMEX'15, pp. 1– 6 (2015). https://doi.org/10.1109/QoMEX.2015.7148134
8. The Knative Authors: Knative (2022). https://knative.dev/. Accessed 1 Mar 2024
9. Moina-Rivera, W., Garcia-Pineda, M., Gutiérrez-Aguado, J., Alcaraz-Calero, J.M., Cloud media video encoding: review and challenges: Cloud media video encoding: review and challenges. Multimed. Tools Appl. (2024). https://doi.org/10.1007/s11042-024-18763-2
10. Gutiérrez-Aguado, J., Peña-Ortiz, R., García-Pineda, M., Claver, J.M.: Cloud-based elastic architecture for distributed video encoding: Evaluating H.265, VP9, and AV1. J. Netw. Comput. Appl. **171**, 102782 (2020). https://doi.org/10.1016/j.jnca.2020.102782
11. Yang, C.-T., Wang, H.-Y., Ou, W.-S., Liu, Y.-T., Hsu, C.-H.: On implementation of gpu virtualization using pci pass-through. In: 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings, pp. 711– 716 (2012). https://doi.org/10.1109/CloudCom.2012.6427531
12. Gutiérrez-Aguado, J., Claver, J.M., Peña-Ortiz, R.: Toward a transparent and efficient GPU cloudification architecture. J. Supercomput. **75**(7), 3640–3672 (2019). https://doi.org/10.1007/s11227-018-2720-z
13. Rodríguez-Sánchez, R., Martínez, J.L., Fernández-Escribano, G., Sánchez, J.L., Claver, J.M.: A fast GPU-based motion estimation algorithm for H.264/AVC. In: Schoeffmann, K., Merialdo, B., Hauptmann, A.G., Ngo, C.-W., Andreopoulos, Y., Breiteneder, C. (eds.) Adv. Multimed. Model., pp. 551–562. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27355-1_51
14. Xiao, W., Li, B., Xu, J., Shi, G., Wu, F.: HEVC encoding optimization using multicore CPUs and GPUs. IEEE Trans. Circuits Syst. Video Technol. **25**(11), 1830–1843 (2015). https://doi.org/10.1109/TCSVT.2015.2406199
15. Srinivasan, M.: Vp9 encoder and decoders for next generation online video platforms and services. In: SMPTE 2016 Annual Technical Conference and Exhibition, pp. 1– 14 (2016). https://doi.org/10.5594/M001734
16. Comi, P., Crosta, P.S., Beccari, M., Paglierani, P., Grossi, G., Pedersini, F., Petrini, A.: Hardware-accelerated high-resolution video coding in virtual network functions. In: 2016 European Conference on Networks and Communications (EuCNC), pp. 32–36 (2016). https://doi.org/10.1109/EuCNC.2016.7560999
17. Sharma, P., Chaufournier, L., Shenoy, P., Tay, Y.C.: Containers and virtual machines at scale: a comparative study. In: 17th International Middleware Conference. Middleware'16. ACM, Trento Italy (2016). https://doi.org/10.1145/2988336.2988337
18. Marathe, N., Gandhi, A., Shah, J.M.: Docker Swarm and Kubernetes in cloud computing environment. In: 3rd International Conference on Trends in Electronics and Informatics. ICOEI'19, pp. 179– 184 (2019). https://doi.org/10.1109/ICOEI.2019.8862654
19. Jangda, A., Pinckney, D., Brun, Y., Guha, A.: Formal foundations of serverless computing. Proc. ACM Program. Lang. **3**, 1–26 (2019). https://doi.org/10.1145/3360575

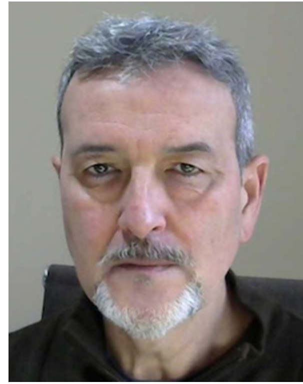
20. Martins, H., Araujo, F., Cunha, P.R.: Benchmarking serverless computing platforms. J. Grid Comput. **18**(4), 691–709 (2020). https://doi.org/10.1007/s10723-020-09523-1
21. OpenFaaS Ltd.: OpenFaas: serverkess functions, made simple. https://www.openfaas.com (2023). Accessed 1 Mar 2024
22. The Knative Authors: Knative. https://knative.dev (2022). Accessed 1 Mar 2024
23. The Apache Software Foundation: Apache OpenWhisk: Open Source Serverless Cloud Platform. https://openwhisk.apache.org (2023). Accessed 1 Mar 2024
24. Iron.io: IronFunctions: Open Source Serverless Computing. https://open.iron.io (2016). Accessed 1 Mar 2024
25. Risco, S., Moltó, G., Naranjo, D.M., Blanquer, I.: Serverless workflows for containerised applications in the cloud continuum. J. Grid Comput. **19**(3), 30 (2021). https://doi.org/10.1007/s10723-021-09570-2
26. Fouladi, S., Wahby, R.S., Shacklett, B., Balasubramaniam, K.V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., Winstein, K.: Encoding, fast and slow: low-latency video processing using thousands of tiny threads. In: 14th USENIX Symposium on Networked Systems Design and Implementation. NSDI'17, pp. 363–376. USENIX Association, Boston (2017)
27. Wang, L., Li, M., Zhang, Y., Ristenpart, T., Swift, M.: Peeking behind the curtains of serverless platforms. In: USENIX Annual Technical Conference. ATC'18, pp. 133–146. USENIX Association, Boston (2018)
28. Ao, L., Izhikevich, L., Voelker, G.M., Porter, G.: Sprocket: a serverless video processing framework. In: Proceedings of the ACM Symposium on Cloud Computing. SoCC'18, pp. 263– 274. ACM, Carlsbad (2018). https://doi.org/10.1145/3267809.3267815
29. Moina-Rivera, W., Garcia-Pineda, M., Claver, J.M., Gutiérrez-Aguado, J.: Event-driven serverless pipelines for video coding and quality metrics. J. Grid Comput. (2023). https://doi.org/10.1007/s10723-023-09647-0
30. Wen, J., Chen, Z., Jin, X., Liu, X.: Rise of the planet of serverless computing: a systematic review. ACM Trans. Softw. Eng. Methodol. **32**(5), 1–61 (2023). https://doi.org/10.1145/3579643
31. Li, J., Kulkarni, S.G., Ramakrishnan, K.K., Li, D.: Analyzing open-source serverless platforms: characteristics and performance. In: The Thirty Third International Conferences on Software Engineering and Knowledge Engineering. SEKE'21. KSI Research Inc., Pittsburgh (2021). https://doi.org/10.18293/seke2021-129
32. CloudEvents Authors: Cloudevents project. https://cloudevents.io/ (2023). Accessed 1 Mar 2024
33. Gutiérrez-Aguado, J.: Adapting embeded Tomcat to develop event-driven serverless functions. In: JCIS2022, SISTEDES. http://hdl.handle.net/11705/JCIS/2022/040 (2022)
34. Salcedo-Navarro, A., Gutiérrez-Aguado, J., Garcia-Pineda, M.: Podinsights: a millisecond pod metric collector for kubernetes. In: Euro American Conference on Telematics and Information Systems - Proceedings of the 12th Euro American Conference on Telematics and Information Systems, EATIS 2024
35. Lederer, S.: Bitmovin: Optimal Adaptive Streaming Formats MPEG-DASH & HLS Segment Length. https://bitmovin.com/mpeg-dash-hls-segment-length/ (2020). Accessed 1 Mar 2024

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Andoni Salcedo-Navarro received the B.Sc. degree in Computer Engineering with a specialization in Computing from the University of Zaragoza and the M.Sc. degree in Web Technologies, Cloud Computing, and Mobile Applications (TWCAM) from the University of Valencia. He is currently in his second year of doctoral studies, with a research focus on multimedia content distribution in cloud-edge environments. His research interests include cloud-based video streaming, and he has published several papers on this topic.

Raúl Peña-Ortiz received his Ph.D. in Computer Science from the Universitat Politècnica de València, Spain. He has more than 20 years experience in software engineering and quality, especially in the application of scientific research knowledge to fix industrial problems related to web-based applications and their ideal deployments, publishing various papers in leading journals and conferences. He worked in several universities, research centers and software companies, participating in more than 60 national and international research & innovation projects. Currently, he is Associate Professor at the Universitat de València, Spain, and his research is aimed at designing and evaluating distributed architectures based on cloud computing.



Jose M. Claver is a Full Professor of Computer Architecture and Technology at the University of Valencia, Master in Physic from the University of Valencia, he has received a Ph.D. in Computer Science from the Technical University of Valencia. He was Associate Professor at the universities of Castilla-La Mancha and Jaume I of Castelló, Spain. He began his research career in the field of High Performance Computing. He has worked on the design of network protocols for real-time applications and tele-operated systems in robotics. Recently, its activity has been focused on the acceleration of video encoding and transcoding, in particular for standard H.264 and H.265, using GPU. Currently, he is working in the field of indoor location using RF signals and in the field of cloud/-fog/dust computing, in order to improve the performance of these infrastructures for different applications. In these subjects, he has

supervised four doctoral theses and has published more than 80 scientific contributions. He is an expert in the field of ICT, a regular reviewer for several international journals, a member of the technical committee of several international conferences and is an IEEE Senior Member.

Miguel Garcia-Pineda received his M.Sc. and Ph.D. degrees in Telecommunication Engineering from the Universitat Politècnica de València in 2008 and 2013, respectively. He currently serves as Vice Dean and Associate Professor in the Department of Computer Science at the School of Engineering (ETSE-UV), Universitat de València. His research interests include multimedia encoding and streaming, Quality of Service and Quality of Experience, and cloud computing. He has involved in several research projects and has authored more than 110 papers in international conference proceedings, as well as over 50 peer-reviewed articles in high-impact international journals. In addition, he is an active associate editor and reviewer for various international journals.

Juan Gutiérrez-Aguado received the Ph.D. degree in Computer Science from the Universitat de València where he is currently an Associate Professor. He has taught undergraduate and graduate courses on image processing, programming, mobile devices, server-side programming, and cloud computing. He is the director of the Master in Web Technologies, Cloud Computing, and Mobile Applications. He has authored or co-authored of journal papers in computer vision, image processing, and recently in cloud monitoring and servlerless platforms for video encoding. His current research focuses on distributed and cloud computing.

Bibliografía

[1] DataReportal, "Digital 2025: Global internet use overview," 2025, consultado: 9 de abril de 2025. [Online]. Available: https://datareportal.com/reports/digital-2025-global-overview-report

[2] AgenciaDigitaLamd, "Estadisticas marketing digital 2025," 2025, consultado: 9 de abril de 2025. [Online]. Available: https://agenciadigitalamd.com/marketing-digital/estadisticas-marketing-digital-2025/

[3] Similarweb, "Top websites ranking - march 2025," 2025, consultado: 9 de abril de 2025. [Online]. Available: https://www.similarweb.com/top-websites

[4] Symphonyai, "Svod subscription video on demand," 2025, consultado: 9 de abril de 2025. [Online]. Available: https://www.symphonyai.com/glossary/media/svod-subscription-video-on-demand/

[5] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia, "A survey on quality of experience of http adaptive streaming," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 469–492, 2014.

[6] W. Moina-Rivera, M. Garcia-Pineda, J. Gutiérrez-Aguado, and J. M. Alcaraz-Calero, "Cloud media video encoding: review and challenges," *Multimedia Tools and Applications*, Mar 2024. [Online]. Available: https://doi.org/10.1007/s11042-024-18763-2

[7] D. Taibi, J. Spillner, and K. Wawruch, "Serverless computing-where are we now, and where are we heading?" *IEEE Software*, vol. 38, no. 1, pp. 25–31, 2021.

[8] C. Cicconetti, M. Conti, A. Passarella, and D. Sabella, "Toward distributed computing environments with serverless solutions in edge systems," *IEEE Communications Magazine*, vol. 58, no. 3, pp. 40–46, 2020. [Online]. Available: http://dx.doi.org/10.1109/mcom.001.1900498

[9] M. Ganguli, S. Ranganath, S. Ravisundar, A. Layek, D. Ilangovan, and E. Verplanke, "Challenges and opportunities in performance benchmarking of service mesh for the edge," in *2021 IEEE International Conference on Edge Computing (EDGE)*, sep 2021.

[10] F. Chiariotti, "A survey on 360-degree video: Coding, quality of experience and streaming," *Computer Communications*, vol. 177, pp. 133–155, 2021.

[11] S. Petrangeli, J. Hooft, T. Wauters, R. Huysegems, P. Alface, T. Bostoen, and F. De Turck, "Live streaming of 4k ultra-high definition video over the internet," in *Proceedings of ACM MMSys*, 2016, p. 4 pages.

[12] C. Ozcinar, A. Smolic, and V. Guillemette, "Tiled streaming for 360º video: A comprehensive survey," *ACM Computing Surveys*, 2019.

[13] V. Chakareski, L. Xie, and W. Zhang, "Viewport-driven rate-distortion optimized 360º video streaming," in *Proceedings of ACM MMSys*, 2019.

[14] J. Gutiérrez-Aguado, R. Peña-Ortiz, M. García-Pineda, and J. M. Claver, "Cloud-based elastic architecture for distributed video encoding: Evaluating h.265, vp9, and av1," *Journal of Network and Computer Applications*, vol. 171, p. 102782, 2020.

[15] C.-T. Yang, H.-Y. Wang, W.-S. Ou, Y.-T. Liu, and C.-H. Hsu, "On implementation of gpu virtualization using pci pass-through," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, 2012, pp. 711–716.

[16] J. Gutiérrez-Aguado, J. M. Claver, and R. Peña-Ortiz, "Toward a transparent and efficient gpu cloudification architecture," *The Journal of Supercomputing*, vol. 75, no. 7, pp. 3640–3672, 2019.

[17] R. Rodríguez-Sánchez, J. L. Martínez, G. Fernández-Escribano, J. L. Sánchez, and J. M. Claver, “A fast gpu-based motion estimation algorithm for h.264/avc,” in *Advances in Multimedia Modeling*, K. Schoeffmann, B. Merialdo, A. G. Hauptmann, C.-W. Ngo, Y. Andreopoulos, and C. Breiteneder, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 551–562.

[18] W. Xiao, B. Li, J. Xu, G. Shi, and F. Wu, “Hevc encoding optimization using multicore cpus and gpus,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 11, pp. 1830–1843, 2015.

[19] M. Srinivasan, “Vp9 encoder and decoders for next generation online video platforms and services,” in *SMPTE 2016 Annual Technical Conference and Exhibition*, 2016, pp. 1–14.

[20] P. Comi, P. S. Crosta, M. Beccari, P. Paglierani, G. Grossi, F. Pedersini, and A. Petrini, “Hardware-accelerated high-resolution video coding in virtual network functions,” in *2016 European Conference on Networks and Communications (EuCNC)*, 2016, pp. 32–36.

[21] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, “Containers and virtual machines at scale: A comparative study,” in *17th International Middleware Conference*, ser. Middleware’16. Trento, Italy: ACM, 11 2016, pp. 1–13.

[22] N. Marathe, A. Gandhi, and J. M. Shah, “Docker swarm and kubernetes in cloud computing environment,” in *3rd International Conference on Trends in Electronics and Informatics*, ser. ICOEI’19, 4 2019, pp. 179–184.

[23] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, “Formal foundations of serverless computing,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–26, 10 2019.

[24] H. Martins, F. Araujo, and P. R. da Cunha, “Benchmarking serverless computing platforms,” *Journal of Grid Computing*, vol. 18, no. 4, pp. 691–709, 7 2020.

[25] OpenFaaS Ltd., “Openfaas: Serverkess functions, made simple,” 2023. [Online]. Available: https://www.openfaas.com

[26] The Knative Authors, "Knative," 2022. [Online]. Available: https://knative.dev/

[27] The Apache Software Foundation, "Apache openwhisk: Open source serverless cloud platform," 2023. [Online]. Available: https://openwhisk.apache.org

[28] Iron.io, "Ironfunctions: Open source serverless computing," 2016. [Online]. Available: https://open.iron.io

[29] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *14th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'17. Boston, MA, USA: USENIX Association, 2017, pp. 363–376. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-fouladi.pdf

[30] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *USENIX Annual Technical Conference*, ser. ATC'18. Boston, MA, USA: USENIX Association, 7 2018, pp. 133–146.

[31] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC'18, Carlsbad, CA, USA, 10 2018, pp. 263–274.

[32] Lift, "Envoy proxy," 2024. [Online]. Available: https://www.envoyproxy.io/

[33] Q. Jiao, B. Xu, and Y. Fan, "Design of cloud native application architecture based on kubernetes," in *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, 2021, pp. 494–499.

[34] B. Lublinsky, E. Jennings, and V. Spišaková, "A kubernetes 'bridge' operator between cloud and external resources," in *2023*

8th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA), 4 2023.

[35] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan, "Dissecting overheads of service mesh sidecars," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC '23, New York, NY, USA, 2023, pp. 142–157.

[36] A. Duque, C. Klein, J. Feng, X. Cai, B. Skubic, and E. Elmroth, "A qualitative evaluation of service mesh-based traffic management for mobile edge cloud," in *2022 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 210–219.

[37] I. E. Richardson, *The H.264 Advanced Video Compression Standard*. John Wiley & Sons, 2003. [Online]. Available: https://www.wiley.com/en-us/The+H+264+Advanced+Video+Compression+Standard%2C+2nd+Edition-p-9780470516928

[38] F. Engineering, "Av1 beats x264 and libvpx-vp9 in practical use case," 2018. [Online]. Available: https://engineering.fb.com/2018/11/07/video-engineering/av1-beats-x264-libvpx-vp9/

[39] A. for Open Media, "A technical overview of AV1," https://aomedia.org/, 2019.

[40] C. Herglotz, H. Och, A. Meyer, G. Ramasubbu, L. Eichermüller, M. Kränzler, F. Brand, K. Fischer, D. T. Nguyen, A. Regensky, and A. Kaup, "The bjøntegaard bible why your way of comparing video codecs may be wrong," *IEEE Transactions on Image Processing*, vol. 33, pp. 987–1001, 2024.

[41] C. Herglotz, M. Kränzler, R. Mons, and A. Kaup, "Beyond bjøntegaard: Limits of video compression performance comparisons," in *2022 IEEE International Conference on Image Processing (ICIP)*, 2022, pp. 46–50.

[42] O. Nawaz, M. Fiedler, and S. Khatibi, "Qoe-based performance comparison of AVC, HEVC, and VP9 on mobile devices with additional influencing factors," *Electronics*, vol. 13, no. 2,

p. 329, 2024. [Online]. Available: https://www.mdpi.com/2079-9292/13/2/329

[43] T. Amestoy, N. Sidaty, W. Hamidouche, and P. Philippe, "Video quality assessment and coding complexity of the versatile video coding standard," *arXiv preprint arXiv:2310.13093*, 2023. [Online]. Available: https://arxiv.org/abs/2310.13093

[44] M. Uhrina, L. Sevcik, J. Bienik, and L. Smatanova, "Performance comparison of VVC, AV1, HEVC, and AVC for high resolutions," *Electronics*, vol. 13, no. 5, p. 953, 2024. [Online]. Available: https://www.mdpi.com/2079-9292/13/5/953

[45] N. Barman, M. Martini, and Y. Reznik, "Bjøntegaard delta (BD): A tutorial overview of the metric, evolution, challenges, and recommendations," *arXiv preprint arXiv:2401.04039*, 2024. [Online]. Available: https://arxiv.org/abs/2401.04039

[46] B. Su, B. Cheng, and J.-L. Chen, "GPU based high definition parallel video codec optimization in mobile device," *IEEE Transactions on Mobile Computing*, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9669133/

[47] A. Salcedo-Navarro, R. Peña-Ortiz, J. M. Claver, M. Garcia-Pineda, and J. Gutiérrez-Aguado, *Cloud-Native GPU-Enabled Architecture for Parallel Video Encoding*. Springer Nature Switzerland, 2024, pp. 327–341.

[48] J. C. Shervin Zare, Ramin Hejazi Moghadam and N. Venkatasubramanian, "Adaptive 360 vr video streaming: Divide and conquer!" in *Proceedings of the 2016 ACM on Multimedia Conference (MM '16)*, 2016, pp. 407–418.

[49] Y. Mao, L. Sun, Y. Liu, and Y. Wang, "Low-latency fov-adaptive coding and streaming for interactive 360° video streaming," in *Proceedings of the 28th ACM International Conference on Multimedia (MM '20)*, 2020, pp. 3696–3704.

[50] Q. Zeng, Z. Yin, Y. Yu, and Y. Zhuang, "A new architecture of 8k vr fov video end-to-end technology," in *Proceedings of the 2022 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, 2022, pp. 148–153.

[51] G. Rigazzi, J.-P. Kainulainen, C. Turyagyenda, A. Mourad, and J. Ahn, "An edge and fog computing platform for effective deployment of 360 video applications," in *Proceedings of the 2019 Cloud Technologies and Energy Efficiency in Mobile Communication Networks (CLEEN)*, 2019, pp. 1–6.

[52] C. Ozcinar, A. D. Abreu, and A. Smolic, "Viewport-aware adaptive 360° video streaming using tiles for virtual reality," in *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, 2017, pp. 2174–2178.

[53] F. Micó-Enguídanos, W. Moina-Rivera, J. Gutiérrez-Aguado, and M. Garcia-Pineda, "Per-title and per-segment crf estimation using dnns for quality-based video coding," *Expert Systems with Applications*, vol. 227, p. 120289, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417423007911

[54] W. Moina-Rivera, J. Gutiérrez-Aguado, M. Garcia-Pineda, and J. M. Claver, "Improving dash encoding with scenes and downscaling techniques for vod streaming," in *2021 IEEE Global Communications Conference (GLOBECOM)*, 2021, pp. 1–6.

[55] W. Moina-Rivera, J. Gutiérrez-Aguado, and M. Garcia-Pineda, "Multi-resolution quality-based video coding system for dash scenarios," in *NOSSDAV '21: Proceedings of the 31st ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, ser. NOSSDAV '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 42–49. [Online]. Available: https://doi.org/10.1145/3458306.3460996

[56] W. Moina-Rivera, M. Garcia-Pineda, J. M. Claver, and J. Gutiérrez-Aguado, "Event-driven serverless pipelines for video coding and quality metrics," *Journal of Grid Computing*, vol. 21, no. 2, p. 20, 2023, published on 2023/03/21. [Online]. Available: https://doi.org/10.1007/s10723-023-09647-0

[57] W. Moina-Rivera, J. Gutiérrez-Aguado, and M. Garcia-Pineda, "Video quality metrics toolkit: An open source software to assess video quality," *SoftwareX*, vol. 23, p. 101427, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711023001231

[58] B. Burns and D. Oppenheimer, “Design patterns for container-based distributed systems,” in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, jun 2016, pp. 108–113. [Online]. Available: https://dl.acm.org/doi/10.5555/3027041.3027059

[59] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow.* Springer International Publishing, 2017, pp. 195–216.

[60] Y. Chen, D. Mukherjee, J. Han, A. Grange, Y. Xu, S. Parker, C. Chen, H. Su, U. Joshi, C.-H. Chiang, and et al., “An overview of coding tools in AV1: the first video codec from the alliance for open media,” *APSIPA Transactions on Signal and Information Processing*, vol. 9, p. e6, 2020.

[61] A. Mercat, M. Viitanen, and J. Vanne, “UVG dataset: 50/120fps 4k sequences for video codec analysis and development,” in *Proceedings of the 11th ACM Multimedia Systems Conference*. ACM, June 2020.

[62] F. Micó-Enguídanos, A. Salcedo-Navarro, M. Garcia-Pineda, and J. Gutiérrez-Aguado, “Features and quality metrics datasets for video coding in dash,” *Scientific Data*, vol. 11, p. 696, 2024. [Online]. Available: https://doi.org/10.1038/s41597-024-03507-6