CloudSkin

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

CloudSkin

(grant agreement No 101092646)

Adaptive virtualization for AI-enabled Cloud-edge Continuum

D5.4 Proof of Concept in different Data Domains

Due date of deliverable: 31-12-2025
Actual submission date: 29-12-2025

Start date of project: 01-01-2023 Duration: 36 months

Summary of the document

Document Type	Report
Dissemination level	Public
State	v1.0
Number of pages	69
WP/Task related to this document	WP5 / T5.4, T5.5, T5.6, T5.7
WP/Task responsible	BSC, URV, NCT, ALT
Leader	Peini Liu (BSC)
Technical Manager	Peini Liu (BSC)
Quality Manager	Ardhi Putra Pratama Hartono (TUD)
Author(s)	Peini Liu (BSC), Joan Oliveras Torra (BSC), Marc Palacín (BSC), Jordi Guitart (BSC), Josep Lluis Berral (BSC), Ramon Nou (BSC), Pol Garcia (BSC), Jordi Torres (BSC), Maria A. Serrano (NBC), Michail Dalgitsis (NBC), Javier Santaella Sánchez (CNX), Ardhi Putra Pratama Hartono (TUD), Jose Miguel Garcia (ALT), Raúl Gracia (DELL), Hossam Elghamry (DELL), Alan Cueva (DELL), Reuben Docea (NCT), Marc Sànchez Artigas (URV), Josep Calero Santo (URV), Carlos Segarra (IMP)
Partner(s) Contributing	BSC, NBC, CNX, NCT, DELL, URV, IMP, ALT
Document ID	CloudSkin_D5.4_Public.pdf
Abstract	Report on the results and experiments from all use cases in the WP, and will report on the experimentation and tests for methods integration on the different use cases. All generated data concerning the workloads and performance of systems and analytics will also be included in the report, with special interest in provide knowledge to other communities related with the use cases.
Keywords	Cloud-Edge Continuum, Use cases, Experiments.

History of changes

Version	Date	Author	Summary of changes
0.1	01-12-2025	Peini Liu	First draft.
0.2	02-12-2025	Peini Liu	Add Mobility use case.
0.3	03-12-2025	Javier Santaella Sánchez	Modify Mobility use case story and hardware.
0.4	03-12-2025	Maria A. Serrano	Update Mobility use case platform.
0.5	03-12-2025	Marc Palacín, Michail Dalgitsis	Add Mobility use case demo.
0.6	03-12-2025	Ardhi Putra Pratama Hartono	Confidential computing in use cases.
0.7	09-12-2025	Raúl Gracia	Add Surgery use cases.
0.8	12-12-2025	Jose Miguel Garcia	Add Agriculture use cases.
0.9	15-12-2025	Reuben Docea	Add Surgery use cases.
1.0	16-12-2025	Peini Liu, Joan Oliveras Torra, Michail Dalgitsis	Update Mobility use case results.
1.1	17-12-2025	Josep Calero Santo, Marc Sanchez-Artigas	Add Metabolomics use case.
1.2	23-12-2025	Carlos Segarra	Add C-Cells description in Metabolomics use case.
1.3	26-12-2025	Peini Liu	Final version.

Table of Contents

List of Abbreviations and Acronyms

API	Application Programming Interface
CC	Creative Commons
CDF	Cumulative Distribution Function
CEC	Cloud-Edge Continuum
CH	Challenge
CPR	Cost Per Request
CSV	Comma-separated values
DOI	Digital Object Identifier
EC2	Elastic Compute Cloud
ECS	Elastic Container Service
FaaS	Function as a Service
FN	False Negative
FP	False Positive
FPS	Frame per Second
IPS	Images Per Second
JCT	Job Completion Time
K-NN	k-Nearest Neighbors
KPI	Key Performance Indicator
LSTM	Long Short-Term Memory
MSE	Mean Squared Error
PoC	Proof of Concept
REST	Representational State Transfer
S3	Simple Storage Service
SGX	Software Guard Extensions
SLA	Service Level Agreement
SLO	Service Level Objective
SSIM	Structural Similarity Index Measure
TEE	Trusted Execution Environment
TN	True Negative
TP	True Positive
UUID	Universally Unique Identifier
VA	Video Analytics
WASM	WebAssembly

1 Executive summary

This deliverable reports on the results and experiments from all use cases in the WP, and will report on the experimentation and tests for methods integration on the different use cases. All generated data concerning the workloads and performance of systems and analytics will also be included in the report, with special interest in provide knowledge to other communities related with the use cases.

Each use case starts with the requirements of the business in a Cloud-Edge continuum, and is then followed by its used Cloud-Edge hardware and CloudSkin platform. Finally, each uses case provides their experiments and results as well as some demo showcases. Table 1 shows the main experiments and KPIs that the use cases focus on.

Table 1: Summary of use cases KPIs.

Use case	**Experiments**	**KPIs (KPIs prefixed with "uc" mean use case-specific KPIs (e.g., ucKP1)**
Mobility	Experiment 1: Intelligent application migration between Edge and Cloud; Experiment 2: Real-time service migration and cost analysis; Experiment 3: Dynamic DNS solution.	ucKPI1: App QoS; ucKPI2(KPI2,KPI3,KPI13): Application Performance and Cost; ucKPI3(KPI13): Migration DNS Latency
Metabolomics	Experiment 1: Lithops Serve against state-of-the-art inference systems (CH1); Experiment 2: Lithops Serve evaluation of cost-driven scaling (CH1); Experiment 3: Performance of GEDS-based WebAssembly Units for preprocessing images (CH2); Experiment 4: Evaluation of image reconstruction privacy and latency (CH2); Experiment 5: Evaluation of job completion time under SLO (CH2); Experiment 6: Elastic C-Cell Scaling (CH2)	ucKPI1:Latency; ucKPI2:Throughput; ucKPI3:Performance/$; ucKPI4:Cost($); ucKPI5:SSIM; ucKPI6:Job completion time (JCT) for vector embeddings($)

Surgery	Experiment 1: Smart CPU & GPU allocation for confidential real-time surgical AI models;
	Experiment 2: Predictive auto-scaling streaming infrastructure to handle fluctuating workloads;
	Experiment 3: Advance surgical stream data management across the Cloud-Edge.
		ucKPI1(KPI2): Edge Resource Utilization;
		ucKPI2(KPI3): Real-time Edge Processing;
		ucKPI3(KPI6): Confidentiality (TEE execution);
		ucKPI4(KPI11): Scalability (lower latency streaming);
		ucKPI5(KPI14): Performance (in-transit data management).

Agriculture	Experiment 1. Agricultural Dataspace; Experiment 2. Continuum integration analysis.	ucKP1: Validation of stakeholder requirements. ucKPI2: MAE (s). ucKPI3: R^2. ucKPI4: Duration.

2 Use case: Mobility

2.1 Overview

2.1.1 Business story

Cellnex, a leading provider of telecommunications infrastructure, is undergoing a significant digital transformation to enhance its operational efficiency and service offerings. One of the critical challenges faced by Cellnex is the efficient management of its Video Analytics (VA) services, which are essential for various applications such as security monitoring and traffic management. The traditional methods of service deployment and management are not sufficient to handle the dynamic and complex requirements of modern VA applications. These applications demand high computational power, low latency, and seamless scalability, which are difficult to achieve with conventional infrastructure. To address these challenges, Cellnex has adopted the CloudSkin platform, which leverages advanced containerization techniques and AI-driven orchestration capabilities, as well as supporting orchestration at Cloud-Edge Continuum.

There are two objectives at the mobility use case:

- **Objective 1. Application placement on edge and cloud:** The first experiment focuses on deploying an AI-based VA application for vehicle detection using real-time video data from the circuit. The target application can operate either on the edge node or on cloud resources. The objective is to validate CloudSkin's ability to determine the most suitable execution environment, taking into account service requirements such as latency, processing efficiency and resource availability.
- **Objective 2. Intelligent application migration between edge and cloud:** The second experiment evaluates CloudSkin's capability to migrate the VA application dynamically between edge and cloud, triggered by factors such as QoS demands. This objective aims to validate the AI-driven orchestration mechanisms, ensuring VA application service performance using migration under a dynamic environment.

The expected outcome is a practical demonstration that intelligent orchestration can significantly improve the deployment, performance and manageability of VA services across distributed infrastructures. This validation supports Cellnex's broader strategic goal of strengthening its technological leadership and preparing its infrastructure for future mobility and security services that rely on advanced edge capabilities.

2.1.2 Why this use case needs the compute continuum?

The adoption of a cloud-edge continuum is crucial for VA because it offers a balanced approach to processing the application that combines the benefits of both edge computing and cloud computing.

- **Low-latency, bandwidth saving benefits from edge computing:** edge computing enables processing the video analytics data closer to the source, which reduces the time taken for analytics and could get real-time results.
- **Scalability and flexibility of cloud computing:** cloud computing provides elastic resource allocation and easily scales up and down sufficient computing resources. For instance, when VA has high user demand, it can request computation resources during peak hours. High-performance computing resources power in the cloud can accelerate the complex processing of the video data.
- **Efficiency and Costs:** Utilizing a cloud-edge continuum allows for the dynamic deployment and migration of VA applications across distributed resources based on geographical needs, ensuring efficient response to user requests and events and saving costs.

2.2 Cloud-Edge continuum infrastructure for mobility use case

2.2.1 Cloud-Edge hardware

Cellnex has a testbed located in Castellolí Parcmotor Circuit. More specifically, for the CloudSkin project and mobility use case, cloud-edge hardware has been defined:

- **Cloud (Control Room):** Two virtual machines have been deployed in Lenovo SR650 servers running VMware vSphere to create different virtual machines (VMs) for various services and apps (referred to as the "Local Cloud") with the purpose of running services in "the cloud."
- **Edge (Pole):** Node 1, A Samsung Wisenet PNO-9080R camera captures real-time images from a specific area of the circuit and sends these images to other devices via the RTSP protocol for analysis. Additionally, an SE350 edge server has been deployed in Node 1 with the purpose of running services at "the edge." Node 1 also features an energy control and management system called ORION.

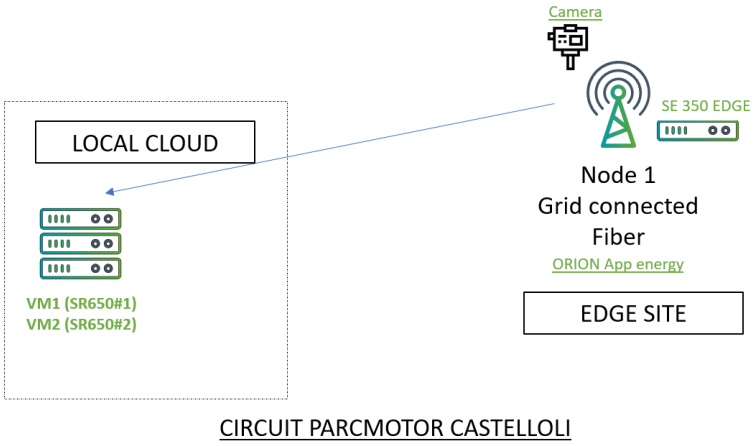


Figure 1: CloudSkin Hardware for Mobility Use Case.

2.2.2 CloudSkin platform

The aim of the CloudSkin mobility use case is to facilitate intelligent service migration between cloud and edge environments, optimizing resource allocation and reducing service latency. Moreover, to provide an innovative approach ensures that video analytics services can be dynamically managed and migrated to meet changing demands, thereby improving the overall quality of service (QoS) and operational efficiency. Figure 2 shows the high-level CloudSkin platform to support the mobility use case. It includes a Monitor to collect the system/service status, an Executor to adjust the system/service, and also a Planner to analyze and understand the scenario to make a decision.

For the software stack towards the Mobility Use case, multiple components are needed. The software components that implement the CloudSkin architecture for the mobility use case are shown in Figure 3.

- **Application Registries:** The application registries serve as the storage location for the containerized VA application, the orchestration artifacts, and the associated services to implement the mobility use case. It includes a container registry for the application and services images, a Helm Chart registry for maintaining the associated cloud-native deployment specifications, and a Block registry, for the orchestration artifacts. More details are provided in Section 2.2.2.
- **Orchestration Platform:** The NearbyOne Orchestrator serves as the multi-site service orchestration engine in the CloudSkin architecture. It enables the lifecycle management of cloud-

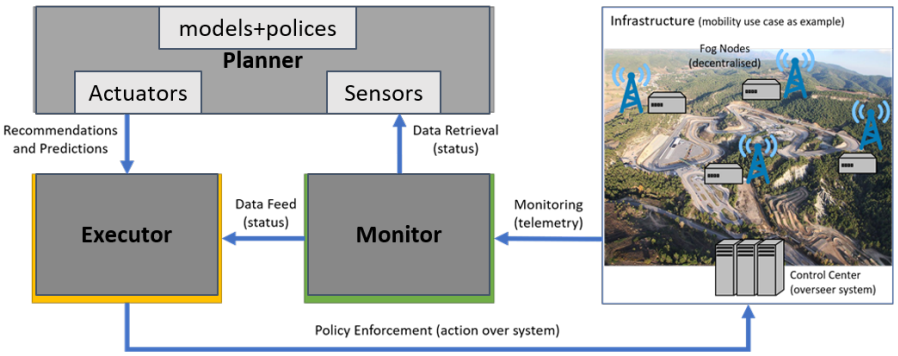


Figure 2: CloudSkin High-level Architecture for the Mobility use case.

native services and infrastructure across heterogeneous edge and cloud sites. NearbyOne simplifies the deployment, scaling, and automation of applications using a declarative approach. It supports closed-loop orchestration, where feedback from external tools (e.g. Planner) triggers real-time decisions via its exposed northbound interface (NBI) API. A detailed description of the Orchestration platform is provided in Section 2.2.2.

- **Learning Plane:** Learning Plane has a set of components to achieve intelligent management. It connects the data for analyzing and triggers the changes within the Cloud Edge Continuum with optimized decisions to achieve business goals. Detailed implementation is in Section 2.2.2.

Registries Besides the CloudSkin software architecture components, other relevant external entities necessary for the operation of the mobility use case are the registries. Registries are the centralized repositories used to store and access application or service essential components like orchestration artifacts and container images.

The orchestration platform, described in Section 2.2.2, interacts with various types of registries, each serving a unique purpose in the orchestration and deployment of applications. Figure 4 shows the NearbyOne orchestration resources (on the left) and the registries associated (on the right).

- **Container Registry**: This is the repository for container Docker images. These images are essential building blocks for deploying services and applications. The registry ensures a reliable and secure distribution of images across different environments. NearbyOne pulls images from public container registries like Docker Hub[1], or private registries, depending on the requirements and security considerations. The CloudSkin container registries are based on Harbor[2] and GitLab[3] projects.

- **Helm Chart Registry**: Helm charts are used to define, install, and upgrade complex Kubernetes applications. The Helm Chart registry is where these charts are stored. The charts in this registry reference the images stored in the Container Registry. CloudSkin Helm Chart registry is implemented under the Harbor open-source project, but the orchestration platform can also interact with public registries that support OCI, such as Docker Hub.

- **Block Registry**: Similar to the Helm Chart Registry, the Block Registry is where the NearbyOne Blocks are stored, which are higher-level components encapsulating a service or application to be managed by NearbyOne orchestration platform. Each Block references a Helm chart and, in turn, container images. The CloudSkin Helm Chart registry is also implemented under the Harbor open-source project.

It is important to note that appropriate access controls and security measures are implemented to protect the integrity and confidentiality of the data in these registries.

[1]https://hub.docker.com/
[2]https://goharbor.io/
[3]https://about.gitlab.com

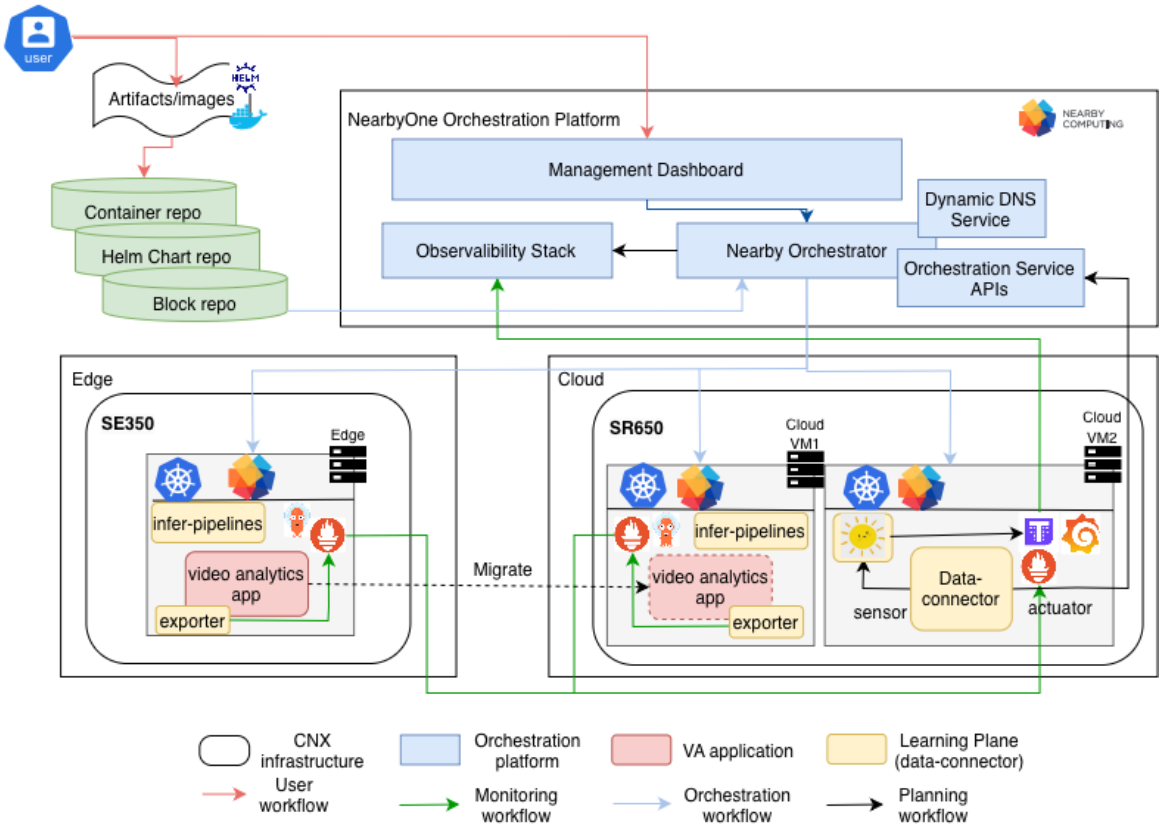


Figure 3: CloudSkin Implementation Architecture.

Orchestration Platform The orchestration platform of the mobility use case is NearbyOne, an edge-to-cloud orchestrator. NearbyOne provides mechanisms to automate and orchestrate the infrastructure located in Castelloli, and the deployment of components, such as the monitoring stack, the learning plane, or the mobility use case applications. The NearbyOne controller itself is a cloud-native platform, packaged as a Helm chart and installed on a Kubernetes platform in a public cloud (AWS), from where it centrally orchestrates applications, services, and infrastructure resources across all managed clusters.

The main components of NearbyOne for the CloudSkin mobility use case are:

1) The Management Dashboard, an intuitive Graphical User Interface (GUI), that provides users with a tailored experience, allowing them to manage their resources, services, and configurations with precision. See CloudSkin deliverable D2.3[1] for more details.

2) The Observability Stack integrates Prometheus, Thanos, Grafana, and MinIO into a scalable multi-cluster monitoring solution (*i.e.*, Monitor). See CloudSkin deliverable D2.3[1] for more details.

3) The Nourthbound Interface (NBI), an API that enables the communication between the NearbyOne orchestrator and the Learning Plane for AI-driven orchestration and management of services and applications (*i.e.*, Executor). See CloudSkin deliverable D2.3[1] for more details.

4) The Nearby Blocks, the core orchestration resources, are the building blocks for application and service deployment and automation. Each Block encapsulates deployment logic, configuration rendering, placement rules, and scaling strategies. Nearby blocks orchestration resources are described as YAML objects that declare the desired state of the system. Then, the platform's reconciliation engine continuously works to converge the actual state of managed resources to match the declared state, ensuring self-healing and eventual consistency. See CloudSkin deliverable D2.3[1] for more details.

5) The Dynamic DNS Service provides a seamless way to maintain stable, user-facing URLs for cloud-native services that may dynamically migrate across multiple Kubernetes clusters. In dis-

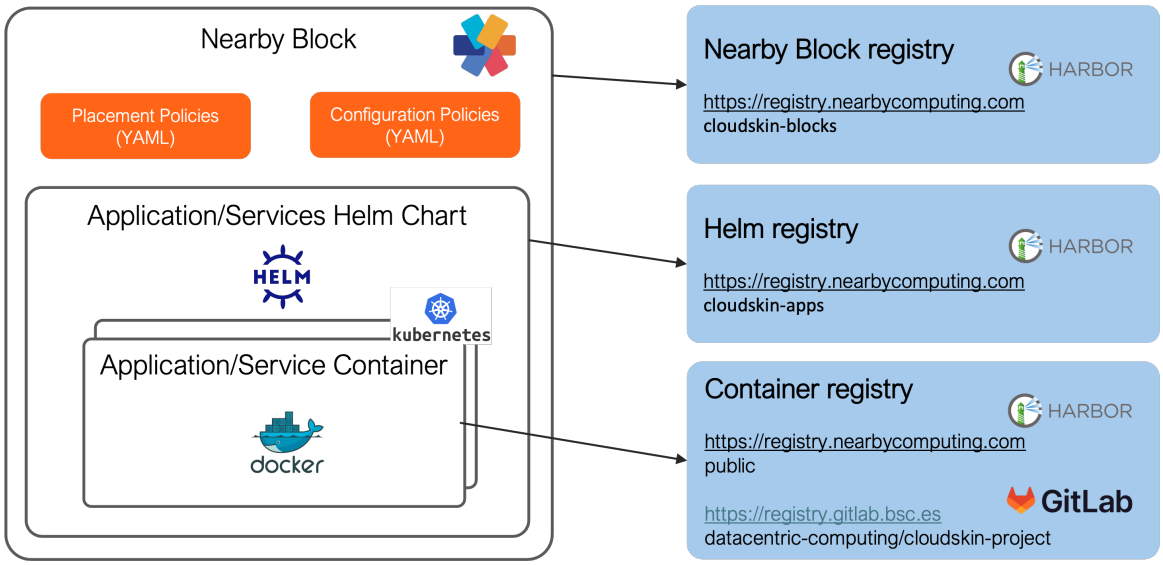


Figure 4: CloudSkin Mobility use case registries.

tributed edge-to-cloud environments, the IP endpoints of applications frequently change as the orchestrator relocates workloads for performance, resilience, or resource-optimization reasons. To prevent disruptions, the dynamic DNS service continuously updates DNS records to reflect the current location of each service. This ensures that client applications can always reach the correct endpoint without requiring any changes in configuration or knowledge of where the service is running. By tightly integrating with the orchestration workflow, the dynamic DNS service acts as a real-time service discovery mechanism, bridging the gap between dynamic service mobility and the need for stable, transparent access. NearbyOne directly manages DNS records during deployment and migration. As a result, client requests to a stable URL are always routed to the correct service endpoint, even as service instances move between clusters.

Figure 5 shows a flow diagram of the DNS service. The NearbyOne orchestrator deploys and migrates services while updating the shared local DNS server to ensure continuous client access. The orchestration begins with the deployment of the video analytics application on the edge Kubernetes cluster. Upon deployment, the orchestrator configures the shared local DNS server to map the service URL to the edge ingress IP. Client applications can then resolve the URL and interact with the service as normal. Meanwhile, the Learning Plane (see Section 2.2.2) observes system metrics and usage patterns. When conditions indicate the need to migrate the service (e.g., due to load or network considerations), it triggers a migration request via the NBI. Upon receiving the trigger, NearbyOne deploys a video analytics application instance on the cloud cluster. When the service is fully initialized, the orchestrator removes the original edge instance and subsequently updates the DNS record to reflect the cloud ingress IP. This approach ensures that service migrations are seamless and transparent to clients. It unifies service orchestration and DNS-based discovery into a single, intelligent control loop that is adaptive, location-aware, and robust in hybrid edge-cloud scenarios.

To support this closed-loop orchestration, the mobility use case is built on a modular and extensible architecture composed of several integrated components. The video analytics application, the dynamic DNS solution, and the observability stack are encapsulated as different Nearby Blocks. The Learning Plane serves as the intelligent trigger, initiating orchestrator actions based on observed system conditions. Together, these components form a cohesive, adaptive framework for managing cloud-edge services with zero-disruption migrations and intent-driven automation.

Learning Plane for Intelligent Service Migration Learning Plane is a concept of using ML methodologies for efficiently solving management challenges, which does not implement a global model or plan. Businesses can focus on what they need to customize Learning Plane to fulfill the manage-

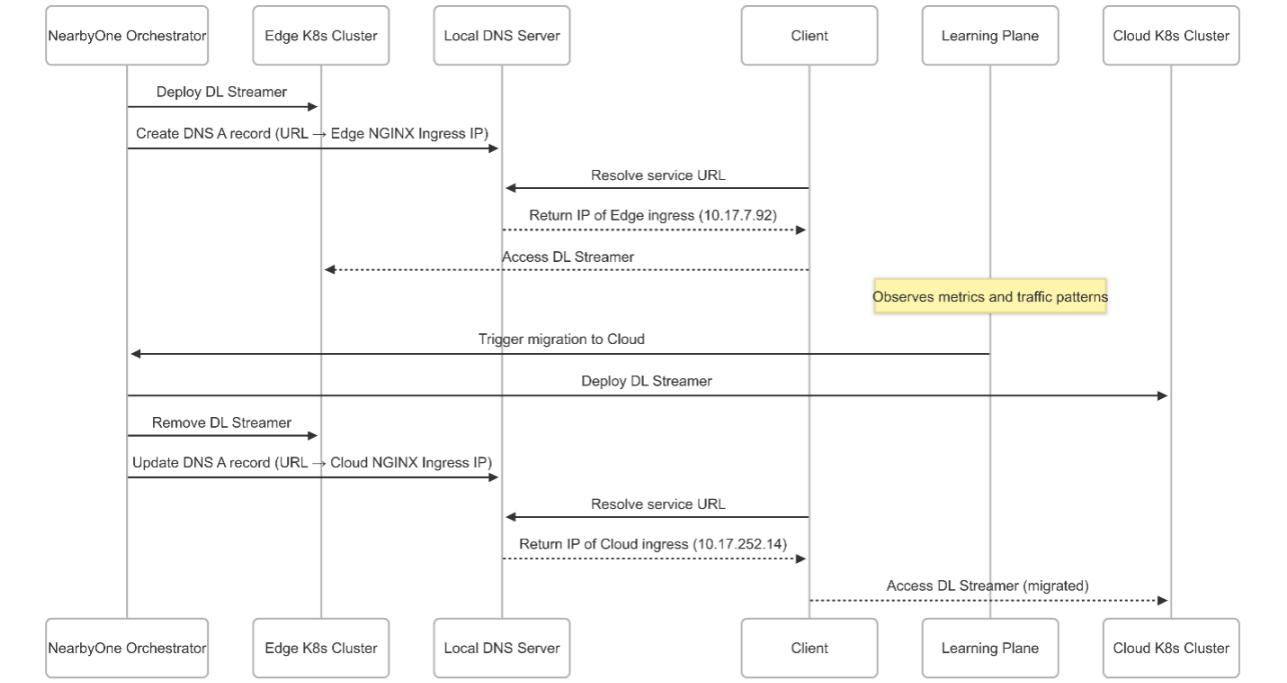


Figure 5: CloudSkin Mobility use case service orchestration and DNS update flow from edge to cloud using NearbyOne.

ment objective. The learning plane abstraction is introduced in the previous paper as data-connector [2] and deliverable D5.3, including a sensor that detects changes in internal and external states, an ML pipeline that responds to relevant observations, and an actuator that activates specific processes within the environment.

Figure 6 shows the mobility use case of intelligent service migration using learning plane under dynamic user demands at Cellnex. The data-connector agent proactively calls model inference to predict the QoS of a target VA application in the cloud and the edge, and also periodically watches the application QoS prediction results with a QoS-aware policy to decide if triggering the VA application migration.

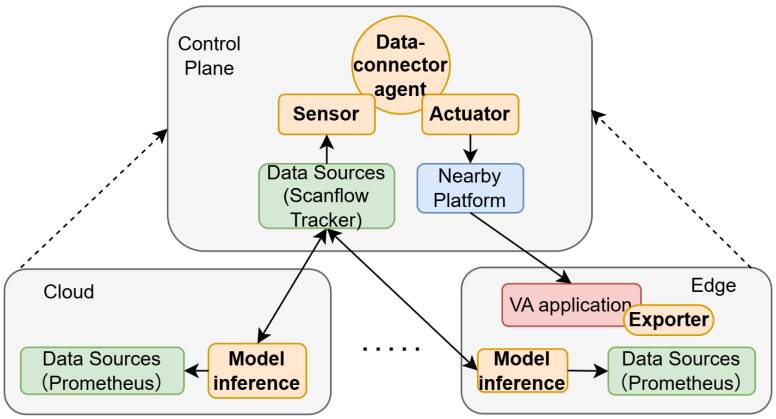


Figure 6: Intelligent migration of VA application using Learning Plane.

1) Data Exporter: Given dynamic user demands within Cellnex, VA application requires service quality monitoring in real time. To achieve this, the application's performance metrics are continu-

ously collected and structured within a JSON object. Then, a JSON exporter[4] is deployed for exporting these application-related metrics into Prometheus-based NBC Monitoring stack with a scraping interval.

2) QoS-aware Migration Policy: The autonomous strategies of the agent for application migration are described in detail in Table 2. The proposed data-connector requires the Data Engineer team to enable the model inference and provide custom functions for sensors and actuators.

Table 2: Agent autonomous management strategy

Agent	***Analyzing and Planning Strategies***
Data-connector agent	Analyzing Strategy: *QoS_predictions* **WHEN** *intervalTrigger*(*5min*, *QoS_prediction*(*data*)) **IF** *successful_call* **THEN** *runWorkflow*(*prediction-pipeline*, *data*) Planning Strategy: *migrate_app* *apiRequest*(*/analyze_qos*) **IF** *app_cluster* $==$ *edge* **AND** *current_cluster_app_qos* $>$ *200ms* **THEN** *Call*(*NearbyOneActuator* : *migrate_service*) **ELSE IF** *app_cluster* $==$ *cloud* **AND** *current_cluster_app_qos* $<$ *45ms* **THEN** *Call*(*NearbyOneActuator* : *migrate_service*) **ELSE THEN** Nothing to do

3) QoS Prediction Model Inference: Model inference contains steps of data loading, QoS prediction and data aggregation. The pipeline periodically predicts the VA QoS with a time window of 5 minutes.

- **Data loading**: This step queries multiple data sources via PromQL syntax to get the status of the nodes and the VA application. Also, the results can be merged into a single DataFrame and saved for the model to use.
- **QoS prediction**: Model inference is to use the pre-trained model for prediction. This step takes the data loaded in the last time window, preprocesses the data and predicts the application QoS in the edge and cloud using the pre-trained Informer model (for dataset 1) or Random Forest model with time features (for dataset 2). The model is pretrained in Deliverable D5.3.
- **Data Aggregation**: After predicting the QoS for the next time window, this step aggregates the predicted data by statistics such as max/average. The agent policy can see those analytics and decide on their usage. In the Cellnex use case experiment, the max is used.

4) Data-connector Agent: To implement the agent towards Cellnex, custom functions of these main components should be developed, i.e., sensor and actuator.

- **Sensor**: Sensor defines which data in the shared artifacts the agent should watch, and the policy to decide if triggering the actuator. In our usecase, the sensor evaluates the predicted QoS data and triggers the service migration based on QoS constraints (see Table 2)
- **Actuator**: Actuator is used to connect different platforms to execute migrations. In Cellnex CEC environment, we use NearbyOne orchestrator to seamlessly migrate our application. Actuator can provide the decision to the orchestrator using NearbyOne northbound interface API.

[4] `https://github.com/prometheus-community/json_exporter`

2.3 Experiments, KPIs and benchmarks

This use case conducts several experiments to show the use case KPIs given in Table 3. All the demo and experiments are running in testbed described in section 2.2. On the one side, we conducted an experiment for VA application placement on our testbed to demonstrate application offloading and performance KPIs, and on the other side, we used the testbed for QoS-aware service migration in a real environment.

Table 3: Summary of use case-specific KPIs for Mobility.

ucKPI	Description
ucKPI1:App QoS	Proactive migration achieves slightly lower F1-score (86.8% vs 87.7%) but higher precision (90.9% vs 87.7%), and reduces SLA violation time by 71% (360s vs 1245.6s)
ucKPI2 (KPI2,KPI3,KPI13): Migration Application Performance and Cost	Proactive migration ensures higher SLA compliance (98.75% vs 95.67%) with slightly higher total daily cost (5.39€ vs 4.91€), but reduces SLA penalties and unnecessary migrations compared to Reactive, yielding better cost-effectiveness per SLA-compliant hour. When compared to Cloud-only baseline, proactive reduces total daily deployment cost by 3× (5.39€ vs 15.97€).
ucKPI3 (KPI13): Migration DNS Latency	The dynamic DNS approach reduces propagation latency by 67% relative to ExternalDNS.

Application placement on edge and cloud This very first experiment pursues the successful deployment of a video analytics (VA) application on the edge and the cloud. In this experiment, we deployed a VA application through NBC orchestration platform on CNX infrastructure, and we enabled fully offloading of the application from the cloud to the edge. The results and KPIs are shown in the previous deliverable D2.3.

Intelligent application migration on edge and cloud This experiment consists of collecting edge and cloud data towards learning multi-dimensional time-series for the future recommendations of application migration in a dynamic environment. In this experiment, we explore VA applications and use an ML model we trained in deliverable D5.3 to predict service QoS and trigger service migration in a dynamic Cloud-Edge continuum. The results are shown in section 2.4.

2.4 Results

Before presenting the experimental results, we clarify the migration decision logic and the evaluation metrics used throughout this section. A migration decision is triggered whenever the estimated service latency exceeds a predefined Service Level Agreement (SLA) threshold. For **reactive migration**, the decision is based on the observed service latency, whereas for **proactive migration**, the decision is based on the predicted future service latency. Whether a migration was actually needed is determined a posteriori using ground-truth future latency: a migration is considered necessary if the future latency indeed violates the SLA.

Reactive vs. Proactive Migration Strategies Both migration strategies execute the same decision pipeline at a fixed interval of 5 minutes, but differ in the information used to trigger a migration. In the **reactive approach**, the system computes the average observed service latency over the last 5 minutes. This temporal averaging is applied to mitigate short-term instability and noise in the application latency. If this averaged observed latency exceeds the SLA threshold, a migration is triggered.

In contrast, the **proactive approach** initiates a prediction pipeline every 5 minutes. At each decision point, the system fetches the most recent telemetry window, including historical latency and multivariate resource metrics, preprocesses the data, and applies a trained machine learning model to predict the service latency in the future 5–10 minute interval, for more details on the pipeline stages or the models used, see Deliverable 5.3 Section 4.2. A migration is triggered if the maximum

predicted latency within this future horizon exceeds the SLA threshold. This allows the system to anticipate upcoming SLA violations and migrate the service before degradation is observed.

Experiment 1: (1) Intelligent Service Migration Results for Dataset 1 This experiment assesses the impact of intelligent migration strategies. We compare proactive service migration, where migration is driven by the service latency predicted by our Informer model if the max predicted service latency is over than SLA, to the traditional reactive migration, which only responds if the observed service latency is over than SLA.

Migration decisions are evaluated using a confusion-matrix-based formulation. A **true positive (TP)** corresponds to a migration that was triggered and was indeed necessary (future latency violates the SLA), a **false positive (FP)** to an unnecessary migration, a **false negative (FN)** to a missed migration where an SLA violation occurred without migration, and a **true negative (TN)** to correctly not migrating when no violation occurred. From these quantities, we compute precision (fraction of triggered migrations that were necessary), recall (fraction of necessary migrations that were correctly triggered), and the F1-score, which balances precision and recall.

Table 4 shows the confusion metrics of reactive and proactive migration approaches. We observe that the proactive approach achieves a much higher absolute amount of migrations (TP+FP), 2002 migrations against 925. Also, proactive migration performs a better migration detection by reducing the number of missed migrations from 4900 to 3915.

Table 4: Confusion matrix for Reactive and Proactive approaches.

Approach	TP	FP	FN	TN
Reactive	619	306	4900	23810
Proactive	1604	398	3915	23718

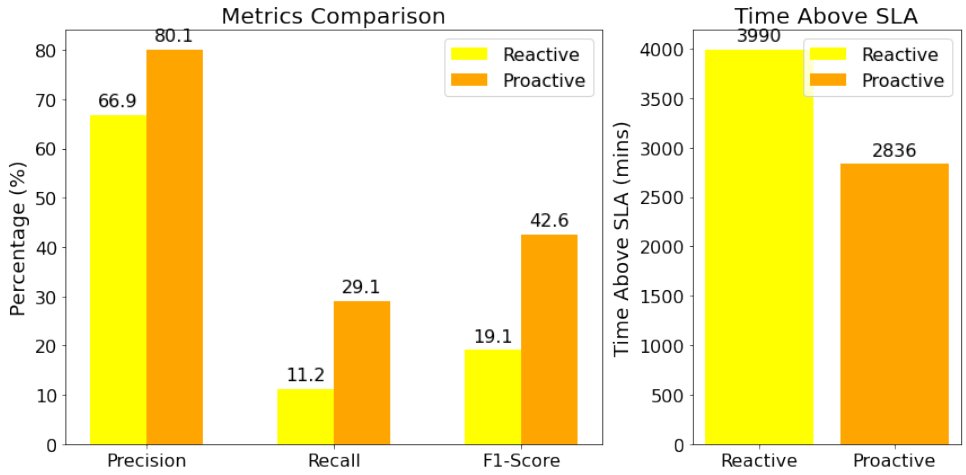


Figure 7: Reactive vs. Proactive Migration Strategies.

Figure 7 presents the comparison of using reactive and proactive migration strategies. The left plot shows the Precision, Recall and FI-Score of two strategies. Specifically, proactive migration achieves 13.9% better precision compared to reactive migration, indicating a high proportion of the predicted migrations are actually necessary and fewer false alarms in the proactive strategy. Proactive has 17.9% better recall than reactive, showing a stronger capacity to catch missed migrations. F1-Score, which balances precision and recall, proactive migration outperform reactive migration for 23.5%. Proactive approach demonstrates better performance across those key metrics compared to the reactive approach, meaning that proactive migration not only makes more accurate decisions but also captures more missed migration cases, thus considered a more effective strategy. The right plot

shows the amount of time spent breaking the QoS constraints for each type of strategy. Under tested workloads, the proactive method leads to lower QoS broken constraint time from 3990 minutes to 2836 minutes, reducing SLA violation time up to 28.9%. This means that a proactive strategy can achieve better service quality and reduce the cost of the company from paying the SLA violations.

(2) Intelligent Service Migration Results for Dataset 2 This experiment evaluates reactive and proactive migration strategies under the updated workload profile (i.e., Dataset 2), as described in Deliverable D5.3, Section 4.2.2. The proactive strategy relies on a Random Forest predictor, selected as the best-performing model in Deliverable D5.3, Section 4.2.4. Table 5 reports the confusion matrices for both approaches. Compared to reactive migration, the proactive strategy triggers fewer total migrations (497 vs. 546) and substantially reduces unnecessary migrations, lowering false positives from 67 to 45. This reduction comes at the cost of a higher number of missed migrations (93 vs. 67), trading sensitivity for selectivity.

Table 5: Confusion matrix for Reactive and Proactive approaches on Dataset 2.

Approach	TP	FP	FN	TN
Reactive	452	45	93	732
Proactive	479	67	67	709

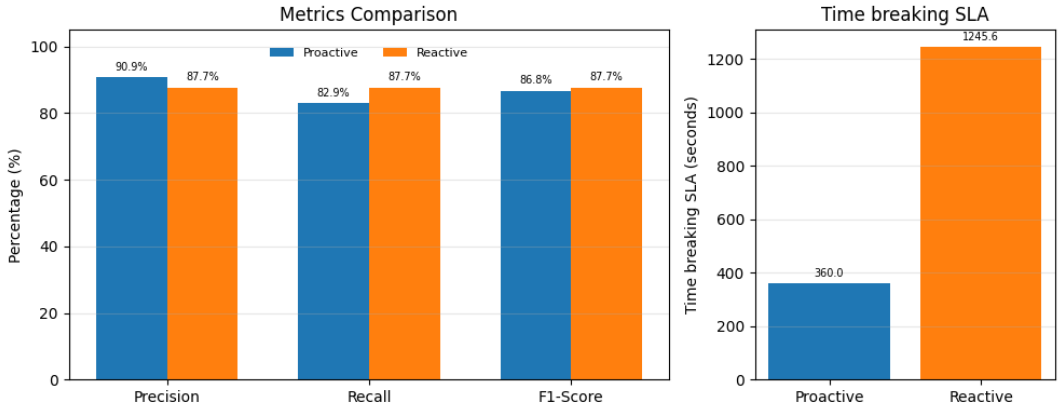


Figure 8: Reactive vs. Proactive Migration Strategies in Dataset 2.

Figure 8 summarizes the performance comparison. The left plot reports precision, recall, and F1-score. Proactive migration achieves higher precision (90.9% vs. 87.7%), indicating more accurate migration decisions with fewer false alarms, whereas reactive migration attains higher recall (87.7% vs. 82.9%). Overall, both strategies obtain comparable F1-scores, with reactive slightly outperforming proactive (87.7% vs. 86.8%).

Unlike Dataset 1, where the proactive strategy consistently outperformed the reactive baseline across all metrics, Dataset 2 exposes the system to highly intensive workloads rather than the expected operational workload distribution. Under these extreme conditions, both strategies exhibit similar detection performance, leading to a marginally higher F1-score for the reactive approach. However, the right plot from Figure 8 shows that when considering the total time spent violating SLA constraints under realistic deployment conditions, proactive migration significantly outperforms reactive migration, reducing SLA violation time from 1245.6 seconds to 360 seconds (a reduction of approximately 71%). This demonstrates that, despite a small trade-off in recall under stress-test conditions, proactive migration delivers substantially better SLA compliance in practice.

Beyond aggregate metrics, Figure 9 provides a time-resolved visualization of the same experiment under an identical workload execution. The figure shows, for each strategy, both the migration decisions taken over time and the resulting deployment location of the application (Cloud or Edge).

This qualitative comparison highlights how proactive migration anticipates upcoming SLA violations and relocates the service earlier, reducing the duration of time spent above the SLA threshold compared to the reactive strategy, which only responds after violations are already observed.

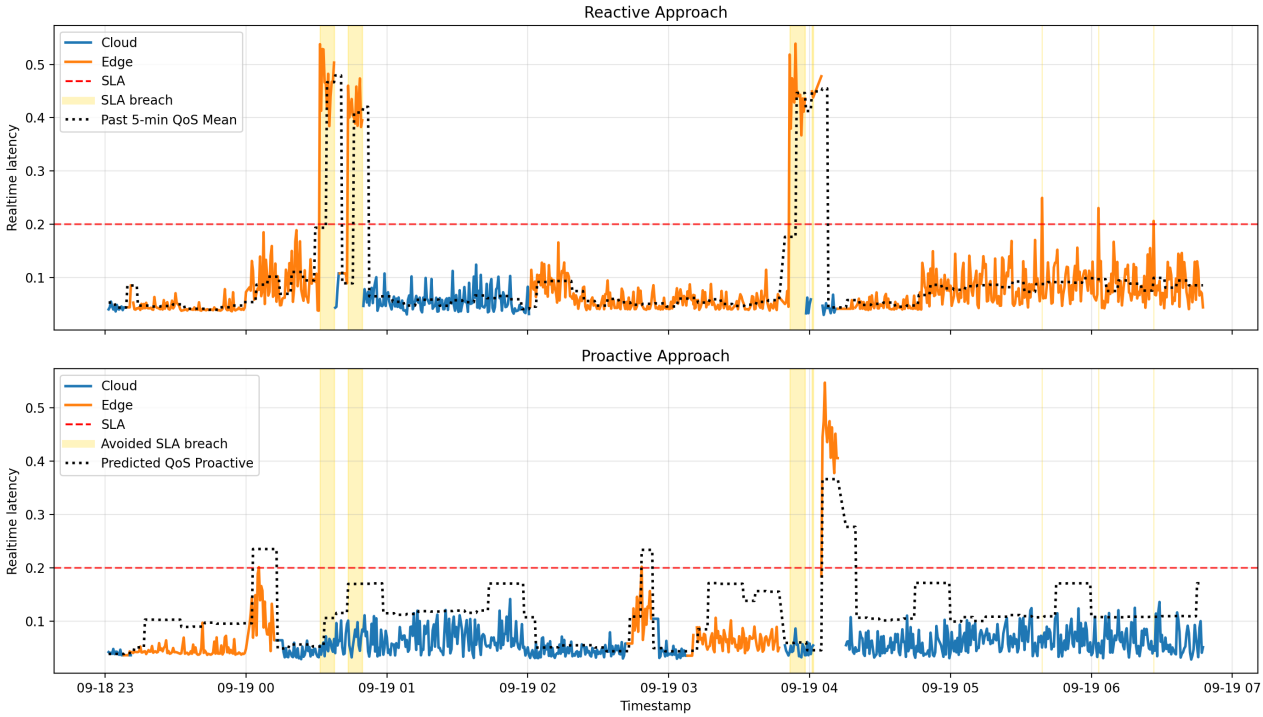


Figure 9: Mobility use case demo comparing reactive and proactive migration under the same workload execution. The figure shows the migration decisions over time and the resulting service placement (Cloud or Edge), illustrating how proactive migration anticipates SLA violations and reduces the time spent above the SLA threshold compared to the reactive strategy.

Experiment 2: Real-time Service Migration and Cost Analysis We have run several full-day experiments in the real environment to test the real-time service migration driven by our proactive strategies, and we calculated the time breaking SLA and cost metrics.

We define the total operational cost of the service during a full day experiment as the sum of infrastructure cost C_{infra}, SLA violation penalties C_{SLA} and the cost of the migrations applied, shown in Equation 1:

$$C_{\text{total}} = C_{\text{infra}} + C_{\text{SLA}} + C_{\text{migrations}} \tag{1}$$

Infrastructure Cost There are two types of infrastructures analyzed. *The cloud:* our baseline, utilizing only on-demand cloud services; and *The hybrid infrastructure:* utilizing on-demand cloud services together with on-premise edge devices. Equation 2 details the infrastructure cost of our baseline, while Equation 3 details the hybrid cost, respectively. Let r_{cloud} and r_{edge} be the respective cost rates (€/s) for Cloud and Edge resources, and let C_{CP} be the cost of having a control plane set up on-demand, shared by both approaches. The infrastructure costs are:

$$C_{\text{infra}_{cloud}} = (t_{\text{cloud}} + t_{\text{edge}} + t_{\text{idle}})\, r_{\text{cloud}} + C_{\text{CP}} \tag{2}$$

$$C_{\text{infra}_{hybrid}} = t_{\text{cloud}}\, r_{\text{cloud}} + t_{\text{edge}}\, r_{\text{edge}} + t_{idle}\, r_{\text{edge}} + C_{\text{CP}} \tag{3}$$

Where t_{cloud}, t_{edge} are the time spent in cloud and edge respectively while the t_{idle} is the time outside those working hours, where the infrastructure still needs to be maintained. For our baseline, all the time is spent in t_{cloud}.

Control Plane Cost Shared across all the experimented approaches, we utilize an instance with 2 vCPUs and 8GB of memory, as well as 100 GB of storage service, which costs \$0.0736/hour [3] and \$0.088/GB-month respectively, summing up to a total monthly cost of 53,4€:

$$C_{\text{CP}} = \$0.0736 \cdot 24h \cdot 30\,days + \$0.088 \cdot 100\,GB = \$61.79 \tag{4}$$

Cloud Cost For reference, our Cloud has 16 CPUs, thus taking a 16 vCPU AWS EC2 instance costs \$0.688/hour [3] ($r_{Cloud} \approx 0.0001645$€/s).

Edge Cost The Edge rate represents the cost of the electrical grid associated with operating the Edge server. Using the experimental power traces, we compute the Edge cost by integrating the measured power usage over time, as illustrated in Figure 10. Such high power figures are expected, as our Edge node corresponds to an enterprise-grade server rather than a lightweight embedded device. We consider it an *Edge* node within the context of our use case due to the computational complexity of the deployed application, which could not feasibly run on traditional low-power edge hardware. Furthermore, the Edge servers are supported by a solar panel installation located within the Cellnex infrastructure, contributing to a reduction in the effective grid energy cost through the use of renewable power.

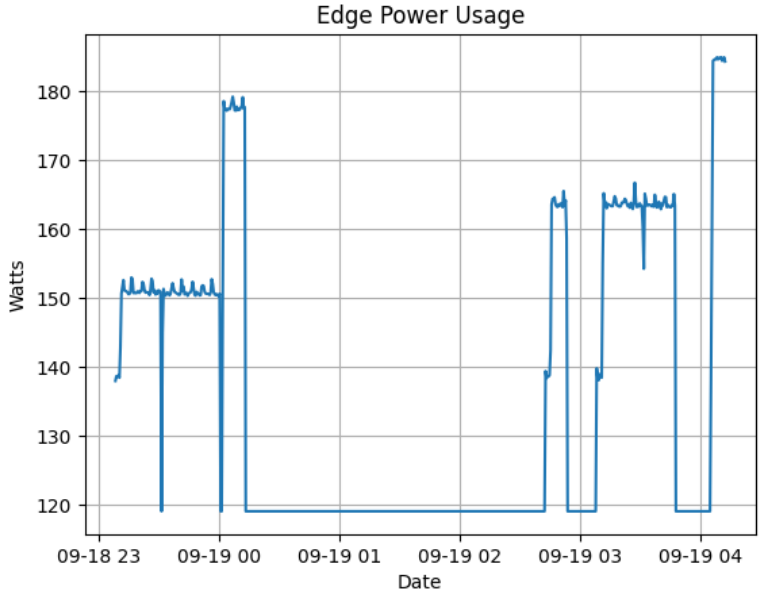


Figure 10: Edge Power usage trace across an 8-hour experiment workload.

We simplify by considering the average business energy price in Spain 0.137€/KWh [4]. The trace from Figure 10 shows the energy usage during a full experiment (8 hours) corresponding to $(t_{edge} + t_{cloud})$, while the rest of the time, we will consider it to be idle t_{idle}.

SLA Breach Cost Different industry standards are applied for defining SLA breach costs. For instance, Amazon splits the SLA breach impact in multiple ranges depending on the total percentage of unmet availability [5]. However, for our real-time inference service, we have availabilty at all times, and our SLA is based on latency thresholds, affecting the perceived QoS by the user, which has an economic impact that grows with the severity of the violation. Following the cost modeling approach introduced in [6], the impact of an SLA breach can be described by a penalty function $P(q)$ that maps the degree of QoS degradation (e.g., latency) to a monetary loss, as seen in Equation 5. The penalty may follow different shapes depending on how sensitive the application is to QoS degradation, for our real-time latency-sensitive use-case, defining a Penalty model $P(q)$ that increases the penalty for larger SLA violations q is key, as the user perceives a very different QoS for large violations compared

to small ones.

$$C_{\text{SLA}} = \int P(q(t))dt \tag{5}$$

$$P(q) = \alpha(e^{\beta(q-q_{SLA})} - 1) \tag{6}$$

Thus, in Equation 6 we define the penalty model as an exponential function in which the penalty grows largely as the latency increases, where the parameters α and β would be chosen by every specific business use-case. Other options such as piecewise-linear models [6], with different slopes of penalty for small or large violations could be used depending on the severity of the impact of larger QoS (impact growth linear or exponential).

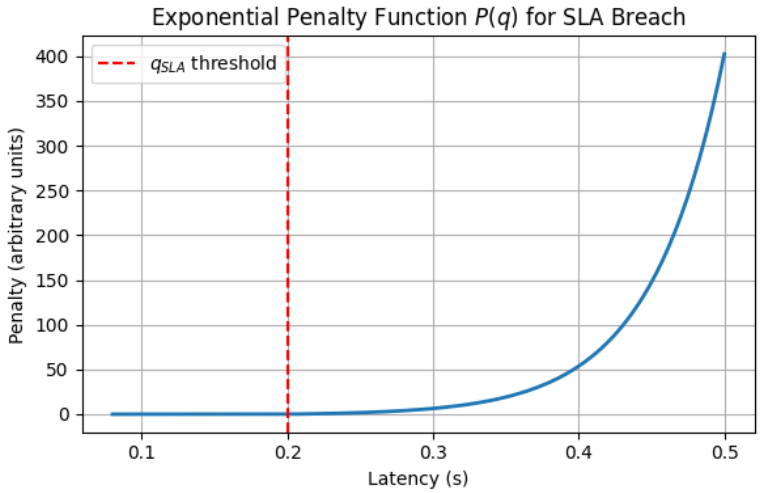


Figure 11: Exponential growth of the cost of SLA breach.

Figure 11 illustrates an example of the exponential penalty model $P(q)$ proposed in Equation 6, showing the accelerated growth of penalty as latency increases beyond the SLA threshold. For simplicity, parameters α and β are chosen to demonstrate the qualitative behavior of the model rather than any specific business calibration.

Migration Cost Migrations between the Edge and the Cloud are not instantaneous nor entirely seamless. As detailed in deliverable D5.3 Section 2.2.2, the current migration mechanism introduces a minimal QoS impact, which is considered negligible for simplification. However, the migration process itself requires a significant amount of time during which both Cloud-Edge resources remain active simultaneously. This overlapping resource utilization represents an additional operational expense, which we define as the *cost of migration*, and is directly proportional to the migration duration and the hourly rates of the resources involved.

Edge to Cloud. When migrating from the Edge to the Cloud, a delay is introduced due to the initialization and booting of the Cloud instance. In our setup, this process lasts approximately three minutes, during which both Edge and Cloud resources are running concurrently. The cost of this transition is therefore computed as the combined cost of both systems for that duration:

$C_{\text{E}\to\text{C}} = 0.029584€$ per migration.

Cloud to Edge. When migrating from the Cloud back to the Edge, the migration time is significantly shorter (around 30 seconds), as the Edge infrastructure is already provisioned and ready to receive workloads. During this time, both environments remain active, resulting in a smaller but still measurable additional cost:

$C_{\text{C}\to\text{E}} = 0.000167€$ per migration.

Approach	Time Fulfilling SLA(%)	Total(€)	Control plane(€)	Cloud(€)	Edge(€)
Proactive	98.75	5.39	1.77	3.02	0.42
Reactive	95.67	4.91	1.77	2.62	0.42
Cloud only	99.68	15.97	1.77	14.20	0.00

Table 6: Daily cost comparison between evaluated approaches. Proactive costs are calculated with the best model for our use case, Random Forest, but all models explored in deliverable D5.3 Section 4.2.4 have been tested. Total cost includes all except SLA breach cost, which is left generalized.

Table 6 summarizes the performance and cost comparison between the evaluated approaches. Showing total costs which include infrastructure and migrations costs, while detailing each of the instances cost. The proposed hybrid approaches (Random Forest and Reactive) significantly reduce the overall deployment cost compared to the *Cloud-only* baseline, while maintaining a high percentage of time below the SLA threshold. Although the Reactive strategy achieves a slightly lower total cost, this comes at the expense of a noticeable decrease in SLA compliance, highlighting the trade-off between cost efficiency and QoS preservation.

Experiment 3: Dynamic DNS solution To evaluate how dynamic DNS updates support seamless application mobility across the edge–cloud continuum, this experiment focuses on the requirements emerging from the Mobility use case, where a Video Analytics (VA) service must be transparently relocated between edge and cloud resources without disrupting client access. In this scenario, maintaining uninterrupted service during application migration is essential: as the VA workload moves in response to latency, load, or QoS triggers, clients must always resolve the service's domain name to its current execution site. The objective of the experiment is therefore to assess how tightly integrated, orchestrator-driven DNS reconfiguration enables fast, consistent, and user-transparent redirection of traffic compared to conventional cloud-based DNS mechanisms. Although the Mobility use case involves migrating a deep-learning–based VA application, for experimental reproducibility and controlled measurement we instead use a lightweight NGINX-based stateless service. This allows us to emulate the application's lifecycle operations while isolating the impact of DNS reconfiguration itself.

To better investigate the benefits of our DNS-integrated orchestration approach, we implemented a comparative baseline using the well-established ExternalDNS with AWS Route53. In this baseline setup, both the edge and cloud Kubernetes clusters run ExternalDNS, which continuously monitors Ingress resources and updates corresponding A records in Route53. When an application is migrated, its Ingress object is recreated in the target cluster, triggering ExternalDNS to update the domain's DNS record to reflect the new service IP via Route53. In this configuration, ExternalDNS functions as the DNS update agent, while the orchestrator remains responsible solely for managing application deployments and migrations. This loose coupling leads to a more asynchronous update model in which DNS changes rely on ExternalDNS's polling intervals and reconciliation logic, potentially resulting in delays or temporary inconsistencies after migrations.

By contrast, our solution integrates DNS updates directly into the orchestration workflow. This tight coupling enables synchronous, atomic migration steps where DNS configuration is updated immediately after deployment, ensuring that clients always resolve to the correct execution site. As a result, the system minimizes propagation latency and avoids transient states with outdated DNS records. To assess DNS reconfiguration latency during service migration, we developed a custom parallel script that continuously performs DNS resolution requests for each application hostname. Each query is sent once per second and targets the authoritative name server directly to avoid caching effects. The script records the time at which each hostname first resolves to its updated IP address, allowing us to measure the delay between migration initiation and DNS propagation. Service migrations are triggered externally by invoking the NBI exposed by the NearbyOne orchestrator, simulating policy-driven control signals. To increase request frequency and study performance under

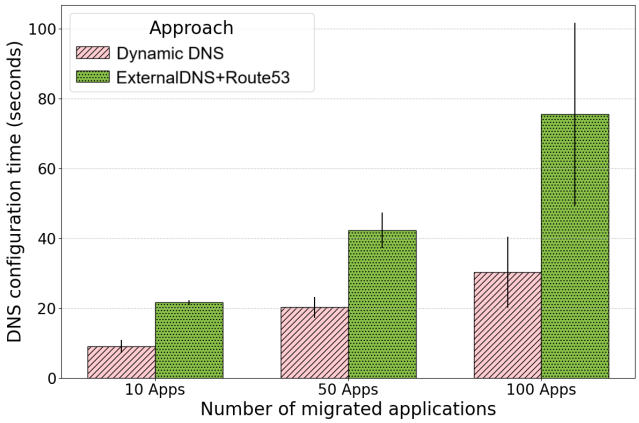


Figure 12: Average DNS A record reconfiguration time after migrations

higher query loads, we additionally employ dnspyre, an open-source benchmarking tool capable of generating large volumes of DNS queries per second. This dual approach provides both fine-grained temporal visibility and stress-testing of the DNS subsystem.

Figure 12 shows the average DNS reconfiguration time and standard deviation observed during the migration of 10, 50, and 100 applications, comparing our proposed dynamic DNS solution with the baseline approach using ExternalDNS and Route53. Reconfiguration time is defined as the delay between triggering a migration and the first successful resolution of the updated A record. As the number of migrated applications increases, dynamic DNS consistently maintains lower reconfiguration times with significantly less variance. For example, at 100 applications, dynamic DNS achieves an average delay of under 30 seconds, whereas the baseline exceeds 90 seconds with high variability. These results highlight the scalability and determinism of tightly orchestrated DNS updates using a local CoreDNS instance, in contrast to the asynchronous and eventually consistent behavior of ExternalDNS with Route53. Beyond latency, operational costs associated with DNS management are also relevant. Cloud DNS providers such as AWS charge per update and per query, which can become significant in large-scale or highly dynamic scenarios. By leveraging a locally hosted DNS service, our approach reduces reliance on external APIs and minimizes ongoing operational expenses.

2.5 Demos

Demo 1: QoS-driven dynamic migration of VA tasks between cloud and edge The demonstration of dynamic migration of VA tasks between cloud and edge is divided into the following sections.

Section 1: Proactive migration of the application (Intelligent service migration) This section demonstrates the proactive migration of the target application during a simple 30-min distribution workload, triggering both Edge-Cloud and Cloud-Edge application migrations.

The recorded screen (see Figure 13) is divided into four quadrants to better understand the entire workflow. The top-left quadrant contains a Grafana dashboard showing the workload evolution of the DL Streamer application and its average latency, and the top-right quadrant shows the Argo UI, where Data Engineer pipelines are scheduled and run every 5 minutes. The bottom-right quadrant displays the Scanflow Tracker (an MLflow instance) where the Learning Plane records the metrics and parameters of each Data Engineer pipeline's run, and the bottom-left quadrant presents a Linux terminal printing the Scanflow Planner logs, where any request sent to the Scanflow QoS analysis sensor and its logs are recorded.

The application's migrations shown in the video go as follows: every 5 minutes a new Argo Workflow is run, creating a new MLflow experiment run, retrieving the last 5-min historical data of the application performance and latency stored in a Thanos service by means of the PromCSV python library, and uploading the results as a CSV file to the Scanflow Tracker's experiment run; after that, the Prediction stage takes the CSV file and infers the latency values for the next 5 min-

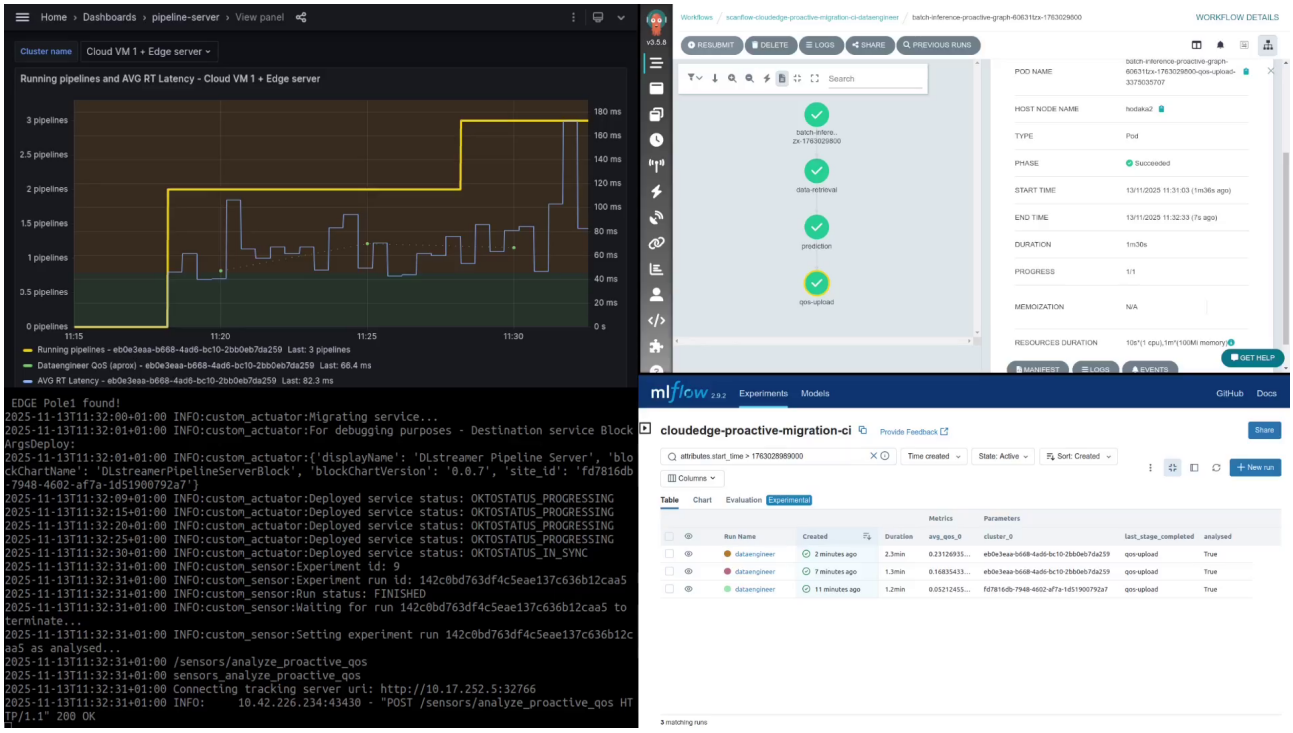


Figure 13: Mobility use case demo: Proactive migration of the application (Learning Plane).

utes, locally storing them into a temporal CSV file. The last stage, QoS upload, reads the temporal CSV file generated by the Prediction stage and computes both the average and max latency value from that file, uploads them alongside the cluster ID where the application is running, and triggers the Scanflow Planner's QoS analysis sensor. The Scanflow Planner then retrieves the latest values uploaded to that experiment's run, and if it finds out that the predicted latency value violates the SLA defined for each cluster type (Edge or Cloud), then it proceeds to trigger the application migration by sending the required API requests to the Nearby One orchestrator. The Scanflow Planner logs show how the application migration is progressing by showing the NearbyOne Service block status, going from "OKTOSTATUS_PROCESSING" to "OKTOSTATUS_IN_SYNC" when completed. This workflow applies to both Edge-to-Cloud and Cloud-to-Edge application migrations.

Section 2: Proactive migration of the application (Dynamic DNS resolution) This section demonstrates the closed-loop workflow that migrates the cloud-native video analytics application between Kubernetes clusters while preserving uninterrupted client access through dynamic DNS updates. At the core of the architecture, the NearbyOne orchestrator manages services as modular Nearby Blocks and performs deployments, migrations, and DNS updates. Dynamic DNS resolution is provided by CoreDNS, configured via Helm and a ConfigMap containing A records that map the stable service URL to the current ingress IP.

The recorded screen (see Figure 14) is divided into four quadrants to make the workflow visible. The top-right quadrant shows the NearbyOne dashboard, where services are presented as Nearby Blocks, including DL Streamer, a CoreDNS service on the Edge, and the Observability stack. The bottom-left quadrant presents a Linux terminal organized around three DNS-related elements: the CoreDNS ConfigMap with A records that map the service URL to IP endpoints, the CoreDNS pods, and the Helm release controlling CoreDNS configuration. The top-left quadrant contains a Grafana dashboard, and the bottom-right quadrant displays the Learning Plane.

Initially, DL Streamer runs on Cloud 1 and CoreDNS maps the service URL to the Cloud 1 ingress IP (10.17.252.14). The Learning Plane then triggers a migration based on predicted latency, and the orchestrator issues a create application request to deploy DL Streamer on the Edge cluster. The Near-

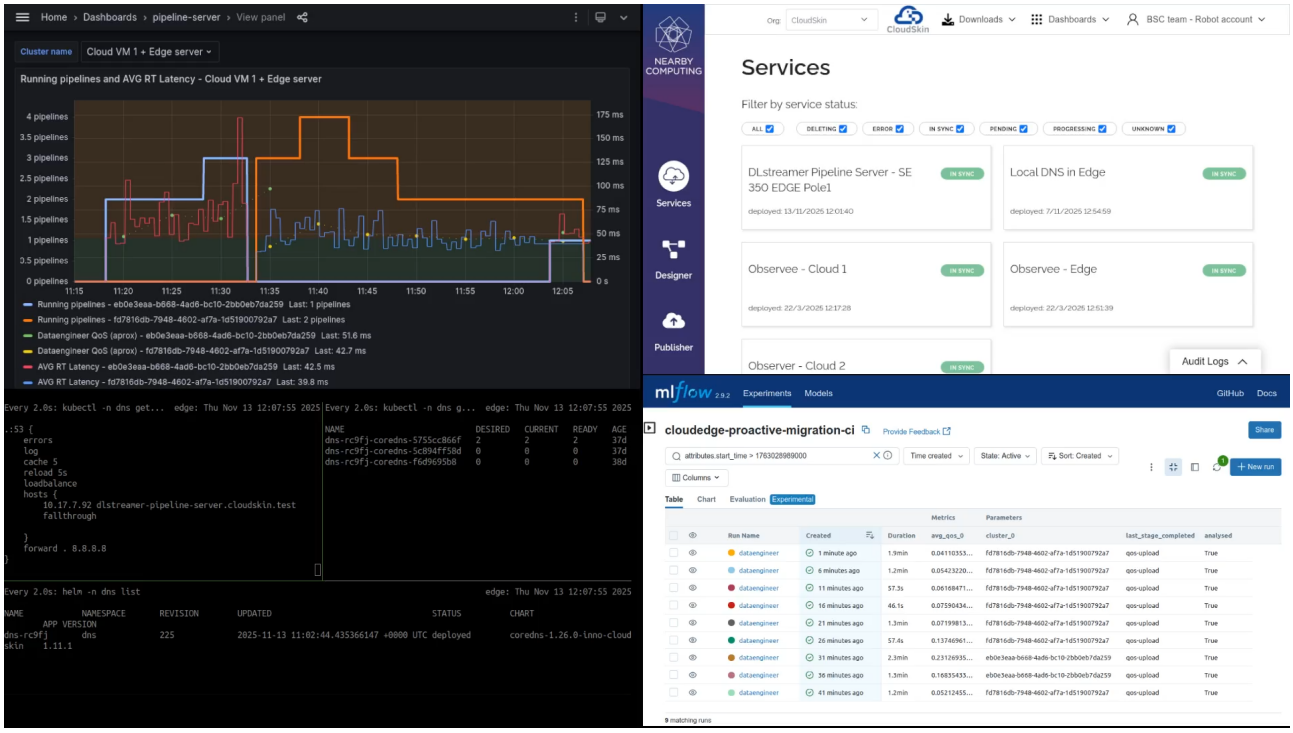


Figure 14: Mobility use case demo: Proactive migration of the application (Dynamic DNS resolution).

byOne dashboard shows the new DL Streamer instance being created on the Edge and, once deployment completes, the original Cloud 1 instance is terminated. During this handover, the CoreDNS ConfigMap temporarily clears its A record list, removing the mapping to 10.17.252.14. CoreDNS pods restart and the Helm release updates to propagate the DNS change coherently across components. The ConfigMap then updates with a new A record that maps the service URL to the Edge ingress IP (10.17.7.92). From this point forward, the service remains accessible via the same stable URL, now resolving to the Edge endpoint, and the migration completes without any client-side configuration changes.

Section 3: Reactive vs. Proactive migration comparison This section demonstrates the advantages of the Proactive approach compared to the Reactive one.

The recorded plot (see Figure 9) is composed of 2 aligned subplots generated from the same 8-hour workload distribution execution for each migration strategy, showing when the application has been migrated from the Edge cluster to the Cloud cluster and the other way around, as well as when a Proactive migration avoids an SLA breach that happened in the Reactive migration experiment.

3 Use case: Metabolomics

3.1 Overview

The `METASPACE` platform[5] integrates a deep learning–based service for recognizing off-sample mass spectrometry images called `OffSampleAI`. As described in prior deliverables, the production-grade `OffSampleAI` inference service leverages AWS ECS to perform image classification at scale. Although this architecture provides elasticity, the `OffSampleAI` service experiences pronounced idle periods due to highly variable, **unpredictable** workloads, as discussed in depth in deliverable D5.3.

The `OffSampleAI` service currently relies on a reactive feedback-control autoscaling strategy, where AWS ECS continuously monitors running instances and adjusts their number based on metrics such as average CPU utilization. In the current configuration, at least one instance is always kept active, and when CPU usage exceeds 80%, the auto-scaler launches four additional container instances, up

[5] `https://metaspace2020.org/`

to a maximum of nine. Because scaling actions are triggered only after increased load is detected, the system suffers from significant provisioning delays under highly variable workloads. This reactive behavior results in unnecessarily long job completion times and further amplifies cost inefficiencies. Consequently, the existing solution is not **cost-efficient** (**Challenge CH1**).

Very importantly, the current implementation lacks support for **privacy-preserving processing of sensitive images** (**Challenge CH2**), underscoring another critical limitation of the existing approach. Currently, sensitive images are processed on AWS ECS in plaintext, allowing AWS to potentially access or infer information from the sensitive data.

To address these challenges, we developed `Lithops` Serve, a novel open-source inference system that unifies **serverless functions for cost-efficient processing (CH1)** with **confidential containers on an on-premises edge cluster (CH2)**. Through a single Python API, `Lithops` Serve orchestrates both types of resources: (1) cloud resources for non-sensitive images; and (2) edge resources for sensitive images, thereby improving cost efficiency while enabling privacy-preserving inference for sensitive data.

3.1.1 Business story

The `OffSampleAI` service for off-sample mass spectrometry image recognition historically faced highly unpredictable demand, resulting in wasted cloud resources during idle periods and slow processing during traffic spikes due to its always-on, reactive scaling solution built upon AWS ECS. Moreover, the system lacked the ability to securely handle sensitive image data. To overcome these challenges, we redesigned the service with `Lithops` Serve, using **serverless functions** for cost-efficient, elastic scaling and **confidential containers at the edge** for privacy-preserving processing.

This novel CloudSkin solution **eliminates idle infrastructure costs, accelerates response times under fluctuating workloads, and enables privacy-preserving handling of sensitive data**, yielding both operational savings and expanded applicability for `METASPACE` cutting-edge research platform.

To highlight the impact of `Lithops` Serve and CloudSkin on this use case, consider the following key results:

- `Lithops` Serve achieves job completion times two orders of magnitude lower than competitors at equivalent cost, demonstrating both high performance and efficient, cost-aware scaling for large-scale metabolomics workloads.
- The reconstructed images reveal minimal structural information across most setups, confirming that high privacy is maintained while quantifying the trade-off between inference latency and protected execution.

These numbers clearly demonstrate the improvements in efficiency, cost savings, and privacy-preserving capabilities introduced by the project in this domain.

3.1.2 Why this use case needs the compute continuum?

The compute continuum is essential for this use case due to the dynamic and unpredictable nature of the workload, where the dataset size in terms of number of images can vary +100 times as shown in Fig. 15.

As discussed in D5.3, the service experiences fluctuating workloads throughout the day, with periods of high activity followed by long idle times. The workload was analyzed to anticipate request bursts and enable proactive scaling. Several powerful models were evaluated, including LSTMs and Time Series Foundation Models; however, the workload proved to be highly spiky and erratic, as illustrated in Fig. 16. Due to the lack of recurring patterns, proactive scaling is not feasible.

This high workload variability requires an online scaling solution that can, given the size of a job in terms of requests, provision **the optimal number of compute units to minimize job latency while respecting a cost ceiling**. This is not achievable with AWS ECS, which lacks support for cost-based SLOs and does not allow explicit budget enforcement when provisioning extra instances. To address this challenge, we leverage AWS Lambda to execute inferences at scale, augmented with mechanisms

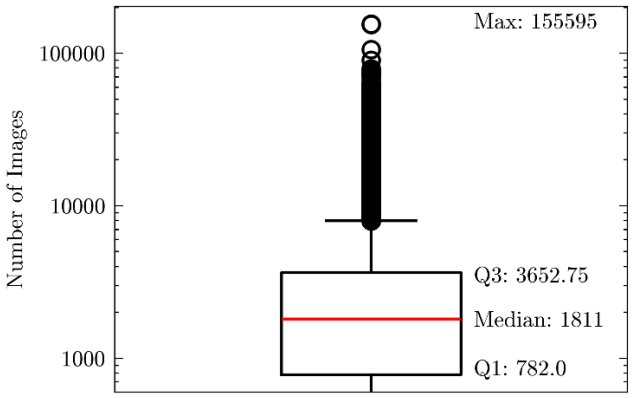


Figure 15: Distribution of dataset sizes in terms of number of images in 2023.

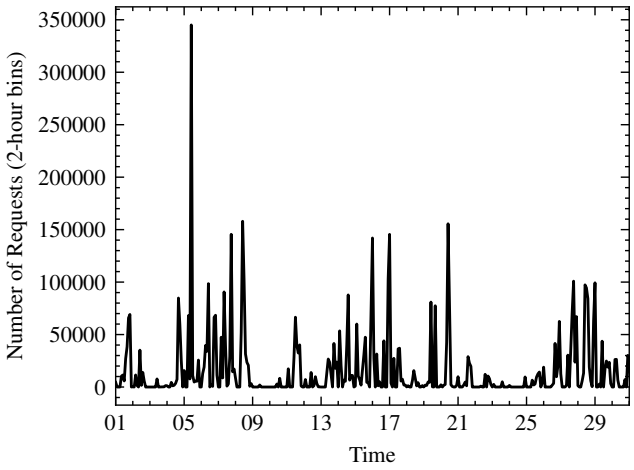


Figure 16: Workload during March 2024 using 2-hour bins.

to operate under strict cost constraints. Thanks to its rapid auto-scaling and inherent cost efficiency, serverless functions form the backbone of `Lithops` Serve.

Unfortunately, commercial serverless cloud platforms such as Lambda do not currently support confidential computing with Trusted Execution Environments (TEEs). To overcome this limitation, this use case combines cost-efficient serverless cloud functions with on-premises edge resources. This combination enables the following two capabilities:

- **Cost-efficiency:** Optimizes job latency while ensuring that the cost per request (CPR) remains within the target CPR (Challenge CH1).
- **Confidential execution:** For datasets containing sensitive images, e.g., from private companies such as AstraZeneca, the system leverages secure enclaves on the edge Kubernetes cluster to perform image classification, keeping the data encrypted and confidential, even from the host infrastructure (Challenge CH2).

3.2 Cloud-Edge continuum infrastructure for the metabolomics use case

3.2.1 CloudSkin platform

To address challenges **CH1** and **CH2**, the metabolomics use case leverages multiple components of the CloudSkin platform. At its core, `Lithops` **Serve** orchestrates serverless functions in the cloud to provide cost-efficient, low-latency processing of non-sensitive images (**CH1**), while **SCONE** enables

privacy-preserving inference of sensitive images within confidential containers on on-premises edge clusters (**CH2**). In both cases, the **Learning Plane** is used to perform intelligent, cost-driven resource provisioning, ensuring that the number of compute units is dynamically adjusted to meet latency and budget targets, as shown in Fig. 17.

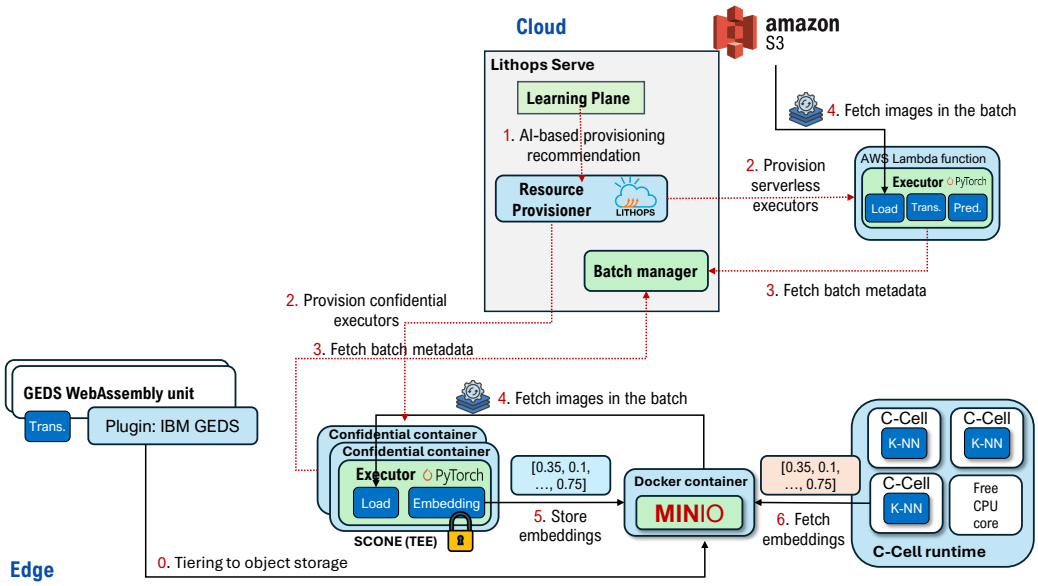


Figure 17: CloudSkin platform for the Metabolomics use case.

In addition, **C-Cells** are employed to perform K-NN similarity searches, allowing us to avoid fully TEE-shielded inference while still maintaining privacy for the sensitive images. While performing full inference entirely within the TEE is technically feasible, we observed that the latency overhead is substantial, often exceeding 5× that of unprotected inference. To mitigate this, we adopted a partially TEE-shielded approach: only the embedding or encoding phase of the model was run inside the TEE, keeping the first layers of the model private. The resulting embeddings are then processed by C-Cells executing an OpenMP job that can elastically scale-up to the number of available vCPU cores to perform the "On-sample" or "Off-sample" binary classification via K-NN similarity search. By elastically scaling up we can significantly improve image processing throughput while maintaining confidentiality of the sensitive data, and utilizing cloud-edge resources efficiently.

Finally, **GEDS-based WebAssembly Units** were employed to preprocess sensitive images prior to encryption and upload to object storage (MinIO). Dataset owners are expected to use a trusted, local GEDS client to securely transfer sensitive images to the edge cluster, ensuring confidentiality throughout the ingestion process.

3.2.2 Challenge CH1: Cost-efficiency with serverless cloud functions

To address **Challenge CH1**, we leveraged two key components of the CloudSkin platform: `Lithops` Serve, configured to use exclusively AWS Lambda functions, and the Learning Plane, which enables cost-driven resource provisioning. This corresponds to the **top pathway** illustrated in Fig. 17.

Integrated into the `METASPACE` platform, we expose a RESTful `POST` endpoint that is invoked by the preceding stage of the annotation pipeline to enqueue a new inference job. Once a running inference job completes, the **Job Manager** (the gray box in Fig. 17) pops the next job from the queue and initiates its execution. Among its responsibilities, the **Job Manager** monitors the progress and status of active jobs, ❶ determines the required degree of parallelism expressed as the number of Lambda functions to fulfill the target SLO by consulting the Learning Plane, and ❷ instructs the **Resource Provisioner** to launch the corresponding number of **Executors** or workers for the job.

Each **Executor** runs in an isolated Lambda instance and processes its assigned batches sequentially. For each batch, ❸ an **Executor** obtains the batch metadata from the **Batch Manager** in first place. This metadata includes a unique batch identifier (UUID) and the **Amazon S3 URIs** of the input images comprising the batch (e.g., `s3://METASPACE/JobXXX/myimage.jpg`). The **Batch Manager** is responsible for partitioning jobs into fixed-size batches, monitoring their execution, and reassigning failed batches to other workers in order to guarantee **exactly-once** processing semantics. A dedicated **Batch Manager** instance is instantiated for each inference job.

Upon receiving the batch metadata, ❹ each **Executor** downloads the corresponding images from Amazon S3 in parallel and processes the batch. Once processing is complete, the **Executor** stores the classification results back to S3 and notifies both the **Batch Manager** and the **Job Manager** of the batch completion. This process continues until all batches have been processed.

Cost-driven resource provisioner. As described in D5.3, the **Learning Plane** was used to train two regression models: one to predict the aggregated inference latency $T(n, r)$, and the other to estimate the total cost $C(n, r)$ required to process a job of r metabolomics images using n serverless executors. Leveraging these predictions, we implemented a cost-driven resource provisioner that chooses the optimal number of executors to minimize overall job completion time while remaining within a cost budget.

More formally, given a batch job with r inference requests, the **objective** of the Learning Plane is to choose the number of workers n that maximize performance under a predefined **cost-per-request** (`CPR`) restriction. We adopted this cost-based SLO because the `METASPACE` DevOps team required us a simple and intuitive metric to constrain the inference cost of each image classification. At the same time, this SLO grants `Lithops` Serve the flexibility to scale out to the number of serverless executors that minimizes job latency under a variable, per-job cost budget.

The result of this decision is the maximization of cost-effectiveness accompanied with cost SLO compliance in a single serving system. Observe that $T(n, r) = \sum_{i=1}^{n} T_i(r_i)$, where $T_i(r_i)$ is the latency contributed by executor i in processing r_i requests. Note that $\sum_{i=1}^{n} r_i = r$. In these terms, the problem can be formulated as follows:

$$\underset{n}{\text{minimize}} \quad T(n, r) \tag{7}$$

$$\text{subject to} \quad C(n, r) \leq \texttt{CPR} \cdot r \tag{8}$$

$$n, r \in \mathbb{N}, n < \min\left(N_{\max}, \left\lceil \frac{r}{b} \right\rceil\right) \tag{9}$$

where the additional constraints are: (8) the monetary cost $C(n, r)$ of the inference job remains within the user-defined cost SLO; and (9) at least one worker is provisioned and their total number does not exceed the minimum of the maximum concurrency limit ($N_{\max}$) set by the cloud provider and the allocation of one batch per worker ($\lceil \frac{r}{b} \rceil$). The key point here is that optimization problem (7) defines a compact search space, where the only parameter of interest is the number of workers. This gives the Learning Plane the ability to make fast scaling decisions on the order of a few milliseconds. Actually, the optimal number of executors, $\widehat{n}$, is determined by performing a binary search over the interval $1 \ldots \min(W_{\max}, \lceil r/b \rceil)$ and evaluating the two regression functions, $T(n, r)$ and $C(n, r)$, as defined in D5.3.

As shown in Algorithm 1, when the `CPR` is set to ∞, the Learning Plane provisions the maximum number of executors allowed by either the number of batches or the system concurrency—referred to as "**Latency-optimized**". Otherwise, it chooses the optimal number of executors that minimizes job latency while remaining within the per-job cost budget, which we call "**Cost-optimized**". Either way, this approach enables efficient, cost-aware scaling of variable inference workloads by turning predictive insights into actionable resource optimization.

3.2.3 Challenge CH2: Privacy-preserving inference on on-premises edge cluster

To address **Challenge CH2**, commercial FaaS platforms such as AWS Lambda currently lack support for confidential computing with TEEs. As a result, we developed an alternative solution leveraging

Algorithm 1 Cost-Constrained Executor Selection

Require: Number of requests r, batch size b, maximum workers $N_{\max}$ (1,000 concurrent function in AWS Lambda)
Ensure: Optimal number of executors $\widehat{n}$

$n_{\min} \leftarrow 1$
$n_{\max} \leftarrow \min\left(N_{\max}, \left\lceil \frac{r}{b} \right\rceil\right)$
$\widehat{n} \leftarrow n_{\min}$
while $n_{\min} \leq n_{\max}$ **do**
  $n \leftarrow \left\lfloor \frac{n_{\min}+n_{\max}}{2} \right\rfloor$
  $T \leftarrow T(n, r)$ {Predicted latency}
  $C \leftarrow C(n, r)$ {Predicted cost}
  if $C \leq \texttt{CPR} \cdot r$ **then**
    $\widehat{n} \leftarrow n$ {Feasible solution}
    $n_{\min} \leftarrow n + 1$ {Try more parallelism}
  else
    $n_{\max} \leftarrow n - 1$ {Reduce cost}
  end if
end while
return $\widehat{n}$

an on-premises edge cluster. As illustrated in the **bottom pathway** of Fig. 17, the solution involves the following steps:

- ⓿ A set of **GEDS-based WebAssembly Units** preprocesses sensitive images and uploads them encrypted to the MinIO object storage deployment on the `METASPACE` on-premises edge cluster, keeping confidentiality throughout the ingestion process.
- ❶ Once preprocessing completes, the job is `POST`ed to the `Lithops` Serve REST endpoint in AWS, together with URIs referencing the encrypted images stored in MinIO. `Lithops` Serve consults the Learning Plane to compute the required level of parallelism under a "relaxed" latency SLO and ❷ provisions the corresponding pool of **confidential Executors**. These executors run inside Intel SGX enclaves via **SCONE**, guaranteeing end-to-end data confidentiality.
- ❸ Each **confidential Executor** iteratively retrieves the metadata for its assigned batches and ❹ downloads the corresponding encrypted images from MinIO for processing.

  Instead of performing fully TEE-shielded inference, which is slow, each **confidential Executor** generates embeddings by running only the first layers of the model inside the TEE. These layers are chosen to prevent reconstruction of the original images, ensuring privacy while significantly speeding up confidential image classification. ❺ Public embeddings are stored back to MinIO.
- ❻ Finally, the distributed **C-Cells** runtime watches MinIO for new embeddings and classifies the associated images through a K-NN similarity search against representative "On-sample" and "Off-sample" embeddings. This approach replaces the forward pass through the model's final layers, allowing classification without exposing them. The search is executed by an OpenMP-based pool of parallel C-Cells that elastically adapts to available resources.

Confidential Executors provisioning. As mentioned in ❶, the allocation of **confidential Executors** is guided by a relaxed latency SLO. Analogous to the serverless model presented in Section 3.2.2, the job completion time here is a function of initialization and inference latencies for r requests across n confidential executors. We define the model components as following:

- **Initialization latency** L_{init}: The total time required to initialize n executors; this value increases as the number of executors grows. This is commonly denoted as *cold start* latency.

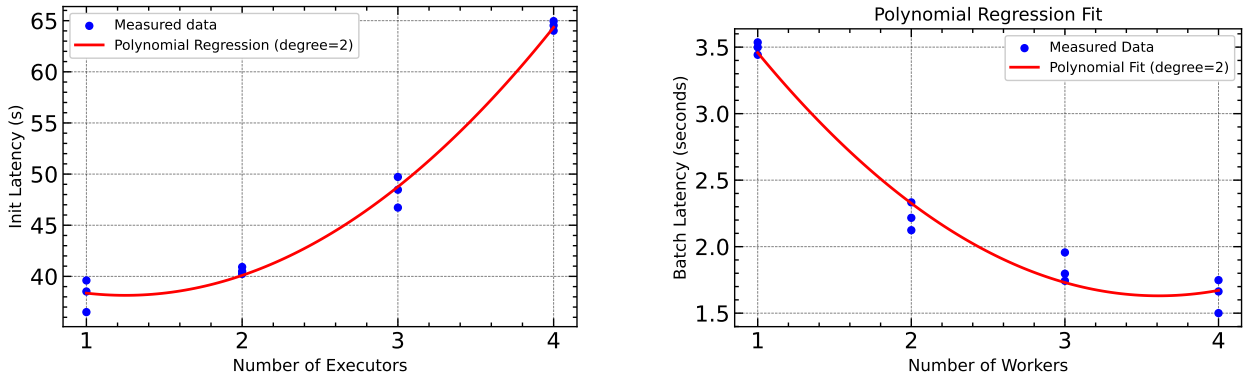
- **Batch latency** L_{batch}: The time required to process a batch by n executors; which decreases as the number of executors grows
- **Execution latency** L_{exec}: The total time required to process all batches after initialization. This can be derived from the batch latency model L_{batch} and the number of batches.

The first two components (L_{init} and L_{batch}) can be effectively modeled with polynomial fits. Therefore, each model would be equivalent to:

$$\hat{L}_{\text{init}}(n) = \text{Poly}_{\text{degree}=d}(n) = \beta_0 + \beta_1 n + \beta_2 n^2 + \cdots + \beta_d n^d, \tag{10}$$

$$\hat{L}_{\text{batch}}(n) = \text{Poly}_{\text{degree}=d}(n) = \gamma_0 + \gamma_1 n + \gamma_2 n^2 + \cdots + \gamma_d n^d, \tag{11}$$

where $\beta_0, \ldots, \beta_d$ and $\gamma_0, \ldots, \gamma_d$ are the polynomial coefficients learned from the collected data. We depict the results of the fitting in Fig. 18. The initialization latency fit achieves a coefficient of determination of $R^2 = 0.991$, while the batch latency fit achieves $R^2 = 0.976$, demonstrating that the polynomial models accurately capture the observed latency behavior.



(a) Init latency polynomial fit.

(b) Batch latency polynomial fit.

Figure 18: Polynomial regression fits for initialization and batch latency.

Based on the batch latency, the execution latency is then:

$$\hat{L}_{\text{exec}}(n, r) = \frac{r}{b} \cdot \hat{L}_{\text{batch}}(n), \tag{12}$$

with r being the total number of requests (images) and b the batch size. The predicted **job completion time** (JCT), $\hat{T}(n, r)$, becomes a sum of initialization and execution latencies:

$$\hat{T}(n, r) = \hat{L}_{\text{init}}(n) + \hat{L}_{\text{exec}}(n, r) = \hat{L}_{\text{init}}(n) + \frac{r}{b} \cdot \hat{L}_{\text{batch}}(n) \tag{13}$$

Given a job completion time SLO, S, and a number of input images r, the optimal number of confidential executors can be found via linear search:

1. For $n = 1, 2, \ldots, N_{\max}$, compute the predicted total latency $\hat{T}(n, r)$.
2. Select the smallest n such that $\hat{T}(n, r) \leq S$.
3. If no n satisfies the SLO, select $n = N_{\max}$ (being $N_{\max}$ the maximum number of available executors in the on-premises cluster).

It is worth noting that the executor selection algorithm is agnostic to N_{max}: stricter SLOs or larger workloads simply increase the ceiling, and the system automatically chooses the optimal number of executors to meet latency targets within the resource limitations.

This method guarantees that the workload is served within the SLO yet minimizing the number of provisioned executors, ensuring cost-effectiveness and efficient resource utilization.

Performance-security trade-off. A powerful way to protect inference is using TEEs such as `Intel SGX` or `ARM TrustZone`. TEEs provide **hardware-isolated enclaves** where code and data run shielded from the rest of the system. When a model executes inside a TEE, both the model parameters and user inputs remain confidential, even if the OS or hypervisor is compromised. This *confidential computing* approach enables untrusted cloud and edge servers to perform inference on sensitive input without exposing it to system administrators or attackers. That is, the sensitive metabolomics images can be decrypted, processed entirely within the enclave, and only the final prediction is revealed. TEEs thus offer a practical solution to the privacy challenges of ML-as-a-Service.

Algorithm 1 Ginver training in the white-box setting

function TRAIN($f_{\theta_A}, \mathcal{F}, \lambda, \epsilon, T, k$)
  /*f_{θ_A} is the victim's model*/
  /*$\mathcal{F}$ is a set of intermediate features*/
  /*λ is the equilibrium coefficient in Equation 4*/
  /*ϵ is the learning rate*/
  /*T is the number of iterations*/
  /*k is BatchSize*/
  /*$\theta_{\mathcal{G}}$ represent the parameter of $\mathcal{G}$*/

  initialize $\mathcal{G}$, $t \leftarrow 0$
  while t < T **do**
    Randomly sample $f_{\theta_A}(x_1), f_{\theta_A}(x_2), \ldots, f_{\theta_A}(x_k)$ from $\mathcal{F}$
    $L = \frac{1}{k} \sum_{i=1}^{k} \| f_{\theta_A}\left(\mathcal{G}\left(f_{\theta_A}\left(x_i\right)\right)\right) - f_{\theta_A}\left(x_i\right) \|_2$
    $L = L + \frac{1}{k} \sum_{i=1}^{k} TV\left(\mathcal{G}\left(f_{\theta_A}\left(x_i\right)\right)\right)$
    $\theta_{\mathcal{G}} = \theta_{\mathcal{G}} - \epsilon * \frac{\partial L}{\partial \theta_{\mathcal{G}}}$
    $t = t + 1$
  end while
return $\mathcal{G}$
end function

Figure 19: Inversion model training.

However, this added security comes at a cost: executing a model inside a TEE typically incurs a substantial latency overhead due to encryption and decryption operations, enclave context switches, and constrained enclave memory. This performance penalty is further exacerbated when compared to inference on hardware accelerators such as GPUs, where the gap between native execution and TEE-based execution can be particularly large. Therefore, while TEEs provide strong confidentiality guarantees, achieving an effective balance between privacy and performance remains a key challenge for latency-sensitive and compute-intensive applications.

For this reason, we chose to execute only the initial layers of the modified **RestNet50** model inside the TEE, using them to encode sensitive images into vector embeddings. These embeddings can then be treated as public representations, enabling efficient K-NN similarity search to identify the closest matches and perform image classification without exposing the full model or the raw input data.

As with related approaches such as **DarkneTZ** [7], protecting only a subset of the model layers

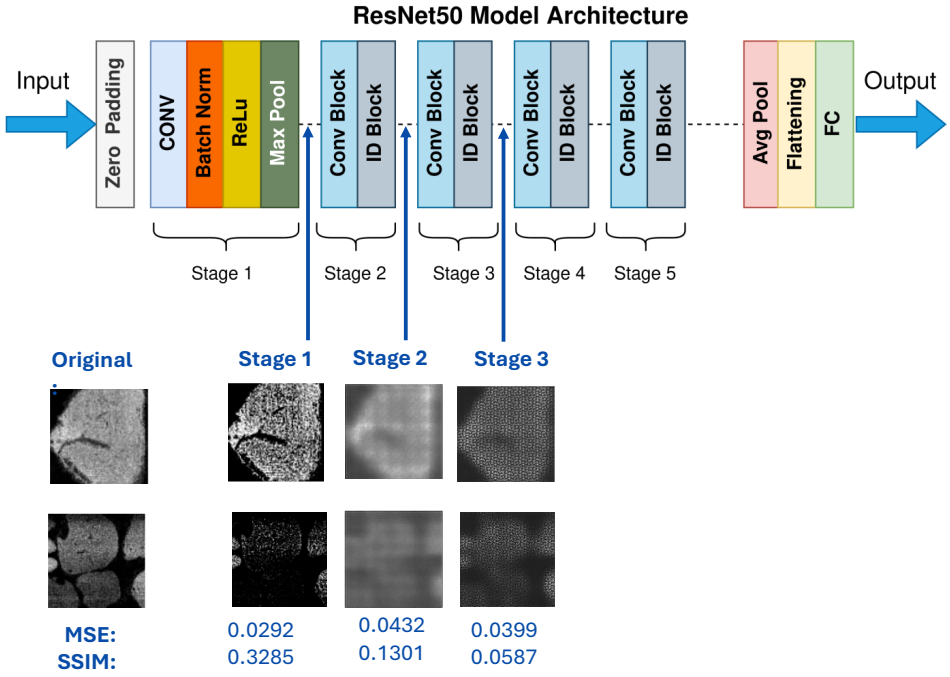


Figure 20: Reconstruction attack feasibility vs. number of layers protected in the TEE.

introduces a fundamental trade-off between performance and security. Public representations, either be exposed intermediate embeddings or unprotected model layers, may leak sufficient information to enable **partial reconstruction** of sensitive metabolomics images. Executing fewer layers inside the TEE improves inference latency but weakens confidentiality guarantees, whereas protecting more layers strengthens privacy at the cost of higher execution overhead. Balancing this trade-off is thus a central design challenge in privacy-preserving inference systems.

In particular, we evaluated the susceptibility of the modified `ResNet-50` model to **reconstruction attacks** under a **white-box** scenario, in which the adversary cannot directly access the internal model parameters but can query the protected portions as needed to train an **inversion model** $\mathcal{G}$ [8]. More specifically, we refactored each residual block of the `ResNet-50` model into its constituent operations: convolutions, batch normalization, ReLU activations, and skip connections, enabling the model to be partitioned at arbitrary layer boundaries and supporting precise, systematic evaluation of both shallow and deep reconstruction attacks. Then, for each chosen layer, we trained its corresponding **inversion model** $\mathcal{G}$ following the same approach as in [8]. Pseudo-code for training the inversion model is presented in Fig. 19.

To evaluate the effectiveness of reconstruction attacks, we used **MSE** (Mean Squared Error) and **SSIM** (Structural Similarity Index Measure). More technically, MSE is defined as the average squared difference between corresponding pixels of a ground-truth image X and a reconstructed image Y:

$$\mathrm{MSE}(X,Y) = \frac{1}{N}\sum_{i=1}^{N}(X_i - Y_i)^2,$$

where N is the number of pixels in the image. SSIM compares local patterns of pixel intensities, taking into account luminance, contrast, and structural information. In one common form, for images X, Y with local means μ_X, μ_Y, variances σ_X^2, σ_Y^2 and covariance σ_{XY}, SSIM is given by

$$\mathrm{SSIM}(X,Y) = \frac{(2\mu_X\mu_Y + C_1)(2\sigma_{XY} + C_2)}{(\mu_X^2 + \mu_Y^2 + C_1)\,(\sigma_X^2 + \sigma_Y^2 + C_2)},$$

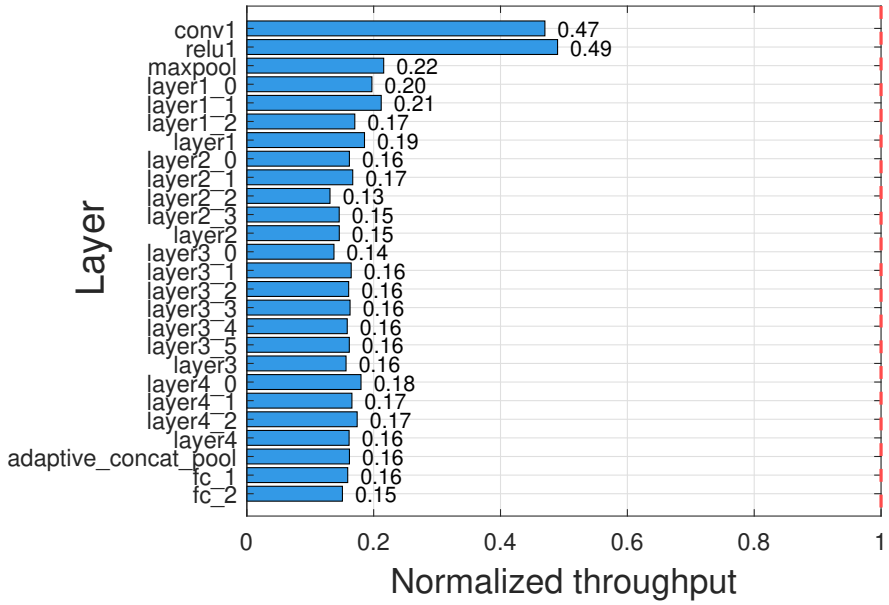


Figure 21: Reconstruction attack feasibility vs. number of layers protected in the TEE.

where C_1, C_2 are small stabilization constants. A higher SSIM (closer to 1) indicates more similar structure.

Despite their simplicity, **MSE** and **SSIM** have limitations in modeling perceptual similarity. MSE treats all pixel differences equally, so minor distortions that are visually insignificant can still produce large MSE values. SSIM is more aligned with human perception than MSE, but it still relies on local low-level statistics. In Fig. 20, we show how the reconstruction of images worsens as an increasingly larger number of layers are placed inside the TEE. As illustrated in this figure, reconstructed images derived from the **Stage 3**-vector embeddings reveal minimal information.

However, performance decreases as more layers are placed inside the TEE, as illustrated in Fig. 21, which plots throughput, measured in vector embeddings per second, normalized to plain execution outside the TEE. As illustrated, throughput drops to only $\sim$ 15% of the unprotected execution rate. Together with Fig. 20, this clearly outlines the inherent **trade-off between performance and privacy**.

For our **prototype implementation**, we generated vector **embeddings** at `layer2_2`, as this layer offers a well-balanced trade-off between performance efficiency and information leakage.

3.2.4 Cloud-Edge hardware

The `Lithops` Serve orchestrator (**Job Manager**, **Resource Provisioner**, etc.) runs on an AWS EC2 `t2.micro` instance (1 vCPU, 1 GB RAM). Executor instances are implemented as AWS Lambda functions, each configured with 2 vCPUs and 3 GB of RAM. The Lambda configuration was optimized via Bayesian optimization to maximize throughput per dollar. Input images and results are stored in AWS S3. The same EC2 instance also hosts Prometheus, Pushgateway, and Grafana for monitoring and metrics collection.

Confidential jobs are executed on a Kubernetes node hosted on a machine with 16 Intel SGX–enabled cores (Intel® Xeon® Platinum 8458P) and 64 GiB of RAM. Input images, intermediate tensors, and results are stored in a MinIO server running on the same machine as the Kubernetes node.

Table 7: Summary of use case-specific KPIs for metabolomics.

ucKPI	Description
ucKPI1:Latency	Curtail classification latency by a factor of 10 relative to the AWS ECS-based METASPACE solution.
ucKPI2:Throughput	Achieve a throughput (images/s) that is at least 10 times greater than the existing AWS ECS solution.
ucKPI3:Performance/$	Achieve at least 2x the performance per dollar compared to the current ECS implementation.
ucKPI4:Cost($)	Ensure that the total cost of processing each dataset does not exceed 3x the cost of the ECS implementation.
ucKPI5:SSIM	Average SSIM between original and reconstructed images lower than 0.2 indicates high privacy.
ucKPI6:Job completion time (JCT) for vector embeddings	$\geq$ 95% of privacy-preserving jobs produce vector embeddings within a 10-minute JCT SLO.

3.3 Experiments, KPIs, benchmarks and results

For benchmarking `Lithops` Serve, we used seven datasets. Each dataset is classified by size (small, medium, large) and annotated with its approximate number of images in thousands, e.g., **small.0.5k** (469 images), **medium.8k** (8,476 images), and **large.30k** (30,068 images).

KPIs. Table 7 lists the key KPIs for the metabolomics use case, targeting both efficiency and privacy. **ucKPI1:Latency**, **ucKPI2:Throughput**, and **ucKPI3:Performance/$** track operational performance: classification latency should drop 10× versus the AWS ECS-based `OffSampleAI` baseline, throughput should increase at least 10×, and performance per dollar should double. Furthermore, **ucKPI4:Cost ($)** ensures total processing remains under 3× the ECS cost, balancing speed and cost-effectiveness.

Privacy and reliability are captured by **ucKPI5:SSIM**, where values below 0.2 point out minimal information leakage, and **ucKPI6:Job Completion Time (JCT)**, which demands $\geq$ 95% of privacy-preserving jobs to produce vector embeddings within a 10-minute SLO. Together, these KPIs provide a clear framework for evaluating privacy-aware performance in the metabolomics workflow.

Experiment 1: `Lithops` **Serve against state-of-the-art inference systems (CH1).** In this experiment, we compare the performance of Lithops Serve against other state-of-the-art batch serving systems:

- **AWS Batch:** Configured using AWS Fargate containers (1 vCPU, 2 GB RAM). It employs an autoscaling strategy based on queue length, mapping each batch to a separate job/container (up to 100 concurrent instances).
- **AWS ECS:** Represents the baseline production solution used by `METASPACE`. It utilizes **AWS Fargate** containers (2 vCPU, 4 GB RAM) with step scaling based on CPU utilization thresholds, adding capacity when CPU usage exceeds 80%. It was configured to scale up to 50 containers.
- **EMBL:** The production deployment used by EMBL, functionally equivalent to ECS but limited to a maximum of 9 containers.
- **SageMaker AI:** Evaluated on AWS EC2 `ml.m5.large` instances (2 vCPUs, 4 GB RAM) using **asynchronous inference**. Autoscaling is configured to double the number of instances if CPU utilization exceeds 50% for one minute, and to scale down to zero when idle.
- **Ray:** running onAWS EC2 `t3.medium` nodes (2 vCPU, 4 GB RAM). It scales the worker pool (by up to 100% per scaling event) based on task and actor logical resource requests rather than application-level metrics.

Results. With `Lithops` Serve configured for a `CPR` SLO of $8 per million requests, it yields a latency of approximately 50 seconds, i.e., 25 to 40 times faster than competing systems, while maintaining a lower overall cost (**ucKPI3:Performance/$**). The only competitor with a lower cost was SageMaker, but this came at the expense of significantly higher latency, demonstrating `Lithops` Serve as the most **cost-efficient** option overall.

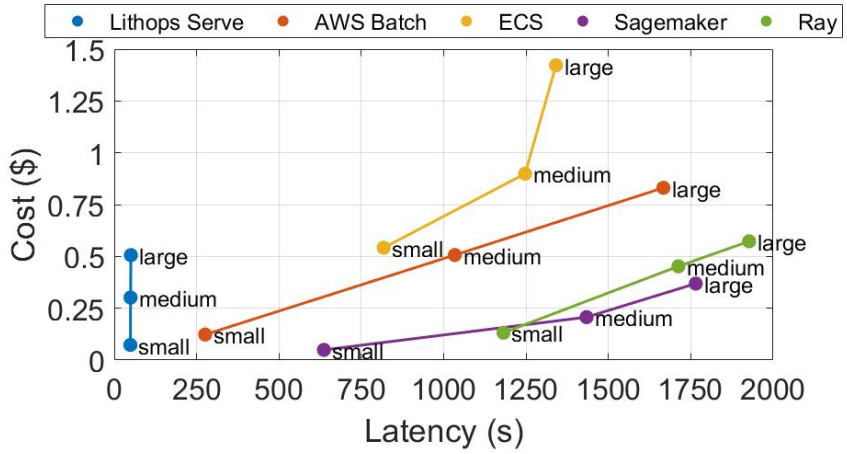
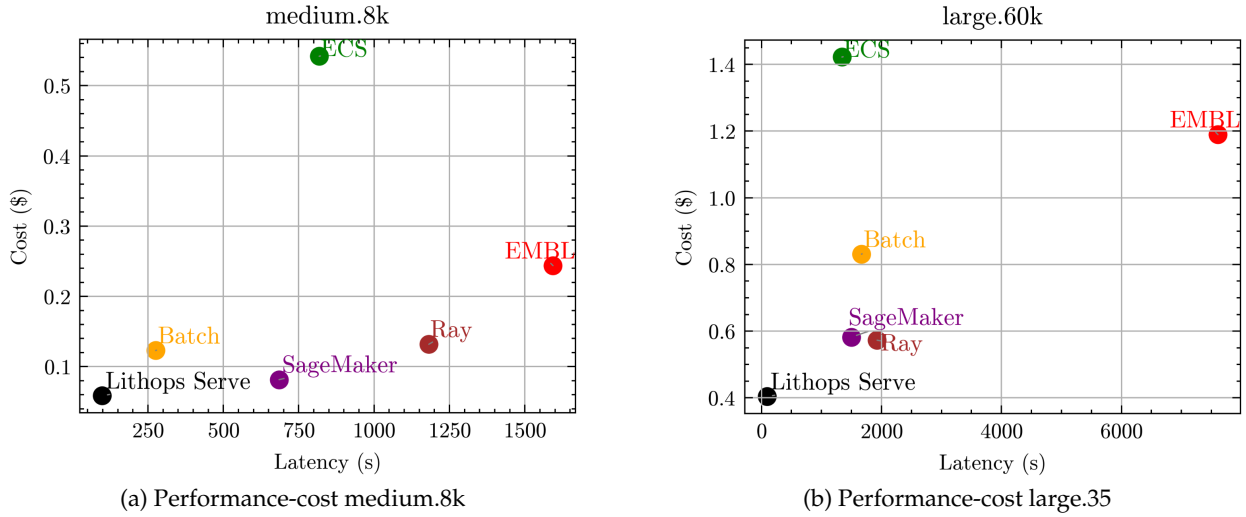


Figure 22: Experiment 1: Cost-optimized scaling against state-of-the-art

Fig. 23 focuses in the relation between latency and cost, showing that not only beats the previous `OffSampleAI` solution (EMBL), but also the rest. Results on two datasets highlight that the gains are especially striking for larger jobs, achieving up to 70× faster execution versus the baseline, well above the **ucKPI1:Latency** target of 10×. Similarly, **ucKPI2:Throughput** sees 70× improvement for `large.60k` and 20× for `medium.8k`. Performance per dollar surpasses expectations, reaching 260× for `large.60k` and 700× for `medium.8k`, far exceeding **ucKPI3:Performance/$**. Importantly, total cost remains below the baseline, fully satisfying **ucKPI4:Cost**.



(a) Performance-cost medium.8k (b) Performance-cost large.35

Figure 23: Experiment 1: Performance-cost against state-of-the-art.

Experiment 2: `Lithops` **Serve evaluation of cost-driven scaling (CH1).** This experiment evaluates the autoscaling capabilities of `Lithops` Serve under a cost-driven configuration using the `large.35k` metabolomics dataset. The platform is optimized for a `CPR` of $7 per million requests. The goal is to

assess how quickly the system can scale out to meet computational demand and how efficiently it utilizes CPU resources during initialization.

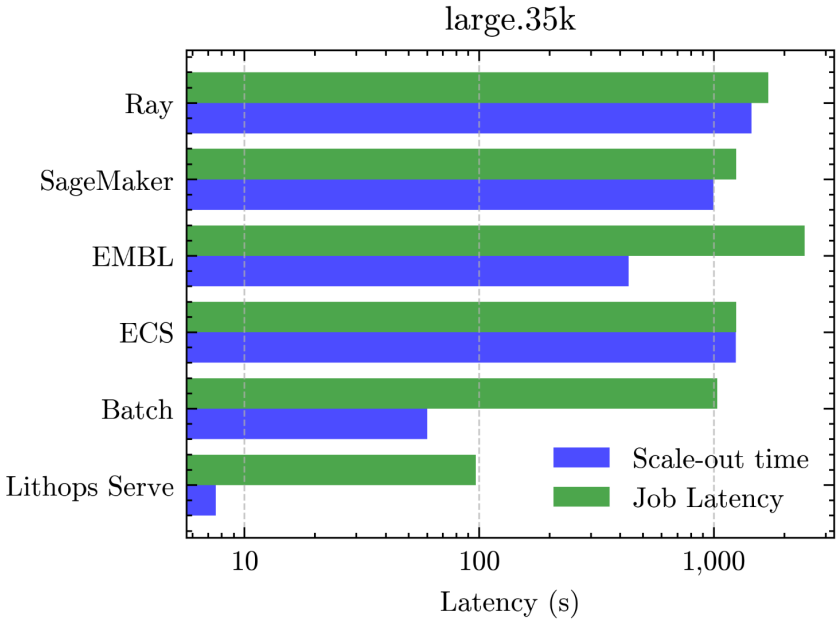


Figure 24: Experiment 2: Cost-driven autoscaling against state-of-the-art systems.

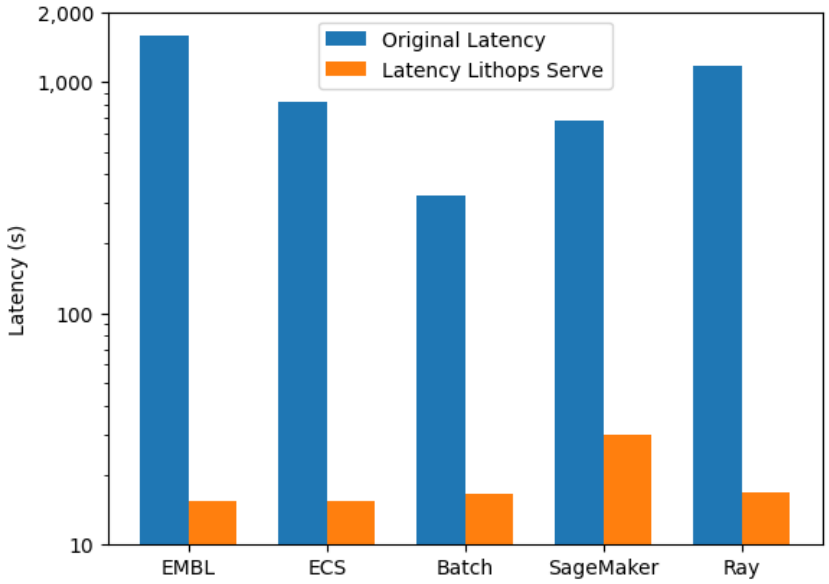


Figure 25: Experiment 2: Latency under the same budget.

Results. Fig. 24 shows that `Lithops` Serve rapidly provisions resources compared with state-of-the-art solutions. Table 8 quantifies these results, reporting both the scale-out time and the peak number of virtual CPUs allocated. As shown in the figure, `Lithops` Serve achieves scale-out times of only 7.5s (configuration Cost-optimized) and 11.2s (configuration Latency-optimized), while reaching peak CPU counts of 104 and 1,980 respectively, far exceeding the capabilities of competing systems such as AWS ECS, Ray, and SageMaker. This demonstrates both the speed and elasticity of `Lithops` Serve for large-scale workloads.

Additionally, Fig. 25 evaluates the `medium.8k` dataset by normalizing performance to the average `CPR` of each competing system. The results reveal that `Lithops` Serve achieves latencies two orders of magnitude lower than its competitors at equivalent cost levels. This highlights its efficiency not only in absolute performance but also in cost-aware scaling, making it particularly suitable for large-scale, cost-sensitive workloads in the metabolomics pipeline.

System	Scale-out time (s)	Peak vCPU count
Ray	1,447	100
AWS SageMaker	997	32
EMBL	433	18
AWS ECS	1,244	46
AWS Batch	66.8	100
`Lithops` Serve (Cost-optimized)	7.5	104
`Lithops` Serve (Latency-optimized: `CPR` $= \infty$)	11.2	1,980

Table 8: Scale-out time and peak vCPU count for different systems.

Experiment 3: Performance of GEDS-based WebAssembly Units for preprocessing images (CH2). This experiment measures the I/O performance of GEDS WebAssembly (Wasm) Units using a Rust application compiled to Wasm for image preprocessing. Originally, the Rust program read images from local storage, rescaled and normalized them, and wrote the outputs back to disk. In our setup, the write operations are replaced by GEDS Wasm Units, which transparently perform in-place image transformations, apply AES encryption, and tier encrypted images to MinIO.

To characterize I/O behavior under load, multiple GEDS Wasm Units (ranging from 10 to 100) are collocated on a single node. Each GEDS Wasm Unit processes a fixed number of images, leading to a total processed dataset size of 2.9 GB, 7.2 GB, 14.3 GB, and 29.7 GB for 10, 25, 50, and 100 modules, respectively. We measure disk write and network upload throughput during execution.

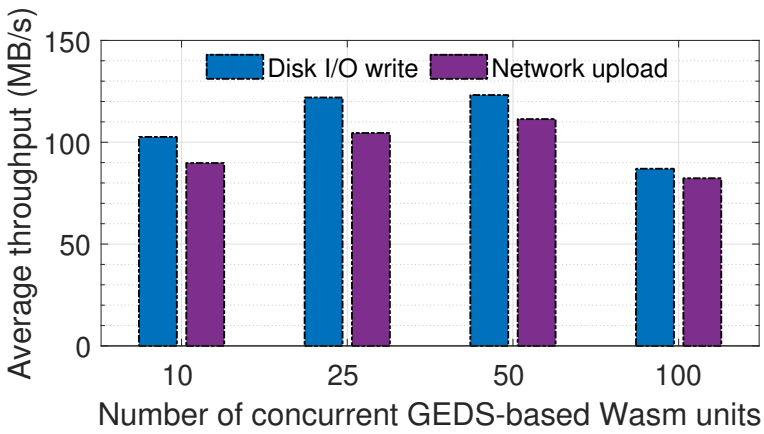


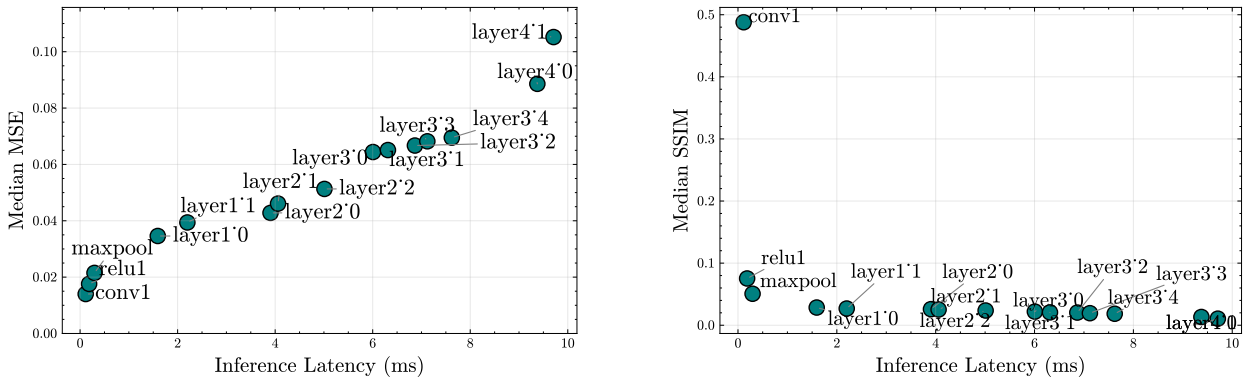
Figure 26: Experiment 3: Performance of GEDS-based WebAssembly Units for preprocessing images.

Results. Fig. 26 plots the results. Initially, all GEDS Wasm Units read their input images from disk, causing a small disk read spike. As GEDS Wasm Units complete pre-processing, output is written to GEDS Tier 0, producing disk write spikes, and subsequently offloaded to MinIO as needed, resulting in increased network throughput. Even collocated in the same physical machine (8 CPU cores), 50 GEDS Wasm Units are able to achieve an average throughput of 122 MB/s for disk writes and of 112 MB/s for network uploads.

The experiment highlights that increasing the number of concurrent units leads to higher total throughput but also longer processing times due to contention for CPU and I/O resources. These

results demonstrate the scalability and efficiency of GEDS Wasm Units in managing ephemeral, high-throughput data transformations while transparently integrating storage tiering and encryption.

Experiment 4: Evaluation of image reconstruction privacy and latency (CH2). This test assesses the trade-off between latency and reconstruction privacy when generating vector embeddings within a **confidential Executor** provisioned using `SCONE`. Two key metrics are analyzed: **Median MSE** and **Median SSIM** between original and reconstructed images. The **confidential Executor** ensures that embedding generation is performed in a secure, isolated environment, providing strong protection for both the model and the input data while allowing measurement of performance impacts.



(a) Median MSE of reconstruction attacks vs. number of layers protected in the TEE.

(b) Median SSIM of reconstruction attacks vs. number of layers protected in the TEE.

Figure 27: Experiment 4: Reconstruction attack feasibility as a function of the number of layers protected inside the TEE.

Results. Fig. 27a plots the Median MSE (y-axis) against inference latency (x-axis) for different stages of embeddings. Non-surprisingly, placing more layers within the **confidential Executor** reduces the fidelity of reconstructed images reflected by higher MSE, while simultaneously increasing the latency required to generate vector embeddings at those layers.

Fig. 27b shows the Median SSIM (y-axis) against inference latency (x-axis). SSIM values remain below the **ucKPI5:SSIM** target of 0.2, confirming that the reconstruction reveals minimal structural information and that high privacy is maintained across all evaluated configurations. As depicted in the figure, while performance decreases with higher TEE protection, the system effectively enforces privacy-preserving constraints, signaling the inherent trade-off between performance and privacy.

Altogether, the results demonstrate that the system meets the privacy KPI (**ucKPI5:SSIM**) while providing a quantifiable understanding of latency impacts due to TEE-protected processing.

Experiment 5: Job Completion Time under SLO Constraints (CH2). This experiment assesses the performance of `Lithops` Serve using the 2,740 jobs executed by the `METASPACE OffSampleAI` service in February 2024. The system is configured with a job completion time (JCT) SLO of 600 s to evaluate the effectiveness of the latency model in Eq. (13) at dynamically provisioning the appropriate number of **confidential Executors** based on job size, while minimizing SLO violations.

Results. As illustrated in Fig. 28, the system achieves a 97.08% SLO compliance rate, with the vast majority of jobs completing well below the 600s threshold. Although a small number of outliers are observed, some exceeding 4,000s, the dense concentration of jobs near the baseline demonstrates that the provisioning model consistently allocates the right number of **confidential Executors**. Overall, these results confirm the effectiveness in sustaining SLO compliance at scale (**ucKPI6:Job completion time (JCT) for vector embeddings**).

Experiment 6: Elastic C-Cell Scaling (CH2). In this experiment, we measure the performance of the last step of the pipeline, the OpenMP job that performs K-NN image similarity search with a pool

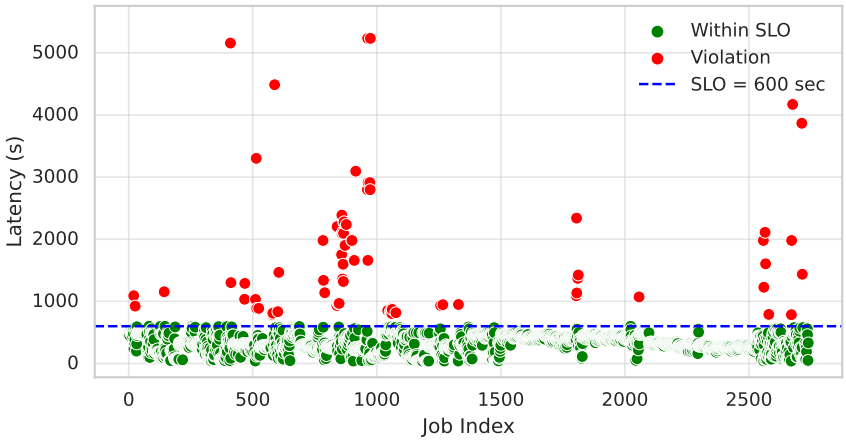


Figure 28: Experiment 5: SLO violations.

of **C-Cells** that can elastically scale up, or down, as requested by the control-plane. This experiment takes as an input a set of embeddings, pre-processed in an SGX enclave, and a set of images populated in MinIO storage. The goal is to perform a K-NN similarity search using a Rust program on each image in the dataset.

To achieve our goal, we implement an OpenMP application that orchestrates a dynamically-sized pool of C-Cells, where each C-Cell performs a K-NN search on one image. The OpenMP job leverages the elastic scaling feature of OpenMP jobs from WP4 in order to allocate more (or less) C-Cells to the running computation. This decision is made dynamically at runtime and does not require restarting the job. To emulate the situation where multiple of this pipelines are running in parallel, we first start two concurrent jobs (that we do not plot) that occupy CPU resources. Then, as resources free-up, the elastic OpenMP job automatically scales-up.

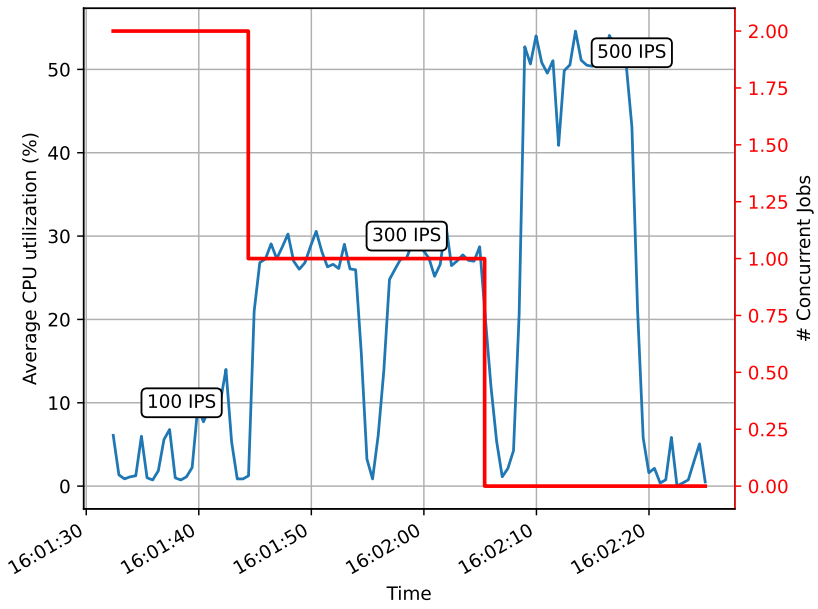


Figure 29: Experiment 6: Image processing throughput as we elastically scale-up to harvest idle vCPU cores. We label the peak throughput achieved in each phase in images-per-second (IPS).

Results. Fig. 29 summarizes our results. We plot with a red overlay the number of concurrent K-NN jobs. As jobs finish, and their CPU resources become available, the K-NN classification job scales-up,

as evidenced by the higher CPU utilization. We also overlay the throughput, in terms of images-per-second (IPS), achieved at each stage. Adding extra vCPU resources, i.e., threads to the OpenMP jobs, enables an improvement in throughput from 100 IPS to a peak 500 IPS. This illustrates the benefits of beginning processing as soon as any resources are available, and then elastically scaling up as the opportunities arise.

3.4 Demos

Demo 1: Cost-driven model serving with serverless functions. This demo showcases telemetry and monitoring capabilities for cost-optimized model serving using `Lithops` Serve deployed on AWS Lambda, addressing Challenge CH1.

Five dashboards have been implemented, each providing observability at a different level, from inference-specific metrics to detailed system performance.

The demo runs two jobs: one processing 2,000 images with a cost per million requests (`CPR`) set to $8, and another processing 15,000 images with `CPR` set to $6. The different `CPR` values showcase how cost can be adjusted according to the client requirements. The `Lithops` Serve API accepts the AWS credentials to download images from object storage (Amazon S3), along with a target `CPR` value, batch size ($b = 32$ images), and output location for saving results back to object storage. During job execution, the dashboards display real-time progress as well as time-lapse visualizations, providing a comprehensive view of system behavior and performance.

Inference Dashboard. The Inference Dashboard is dedicated to monitoring inference workloads and is composed of four main components. First, an execution summary reports the total number of requests, the configured batch size, and the number of active executors, as shown in Fig. 30.

Figure 30: Inference dashboard execution summary for Lithops Serve on AWS Lambda.

Second, the batch assignment timeline visualizes how batches are distributed over time across executors, enabling the analysis of scheduling behavior and load balancing (Fig. 31).

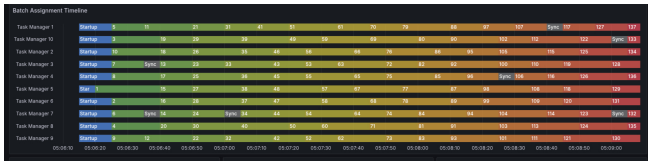


Figure 31: Inference Dashboard: batch assignment timeline.

Third, the dashboard exposes real-time inference metrics, including completed number of batches, running time, throughput, average batch latency, average number of batches and requests processed per executor, total execution cost, and throughput per dollar, as illustrated in Fig. 32.

Finally, progress and distribution metrics are presented as bar charts showing the number of batches processed by each executor, batch-level latency, and executor running time (Fig. 33).

General Dashboard. The General Dashboard provides a high-level summary of each job execution, including execution time, number of workers and processes, CPU utilization, user and system time, memory consumption, and network usage, as shown in Fig. 34.

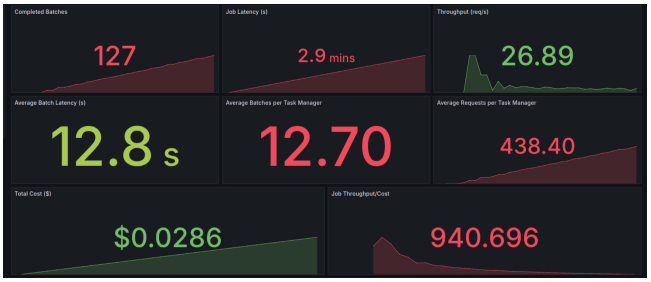


Figure 32: Inference Dashboard: real-time metrics.

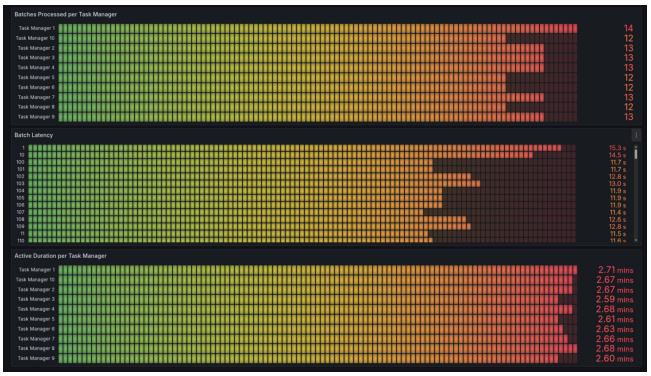


Figure 33: Inference Dashboard: executor progress and performance.

Figure 34: General Dashboard: job summary.

General Performance Dashboard. The General Performance Dashboard presents an aggregated view of system-wide performance across all workloads, highlighting CPU usage, disk I/O, memory consumption, and network activity (Fig. 35).

Figure 35: General Performance Dashboard: system-wide metrics.

Job Detailed Dashboard. The Job Detailed Dashboard offers a fine-grained view of resource utilization for individual jobs, enabling deeper analysis of CPU, disk, memory, and network behavior on a per-job basis, as illustrated in Fig. 36.

Figure 36: Job Detailed Dashboard: per-job metrics.

Executor Detailed Dashboard. The Executor Detailed Dashboard aggregates resource usage and performance metrics at the Executor level, allowing comparison of CPU, disk, memory, and network

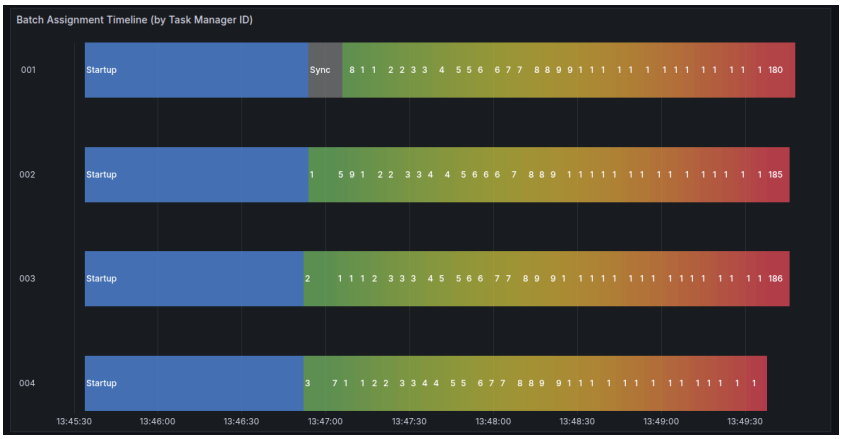
utilization across executors, as shown in Fig. 37.

Figure 37: Executor Detailed Dashboard: aggregated metrics.

Demo 2: Confidential Batch Serving In the same direction as the previous demo, this demo aims to evaluate system behavior under concurrent execution of two confidential jobs. The `small.2k` and `medium.8k` datasets are used, both subject to a latency SLO of 600 seconds. We observe that this SLO has been set less restrictively in this case due to the lack of elastic CPU capacity available in cloud-based serverless platforms. Consequently, the maximum number of **confidential Executors** has been limited by the available local Kubernetes resources: a practical limit of $N_{\max} = 4$ executors. We note that a pool of four executors was sufficient to meet the JCT SLO of 600 seconds, resulting in less than 5% of jobs exceeding it.

(a) Overview

(b) Metrics



(c) Batch assignment timeline

Figure 38: Inference dashboard for Lithops Serve with confidentiality.

Images are preprocessed using GEDS-based WebAssembly units and stored in encrypted form in MinIO. During the demo, the transformed image are retrieved and inference is executed up to `layer 2_2` to produce the vector embeddings for the K-NN classifier; after each batch, the resulting vector embeddings are written back to MinIO.

The **Inference Dashboard** shows the same metrics as in the first demo, now featuring Kubernetes with SCONE as the execution backend (Fig. 38).

4 Use case: Surgery

The National Center for Tumor Diseases (NCT) in Germany combines data scientists and surgeons to apply advanced analytics to surgery-related multimedia. Their primary requirement is to ingest video streams from endoscopic cameras reliably and perform real-time AI inference to assist surgeons during procedures. These video streams must also be durably stored for later batch analytics, such as AI model training. This use case is highly latency-sensitive, as delays in video analytics can directly impact surgical decision-making.

To address NCT's needs, in D2.3 we reported a Proof of Concept (PoC) that enables real-time AI video inference over streaming data. The PoC integrates Pravega [9, 10] as the backbone for durable and elastic video stream storage, combined with GStreamer [11, 12] for video ingestion and pipeline management. Containerized AI inference models run on top of this stack to process frames in real time, while Kubernetes orchestrates containerized services across the Cloud-Edge continuum. The system also includes telemetry and metrics collection via Prometheus and visualization through Grafana, enabling monitoring and operational insights. This architecture provides a robust foundation for real-time video analytics in latency-sensitive environments.

Next, we provide an extensive description of the advanced NCT-related challenges we addressed in CloudSkin for the second half of the project.

4.1 Overview

Work at the NCT aims to provide surgeons with AI-assisted video inference to improve the quality and safety of surgical procedures. In this project, NCT has contributed AI models to identify instruments, detect surgical phases, and perform liver segmentation tasks while operating. Video streams collected from endoscopic cameras must be processed at high frame rates ($\geq$ 30 FPS) and stored durably for later batch analytics and AI model training. In D2.3, we demonstrated via a PoC that achieving real-time AI video inference on a streaming data platform is a feasible approach.

However, with the initial PoC developed, more advanced requirements came into play. In particular, an overarching challenge for a surgical AI video platform is *resource optimization* across the compute continuum—from surgical edge nodes to central cloud facilities—while ensuring flexible Cloud-Edge *data management*. This challenge manifests in three key technical challenges:

- **Inefficient CPU and GPU Allocation for AI Models (CH1)**: Surgical AI models (instrument detection, phase recognition, liver segmentation) exhibit heterogeneous resource demands that need to be concurrently executed on a limited Edge infrastructure. We empirically found that naive CPU & GPU packing or aggressive time-slicing degrades frame rates, especially for segmentation tasks. Intelligent allocation strategies—based on profiling and bin-packing heuristics—are needed to maximize GPU utilization without violating latency Service Level Objectives (SLOs) in terms of FPS. This involves solving multi-dimensional resource allocation problems considering CPU cores, GPU memory, and model-specific constraints.

- **Need for Latency-aware Auto-Scaling of Streaming Storage (CH2)**: We observed that NCT surgery room utilization exhibits strong daily patterns, with workloads fluctuating significantly between peak and off-peak hours. This variability requires dynamically adapting the underlying streaming storage infrastructure to match demand. However, scaling up or down the number of segment store instances in a streaming service like Pravega introduces latency spikes during reconfiguration, which is highly undesirable for real-time video analytics where stable

ingestion and inference are critical. The core challenge lies in how to auto-scale the streaming storage infrastructure while minimizing the latency impact caused by instance scaling events.

- **Lack of Advanced Cloud-Edge Data Management (CH3)**: Video data ingested in real time at the Edge must eventually be moved to long-term storage in the Core/Cloud for batch analytics and AI training. Current streaming storage systems like Pravega simply offload video streams as large, opaque data objects, without exploiting the data path for additional value. This lack of programmability means missing opportunities for in-transit data management. The challenge is to design a new programmability model for data flowing from Edge to Cloud that allows users to deploy custom functions across the continuum. These functions should transparently add value, for example by implementing buffering strategies during connectivity issues or running AI models to annotate video chunks with surgical phases and instrument metadata, enabling smarter storage and faster retrieval for downstream analytics.

4.1.1 Business story

At the NCT, surgical precision and patient safety depends on uninterrupted AI-assisted video analytics. In the future, it is expected that surgeons, or CAS systems which surgeons may use, will depend on real-time inference from endoscopic video streams to, among other things, identify instruments, detect surgical phases, and guide critical decisions during procedures. Any latency spike or resource bottleneck can compromise this process, making infrastructure reliability and efficiency paramount.

However, hospitals face mounting complexity and cost in managing heterogeneous compute resources across operating rooms and central IT facilities. Workload patterns fluctuate significantly throughout the day, requiring dynamic adaptation of streaming storage and compute resources. At the same time, video data must be durably stored and enriched for downstream analytics and AI training, without disrupting real-time operations.

CloudSkin transforms this reality by introducing **intelligent resource allocation**, **predictive streaming elasticity**, and **programmable data flows** across the compute continuum. For NCT, this means:

- **Maximizing utilization of scarce edge resources (CH1):** Smart GPU allocation enables concurrent execution of heterogeneous AI models without over-provisioning hardware. This allows NCT to support more surgeries simultaneously, reducing idle capacity and operational costs.
- **Consistent real-time performance under dynamic workloads (CH2):** Predictive auto-scaling mitigates disruptive latency spikes during storage reconfigurations, ensuring stable ingestion and inference even during peak surgical hours. Surgeons gain confidence that AI insights remain timely and reliable.
- **Data managed as it moves, not after (CH3):** Programmable data management turns the data path into a value-added pipeline. Deploying data management functions like storage buffering to mask network disruptions or AI-powered annotation for tagging video data objects accelerates innovation and adds value to the flow of streaming data across the Cloud-Edge continuum.

By solving these challenges, CloudSkin positions NCT at the forefront of surgical AI, delivering a platform that not only meets clinical demands but also drives research excellence and efficiency.

4.1.2 Why this use case needs the compute continuum?

The NCT surgical workflow spans both Edge and Cloud, and its requirements cannot be met by isolated infrastructures:

- **Edge:** Real-time AI inference on surgical video streams demands ultra-low latency and guaranteed throughput. This requires efficient GPU allocation (CH1) and elastic streaming storage (CH2) to handle fluctuating workloads without compromising frame rates. Predictive scaling ensures stable ingestion even during peak operating room activity, avoiding disruptive latency spikes that could impact surgical decisions.

- **Cloud:** During a surgery, large volumes of video data must be offloaded to central IT facilities for batch analytics and AI model training. Advanced data management (CH3) ensures seamless movement of data across heterogeneous infrastructures, while adding value through buffering to mask network disruptions and AI-powered annotation pipelines that enrich video content for faster retrieval and research.

Without a compute continuum, hospitals would face two undesirable extremes: over-provisioning Edge resources (leading to high costs and inefficiency) or suffering performance degradation during peak workloads. The continuum enables dynamic orchestration of compute and storage resources, combined with policy-driven data flows, delivering both cost-efficiency and reliability. For NCT, this means uninterrupted real-time analytics during surgery and accelerated innovation in surgiomics through enriched, readily available datasets.

4.2 Cloud-Edge continuum infrastructure for the surgery use case

4.2.1 CloudSkin platform

The CloudSkin platform provides an integrated architecture to support real-time surgical video analytics across the compute continuum, combining Edge resources in operating rooms with centralized IT infrastructure for long-term storage and batch analytics. Its design addresses three critical challenges in the NCT scenario: GPU allocation for AI models, predictive streaming storage auto-scaling, and advanced Cloud-Edge data management (see Fig. 39).

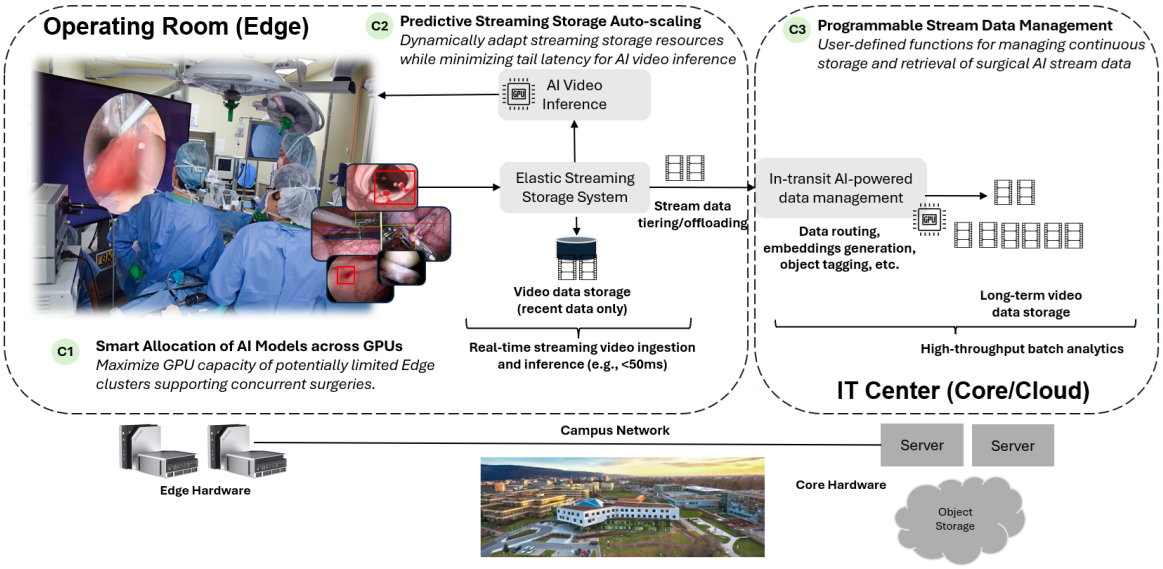


Figure 39: CloudSkin platform for the NCT use case.

At the edge, CloudSkin orchestrates AI inference pipelines for surgical video streams under strict latency constraints (*e.g.*, < 50 ms per frame). The platform leverages Kubernetes-based orchestration to deploy NCT AI models for instrument detection, phase recognition, and segmentation. To maximize GPU utilization in resource-constrained surgical clusters, CloudSkin implements smart allocation of AI Models across GPUs (CH1). This involves profiling model resource demands and applying placement strategies to optimize GPU sharing without violating frame-rate SLOs (*e.g.*, ≥ 30 FPS). Our experiments show that naive time-slicing degrades performance for heavy models like segmentation, while intelligent allocation policies maintain throughput and reduce idle GPU capacity.

Video streams ingested from surgical cameras are stored in Pravega, a tiered streaming storage system integrated into CloudSkin. Pravega ensures low-latency writes and durable retention by combining a write-ahead log with long-term object storage. To prevent latency spikes during workload fluctuations exhibited by NCT surgery room usage patterns, CloudSkin introduces predictive streaming storage auto-scaling (CH2) using Long Short-Term Memory (LSTM)-based forecasting. LSTM is a

popular type of recurrent neural network (RNN) architecture, designed to learn and retain long-term dependencies in sequential data. This mechanism anticipates and predicts operating room utilization patterns and proactively adjusts segment store instances, reducing disruptive scaling events and improving tail latency—the high-percentile response times (e.g., 95th or 99th percentile) that represent the slowest requests impacting user experience. This is critical for real-time AI video inference.

Beyond real-time analytics, CloudSkin enables programmable stream data management (CH3) for continuous storage and retrieval of surgical multimedia. The CloudSkin platform intercepts tiered storage operations and executes user-defined data management functions across the Cloud-Edge continuum. For example, during storage outages, buffering streamlets—data management functions executed on a chunk of tiered data—maintain uninterrupted ingestion, while annotation streamlets tag video chunks with surgical phases and instrument metadata for efficient retrieval. These operations occur transparently across Edge and Cloud infrastructures, adding value to the flow of data. Long-term video data is stored in object storage at the IT center, enabling high-throughput batch analytics and AI model training.

In summary, CloudSkin unifies elastic compute, predictive scaling, and programmable data flows to deliver a robust platform for latency-sensitive, AI-driven surgical workflows. It ensures that operating rooms benefit from real-time intelligence while central facilities handle long-term video storage and batch analytics, all under a cohesive Cloud-Edge architecture.

4.2.2 Cloud-Edge hardware

In Fig. 39, we provide a unified, global view of the CloudSkin platform for NCT. In practice, however, we have opted to evaluate each of the challenges independently, which leads to specific hardware. In the following, we describe the hardware platforms we used to evaluate our research contributions regarding the NCT challenges:

Smart CPU & GPU hardware allocation (CH1): The CPU & GPU allocation experiments were conducted on a Kubernetes-based cluster designed to emulate a surgical edge environment. The cluster consisted of four nodes interconnected via 1 GbE Ethernet: one control plane node and three worker nodes. The control plane hosted the Pravega segment store, while the worker nodes provided heterogeneous compute resources. Two of the worker nodes were equipped with 4× NVIDIA RTX A5000 GPUs each, paired with Intel Xeon Silver 4216 CPUs (64 logical cores), and 16 GB of RAM per node. The third node featured a GTX 1080 GPU and an Intel i7-7700K CPU (4 cores). Storage was backed by a Samsung NVMe SSD (2 TB, 3,200 MB/s read, 2,400 MB/s write) to ensure high throughput for video ingestion. This setup allowed experiments on GPU time-slicing, pod anti-affinity, and bin-packing strategies for NCT AI models (instrument detection, phase recognition, liver segmentation), focusing on maintaining ≥ 30 FPS under varying resource allocation policies.

Predictive streaming auto-scaling hardware (CH2): The predictive auto-scaling experiments were deployed on a Kubernetes cluster comprising seven nodes, each provisioned with 8–16 CPU cores, 16 GB of memory, and 80–100 GB of persistent storage. The cluster hosted a full Pravega deployment, including three Bookkeeper instances, three Zookeeper instances, and a horizontally scalable segment store. Long-term storage was integrated via MinIO, simulating object storage for tiered data management. The experimental environment was designed to mimic real-world conditions for latency-sensitive workloads, with video streams ingested through GStreamer pipelines and processed by AI inference models running in containerized pods.

Programmable stream data management hardware (CH3): The hardware used for evaluating CH3 is the same as for CH2. In this case, we deployed an additional Nexus instance (plus Redis for metadata) to perform data management operations between Pravega and MinIO. Note that one VM was equipped with an NVIDIA A16 GPU (16 GB VRAM) for executing AI-based streamlets (*e.g.*, NCT models for surgical annotation).

4.3 Experiments, KPIs and benchmarks

This section details the experiments conducted to validate CloudSkin in the Surgery use case, focusing on three core challenges: (CH1) smart CPU/GPU allocation for real-time AI inference, (CH2)

Table 9: Summary of use case-specific KPIs for Surgery.

ucKPI	Description
ucKPI1(KPI2) : Edge resource utilization	3× workload density (12 vs 4 streams) and better GPU utilization up to 50%.
ucKPI2(KPI3): Real-time edge processing	30FPS sustained for multiple surgical endoscopic videos in an Edge constrained infrastructure.
ucKPI3(KPI6) : Confidential TEE execution	Running a GStreamer pipeline in SCONE incurred a 3.41× slowdown, while pure Python inference in TEE added 14.96% overhead, proving feasibility.
ucKPI4(KPI11) : Low-latency streaming	Predictive auto-scaling improves by nearly 6× worst-case p90 latency compared to reactive approach, which is key for real time AI video inference.
ucKPI2(KPI14) : In-transit data management	Buffering masks long-term storage outages in Pravega; Semantic annotation adds 0.3s to 1.2s per PUT request to automatically add metadata to objects.

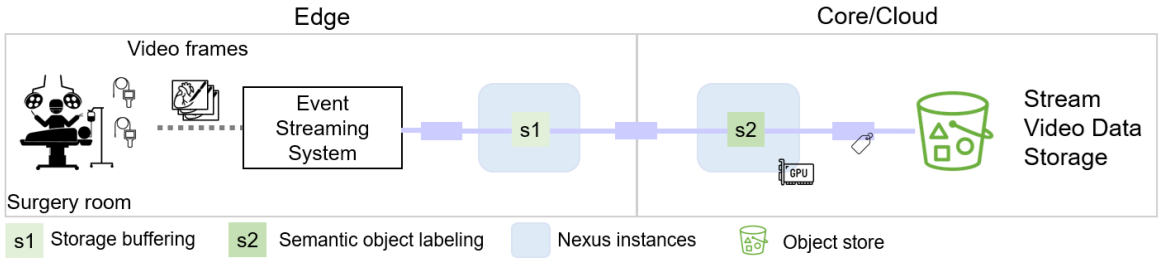


Figure 40: Surgiomics challenges addressed with `Nexus`.

predictive auto-scaling of streaming storage, and (CH3) programmable in-transit data management. An additional experiment evaluates confidential execution of AI inference.

Experiment 1 (CH1): Smart CPU & GPU Allocation

- *Objective*: Validate bin-packing strategies to maximize GPU utilization while maintaining frame rate SLOs (≥30 FPS).
- *Methodology:* Kubernetes-based edge cluster with NVIDIA RTX A5000 GPUs; workloads include instrument detection, phase recognition, and liver segmentation. Metrics collected via Prometheus/Grafana.
- *Benchmark:* Compare baseline (1 stream/GPU) vs bin-packing (3 streams/GPU).

Experiment 2 (CH2): Predictive Streaming Storage Auto-Scaling

- *Objective*: Reduce latency spikes during Pravega scaling events using LSTM-based predictive elasticity. Metrics collected via Prometheus/Grafana.
- *Methodology:* Replay 2-month NCT OR workload traces at 15× speed; compare reactive vs predictive scaling.
- *Benchmark:* Custom workload generator that auto-scales video pods according to NCT surgery room traces.

Experiment 3 (CH3): Programmable Data Management

- *Objective*: Enable buffering under storage outages and semantic annotation for enriched retrieval. Metrics collected via Prometheus/Grafana.

- *Methodology:* Nexus streamlets intercept Pravega → MinIO tiering; measure ingestion continuity and PUT latency.
- *Benchmark:* OpenMessaging Benchmark generating surgical images as events to a Pravega stream.

The KPI results for these experiments can be seen in Table 9.

4.4 Results

Experiment 1 (CH1): Smart CPU & GPU Allocation for Surgical AI Models A critical challenge for NCT's surgical AI video platform is achieving efficient CPU and GPU allocation across heterogeneous AI models (instrument detection, phase recognition, liver segmentation) on limited Edge infrastructure. Traditional one-stream-per-GPU approaches avoid contention but lead to significant underutilization ($<$ 20% GPU usage), preventing hospitals from supporting multiple concurrent surgeries with available hardware. To address Challenge CH1, we investigated a bin-packing optimization strategy that consolidates multiple streams onto shared GPUs via time-slicing while respecting multi-dimensional resource constraints: CPU cores, GPU memory, and model-specific frame rate Service Level Objectives (SLOs $\geq$ 30 FPS).

The key objectives of this experiment are:

- *Characterize CPU requirements*: Empirically determine the minimum CPU allocation needed to maintain $\geq$30 FPS for each NCT AI model through profiling-based analysis.
- *Validate bin-packing feasibility*: Demonstrate that GPU time-slicing enables consolidation of multiple heterogeneous streams without violating frame rate SLOs or degrading inference quality.
- *Quantify resource efficiency gains*: Compare baseline deployment (one stream per GPU) against intelligent bin-packing to measure improvements in GPU utilization and concurrent surgery capacity.

The experiments were conducted on a Kubernetes-based edge cluster emulating a surgical environment. The cluster comprised four nodes interconnected via 1 GbE Ethernet, with worker nodes equipped with 4$\times$ NVIDIA RTX A5000 GPUs, Intel Xeon Silver 4216 CPUs (64 logical cores), and 16 GB RAM. Storage was backed by Samsung NVMe SSDs (2 TB, 3,200 MB/s read) to ensure high throughput for video ingestion. Each GPU was exposed as a distinct Kubernetes resource (`nvidia.com/gpu-0`, `nvidia.com/gpu-1`, etc.) via NVIDIA GPU Operator, enabling fine-grained allocation. Time-slicing was configured to allow up to 6 concurrent pods per GPU. The NCT surgical video processing pipeline was deployed using Pravega for stream storage and GStreamer for video ingestion, with three AI model workloads: Instrument Detection (4 CPU cores), Phase Detection (4 CPU cores), and Liver Segmentation (8 CPU cores, higher computational demand due to semantic segmentation complexity).

Phase 1: CPU Profiling and Bin-Packing Scalability Before evaluating bin-packing strategies, we first characterized the CPU requirements for each workload type to ensure adequate resource allocation, then validated scalability under GPU time-slicing. Figure 41 presents results for Liver Segmentation, the most computationally demanding workload. The CPU profiling (left) reveals that this workload requires 8 CPU cores to achieve stable 30 FPS performance, reflecting the higher computational overhead of semantic segmentation models. GPU utilization ranges from 8–27.5% across CPU allocations, and GPU memory usage increases to $\approx$600–1175 MB, with power consumption reaching 108–150 watts under full load. The bin-packing scalability results (right) show that framerate remains acceptable up to 3 concurrent streams, beyond which slight degradation begins to occur. GPU utilization scales from $\approx$18% for 1 stream to $\approx$75% for 6 streams. GPU memory usage scales to $\approx$3500 MB for multi-stream configurations, with memory utilization reaching $\approx$14%. We found that packing the processing of too many streams onto one GPU concurrently leads to this degradation - spreading the processing of liver segmentation streams across multiple GPUs with anti-affinity resolves this, as can be seen in Figure 42.

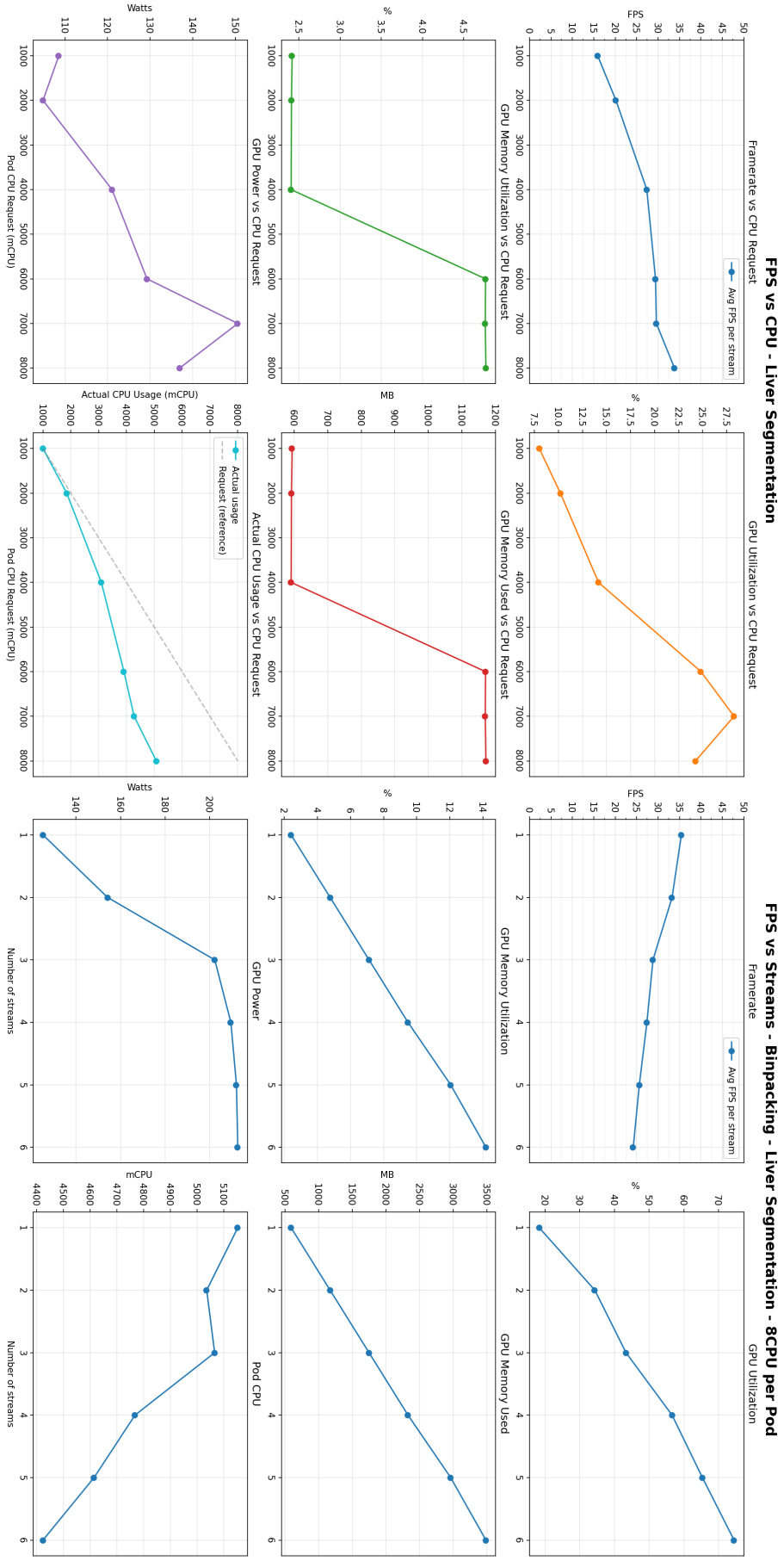


Figure 41: Liver Segmentation workload: (left) CPU profiling showing stable 30 FPS at 8 CPU cores with 8–27.5% GPU utilization, (right) bin-packing scalability maintaining almost consistent performance up to 6 concurrent streams with GPU utilization scaling from 18% to 75%.

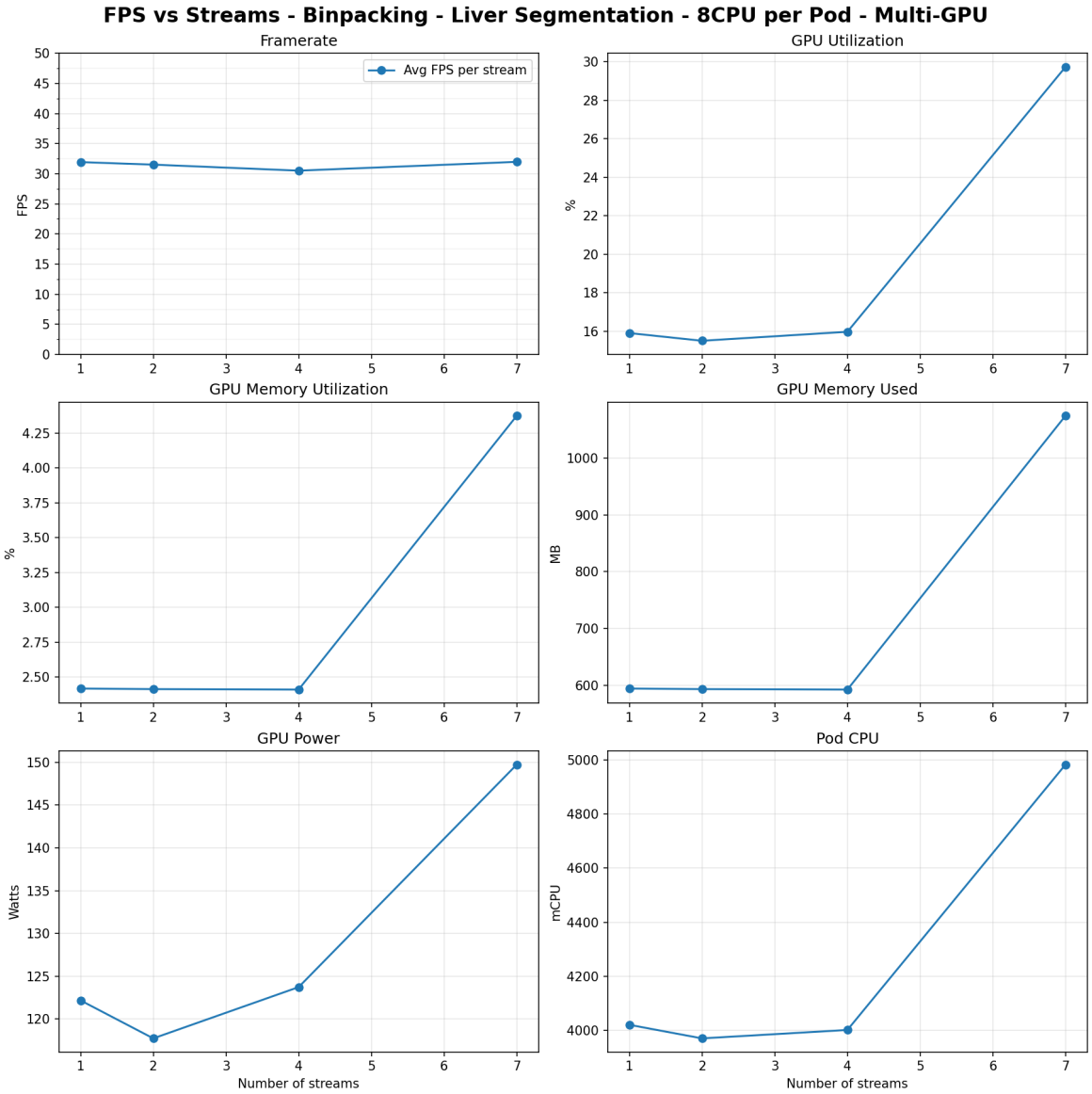


Figure 42: Multi-GPU bin-packing scalability for Liver Segmentation workloads, demonstrating successful consolidation of heavyweight models across multiple GPU devices while maintaining quality of service.

Figure 43 shows results for Instrument Detection workloads. The CPU profiling (left) reveals that framerate plateaus at approximately 30 FPS when allocated 4 CPU cores (4000 mCPU), with diminishing returns beyond this threshold. GPU utilization remains very low at ≈0.2–1.4% across all CPU allocations, indicating that the workload is not GPU-bound but rather requires sufficient CPU resources for video decoding and pre-processing operations. The bin-packing scalability results (right) demonstrate that Instrument Detection workloads scale gracefully from 1 to 15 concurrent streams on shared GPUs, maintaining stable 30 FPS per stream across all configurations. GPU utilization increases modestly with stream count (from ≈2.3% for 1 stream to ≈4.9% for 15 streams), confirming efficient GPU sharing without performance degradation. This validates the feasibility of consolidating multiple streams per GPU for lightweight detection models.

Figure 44 presents results for Phase Detection workloads, which exhibit comparable CPU requirements to Instrument Detection. The CPU profiling (left) shows that this workload achieves 30 FPS with 4 CPU cores, with very low GPU utilization around 0.6–1.7%. CPU usage patterns show near-linear correlation with CPU request allocation, confirming that the Kubernetes resource management system correctly enforces CPU limits. The bin-packing scalability results (right) show similar scaling behavior, with consistent 30 FPS maintained across 1–15 concurrent streams. GPU utilization increases modestly with stream count (from ≈3% for 1 stream to ≈7% for 15 streams), and CPU usage scales linearly with stream count, demonstrating predictable resource consumption. These profiling results establish practical limits for bin-packing: up to 6 lightweight streams or 3 heavyweight streams per GPU while maintaining quality of service.

Phase 2: Baseline vs Bin-Packing Comparison To quantify the resource efficiency gains of bin-packing, we conducted a controlled comparison between baseline deployment (one stream per GPU, no sharing) and optimized bin-packing (multiple streams per GPU). Figure 45 presents the comparison results. The baseline configuration (left) shows 4 streams distributed across 4 dedicated GPUs. Each stream maintains stable 30 FPS, but GPU utilization is only ≈20% across the four devices, indicating significant under utilization. The total system capacity is 4 concurrent streams. In contrast, the bin-packing approach (right) consolidates 12 streams onto 4 GPUs (3 streams per GPU via time-slicing). All 12 streams maintain stable 30 FPS, while GPU utilization increases to ≈50% across the four devices, representing much more efficient resource usage. Compared to baseline, bin-packing achieves 3× higher workload density (12 vs 4 streams) using the same GPU resources (4 GPUs). This translates to a 2.5× improvement in resource utilization without compromising inference quality.

These results validate the intelligent bin-packing optimization strategy for Challenge C1, demonstrating how profiling-based CPU allocation and GPU time-slicing enable efficient multi-dimensional resource packing. By achieving 3× higher workload density (12 vs 4 streams) while maintaining strict frame rate SLOs (≥30 FPS), this approach maximizes utilization of scarce edge resources, allowing NCT to support more concurrent surgeries with limited GPU infrastructure. This directly addresses the hospital's challenge of managing heterogeneous AI models on constrained edge hardware, reducing idle capacity and operational costs while ensuring uninterrupted real-time video analytics for surgical decision-making.

Phase 3: Confidential inference for inference Lastly, we evaluated the performance of adding confidentiality to the inference. This is done by leveraging SCONE's lift-and-shift approach to the workload we have. We designed two experiments: one with GStreamer and one with pure Python. Both experiments were conducted on a 32-core server with an Intel Xeon Gold 6346 processor at 3.10 GHz and 236 GB of memory. For SCONE-variants, we executed in multiple modes: Hardware, EDMM, and Simulation. However, for compactness, we show results only for the Hardware mode, which is more compatible with most hardware.

In the first experiment with GStreamer, the whole framework, including the ML inference plugin, is executed in the TEE. The time is measured by GStreamer itself. Meaning, it begins when the enclave has been created (in the confidential variant) and GStreamer is ready, and continues until the final output has been written to disk. In addition to the ML inference plugin, the process also uses multiple other plugins, such as `qtdemux`, `h264parse`, `videoconvert`, `x264enc`, and `mp4mux`. Here, we

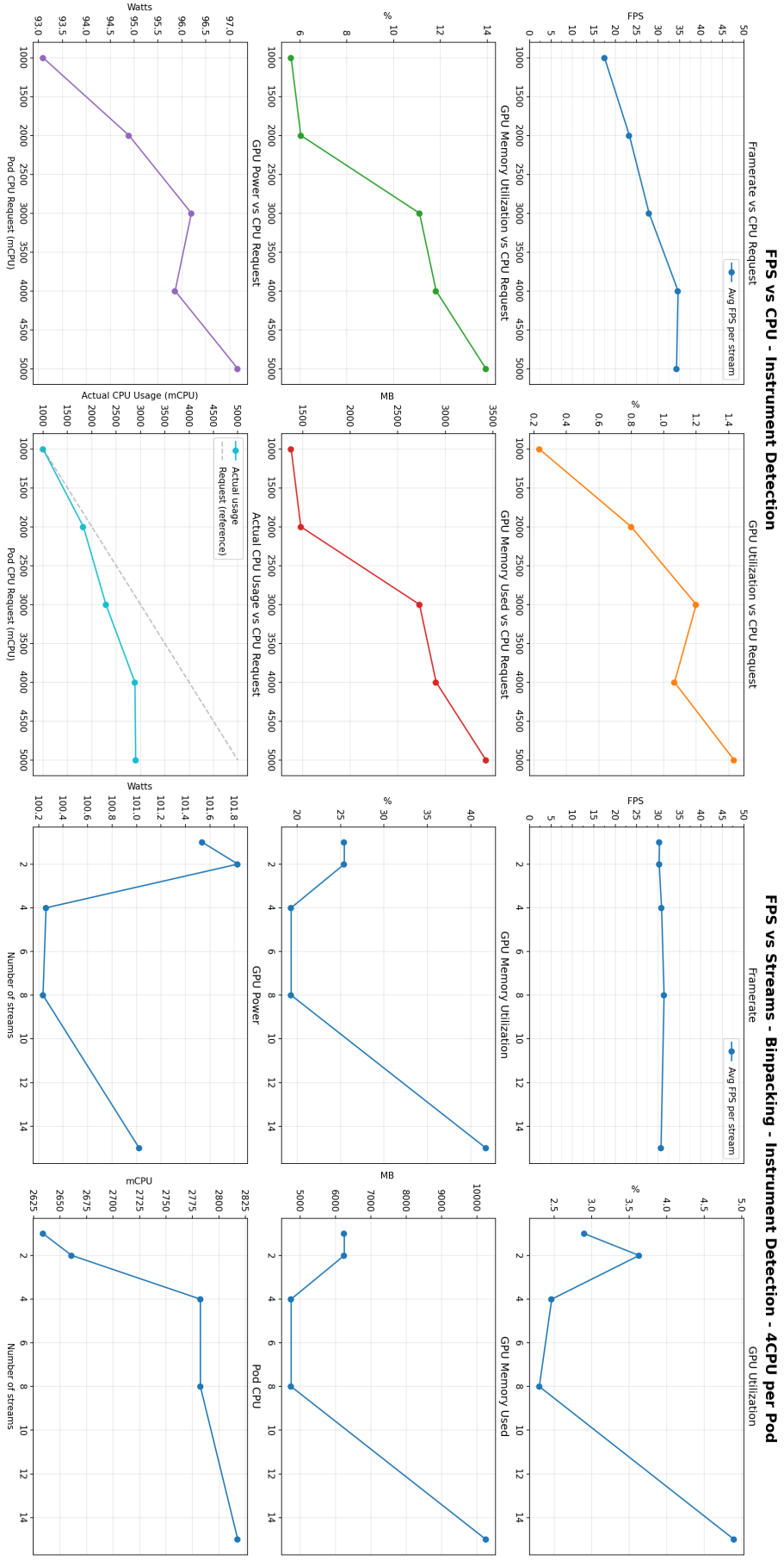


Figure 43: Instrument Detection workload: (left) CPU profiling showing stable 30 FPS at 4 CPU cores with very low GPU utilization (0.2–1.4%), (right) bin-packing scalability maintaining performance up to 15 concurrent streams with GPU utilization reaching 2.3–4.9%.

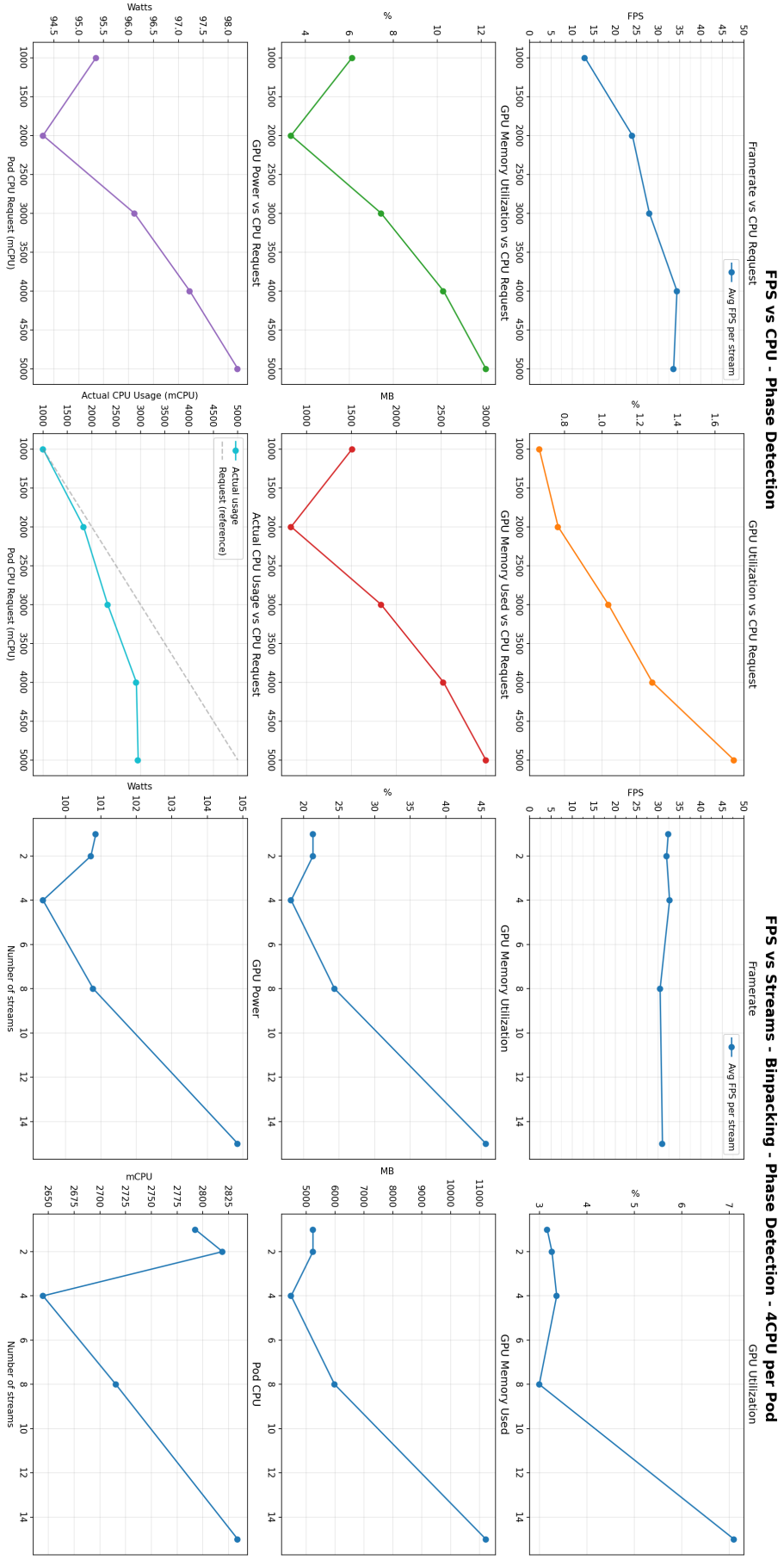


Figure 44: Phase Detection workload: (left) CPU profiling showing stable 30 FPS at 4 CPU cores with very low GPU utilization (0.6–1.7%), (right) bin-packing scalability maintaining performance up to 15 concurrent streams with GPU utilization reaching 3–7%.

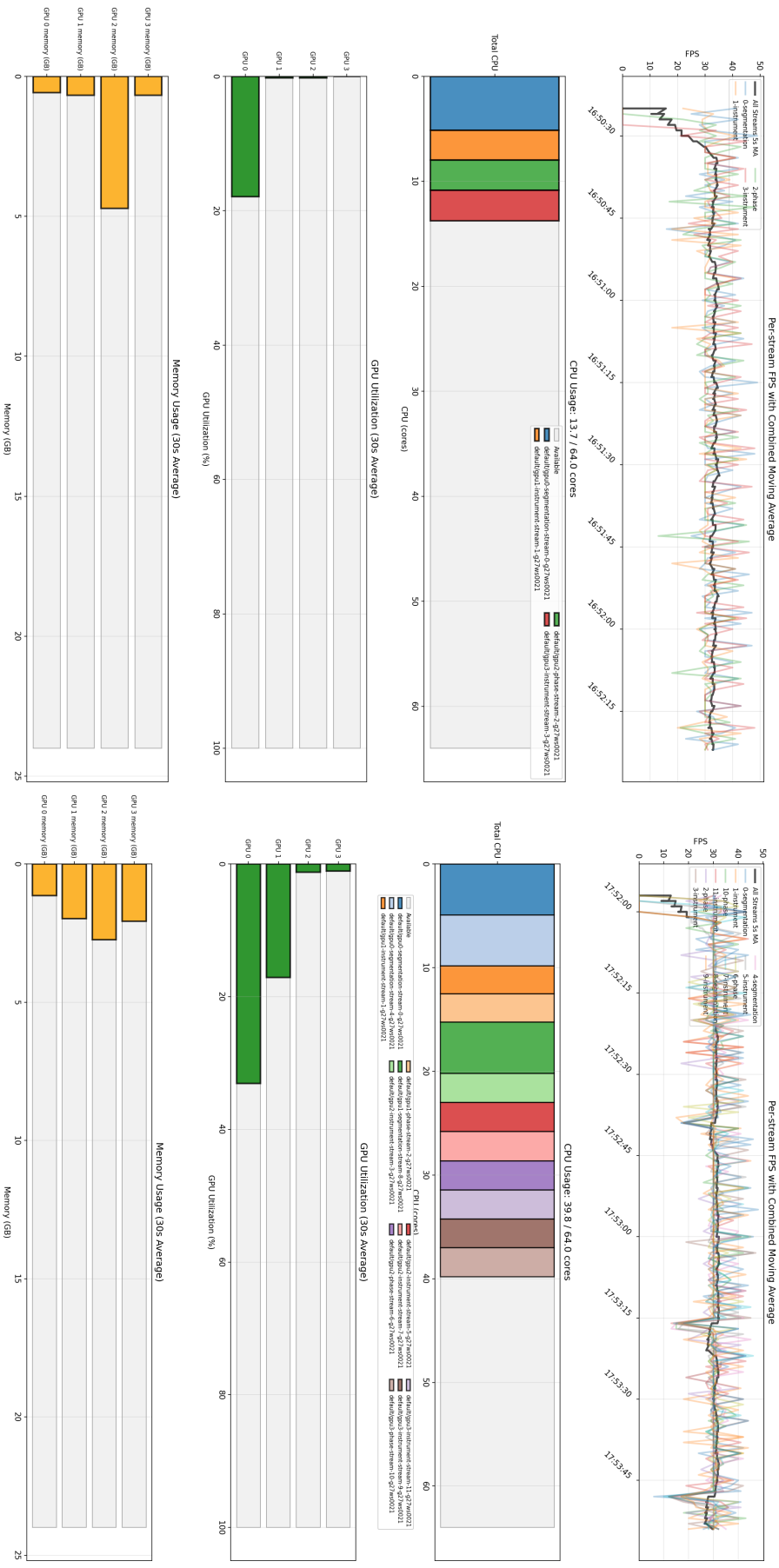


Figure 45: Baseline vs bin-packing comparison: (left) Baseline deployment with 4 streams on 4 dedicated GPUs showing low utilization (≈20%), (right) bin-packing deployment with 12 streams on 4 GPUs showing 3× higher density and ≈50% utilization. CPU utilization rises from ≈21% to ≈63%.

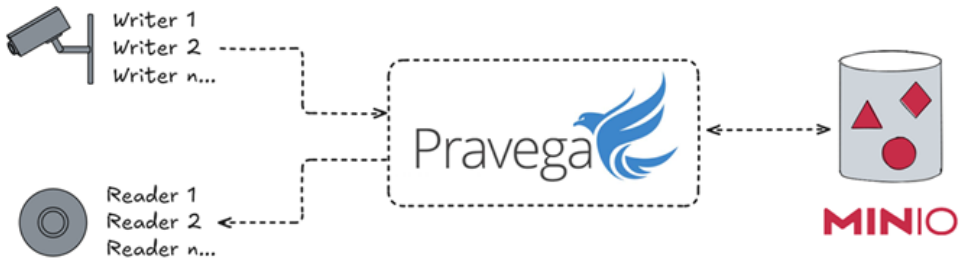


Figure 46: Experiment 2 deployment.

noted that the performance overhead against the native execution is 3.41×.

Since the overhead is relatively large, we consider our second experiment: comparing only the inference. To this end, we implemented a pure Python application to run load and process the input video, perform the inference, and store the combined output on the disk. We start measuring the duration just before the inference and after the loading and pre-processing of the video frame. For completeness, we also add an option to encrypt/decrypt the input and output to understand its overhead. We noted that SCONE's overhead is 14.96% and the encryption overhead (in Native execution) is 2.1%.

Experiment 2 (CH2): Predictive streaming storage auto-scaling A key challenge in the NCT PoC is adapting the streaming infrastructure to fluctuating workloads without incurring latency spikes caused by frequent reactive auto-scaling events. To address this, we proposed a predictive LSTM-based auto-scaling mechanism, which anticipates workload changes and adjusts resources proactively. This mechanism was validated through simulation experiments in D5.2, and the present study serves as an empirical demonstration of its effectiveness.

In particular, our goals in this experiment are:

- *Validate predictive scaling in practice*: Demonstrate the effectiveness of the LSTM-based predictive auto-scaling mechanism in a real-world PoC environment for latency-sensitive video analytics.
- *Compare predictive vs. reactive approaches*: Quantify improvements in system stability and latency by contrasting predictive scaling with traditional reactive auto-scaling strategies in real settings.

The experiments were deployed on a Kubernetes cluster comprising seven nodes, each provisioned with 16 GB of memory, 80–100 GB of persistent storage, and between 8 and 16 CPU cores to ensure sufficient computational capacity for workload scaling. The cluster also hosted a Pravega deployment configured with three Bookkeeper instances, three Zookeeper instances, alongside the default segment store that will be horizontally scaled accordingly. Long-term storage was integrated via MinIO. This way, we ensure the experiments run under realistic conditions (see Fig. 46).

The need for auto-scaling is evident in the use-case we target in this study: the National Center for Tumor Disease (NCT, Germany). Fig. 47 shows the complete 2-month trace that NCT collected and anonymized for us. By a simple inspection of the trace, we can observe strong daily patterns in the utilization of the available surgery rooms (10). Even more, weekends tend to exhibit lower utilization compared to weekdays. These kinds of patterns are inherent to human activity and have been observed in many other scenarios. To ensure realistic simulation and execution of reactive and LSTM-based workloads, the provided NCT workload traces were replayed at 15× real-time speed. This approach allows one week of NCT workload data to be processed by the system in approximately 12 hours of actual time.

The pods generating video streams against Pravega were using a video file from a "phantom" test in NCT. In surgical fields, a phantom is a representation of a human body made with synthetic materials for training purposes. Next, we provide a definition for the main metrics used in this study:

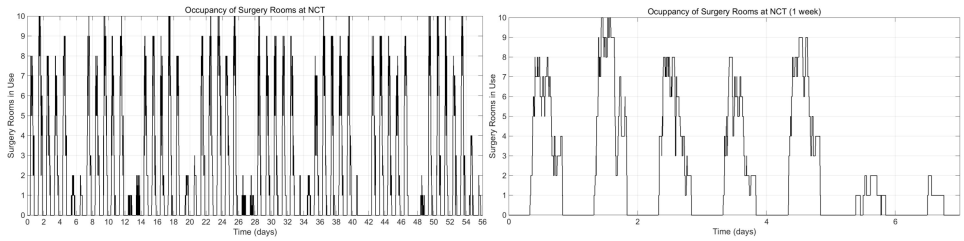


Figure 47: NCT surgery room utilization traces.

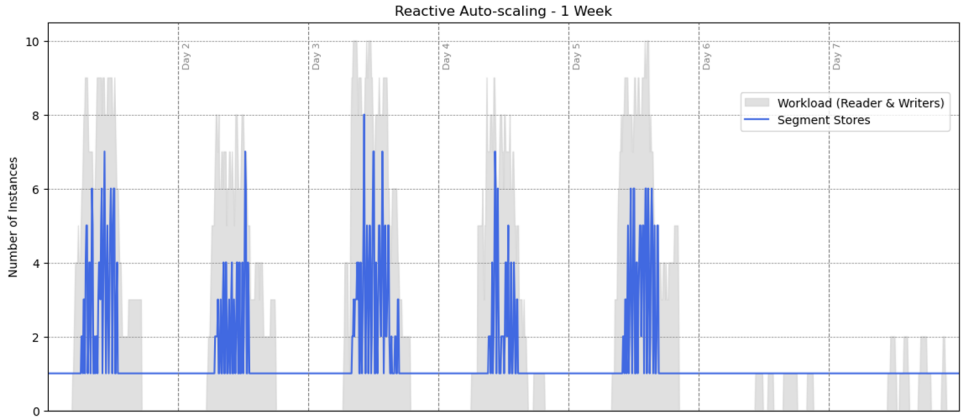


Figure 48: NCT storage buffering and annotation streamlets on Pravega streams.

- *Segment Write Latency*: Captures the latency of write operations to Pravega segment stores. It reflects the impact of scaling events on the streaming storage layer and is key for understanding performance stability.
- *Number of Auto-Scaling Events*: Counts how often the system scales up or down during the experiment. This metric indicates the stability of the scaling algorithm and its ability to avoid disruptive reconfiguration.
- *Tail Latency Distribution*: Analyzes the extreme latency values (e.g., p99) to assess the severity of performance degradation during scaling events. This is particularly important for latency-sensitive workloads like real-time video analytics.

Fig. 48 illustrates the behavior of the reactive auto-scaling algorithm over a 1-week execution period. The plot shows frequent scaling events triggered by short-term latency observations, which lead to instability in the number of active Pravega segment stores. Specifically, the system performs approximately 112 scaling actions, with instance counts oscillating between 1 and 8 segment stores. This oscillatory behavior highlights the main drawback of reactive approaches: they respond to immediate conditions without considering workload patterns, causing unnecessary scaling actions that can disrupt system performance.

In contrast, Fig. 49 depicts the execution under the LSTM-based predictive auto-scaling mechanism. The number of scaling events is 16 in total, which is roughly 7× less than the reactive approach during said execution timeframe. The system exhibits smoother transitions between scaling states, maintaining a maximum store instances of 2 during peak workload hours. This stability stems from the predictive nature of the algorithm, which anticipates workload changes based on historical patterns, reducing the need for frequent adjustments, and minimizing operational overhead.

Fig. 50 shows the segment write latency distribution for reactive auto-scaling during a single day of workload replay. Noticeable latency instabilities and spikes occur around scaling events, with 5 major peaks reaching up to 1 second, while baseline latency (median) stays between 50ms and

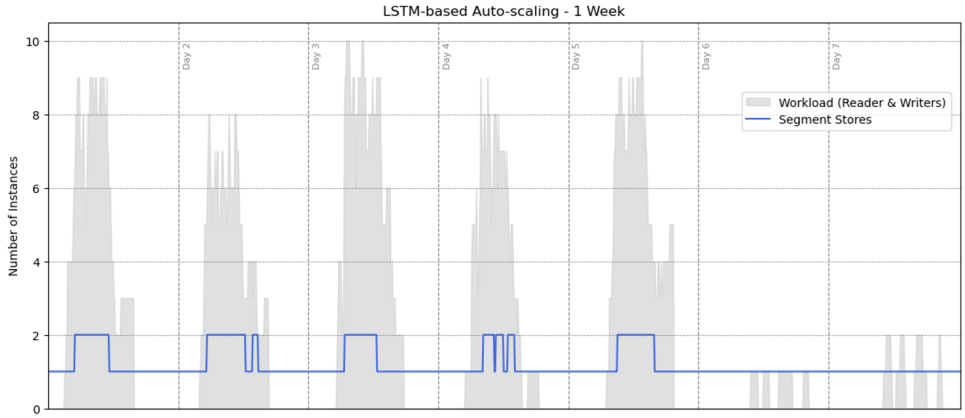


Figure 49: NCT storage buffering and annotation streamlets on Pravega streams.

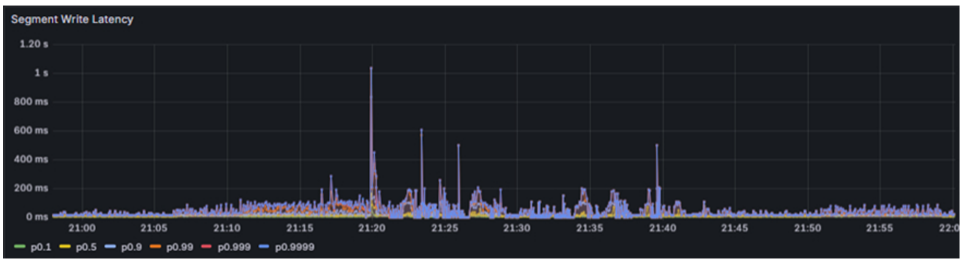


Figure 50: NCT storage buffering and annotation streamlets on Pravega streams.

80ms. These spikes confirm that reactive adjustments introduce transient performance penalties, as tail latencies (p0.99) surge by an order of magnitude during reconfigurations. Such variability can severely impact latency-sensitive applications like real-time video analytics, where maintaining low tail latency is critical. In Fig. 51, the latency profile under predictive scaling is presented. Compared to the reactive approach on the same day's workload, latency remains more stable, with only 2 minor spikes peaking at $\approx$ 600ms, and a median consistently 40ms and 70ms. This represents roughly a 3$\times$ reduction in spike frequency and a 40% decrease in peak latency compared to reactive scaling. These improvements show the advantage of predictive scaling in maintaining consistent and smoother performance, even during workload fluctuations, by reducing disruptive reconfiguration events.

Fig. 52 shows the cumulative distribution function (CDF) comparing the p90 latency for both auto-scaling approaches across the entirety of the workload week. The predictive LSTM-based method achieves a much tighter latency distribution, with 99.9% of requests completing under 150ms, whereas the reactive approach exhibits a heavy tail, with latencies extending up to 1000ms at the extreme. This represents an improvement of nearly 6$\times$ in worst-case p90 latency and eliminates high outliers that can degrade real-time streaming performance. This result reinforces the hypothesis that predictive scaling improves latency guarantees for real-time streaming workloads.

Experiment 3 (CH3): Advance surgical stream data management across the Cloud-Edge To address the limitations of streaming storage in surgical workflows, CloudSkin integrates Nexus, a programmable data management layer that intercepts tiered storage operations between Pravega and long-term object storage (see D3.4). Nexus enables in-transit data processing through *streamlets*, lightweight functions deployed transparently across the Cloud–Edge continuum. For the NCT use case, we developed two specialized streamlets: (i) a *Buffering Streamlet* that provides temporary edge-side persistence during object store outages, ensuring uninterrupted ingestion and real-time analytics in Pravega; and (ii) an *Annotation Streamlet* that enriches video chunks with semantic metadata (*e.g.*, surgical phase, instrument tags) by invoking NCT AI models during offload. These programmable

Figure 51: NCT storage buffering and annotation streamlets on Pravega streams.

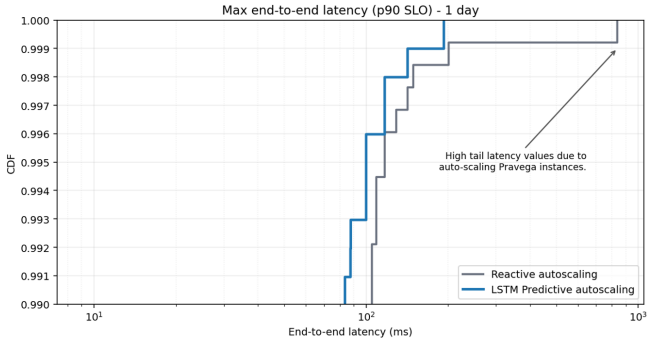


Figure 52: NCT storage buffering and annotation streamlets on Pravega streams.

functions transform the data path into a value-added pipeline, improving resilience and accelerating downstream analytics without disrupting real-time inference.

We evaluate the streamlets developed for NCT, starting with the buffering streamlet designed to provide temporary Edge-side storage and mask long-term storage unavailability in Pravega. Fig. 53 compares Pravega's behavior with and without our streamlet during induced unavailability of its backing object store (MinIO). Pravega is configured with only 1GB of cache to accelerate the issue manifestation. Visibly, ingestion throughput (10MBps) halts after $\approx$ 110s when MinIO becomes unavailable. In contrast, our buffering streamlet absorbs the data during the outage and asynchronously uploads it once MinIO is restored, effectively masking the disruption. This streamlet adds value when relying on Pravega streams to ingest and perform AI on real-time surgical multimedia.

Moreover, the NCT pipeline runs AI models on streamlets to annotate data with semantic tags. Fig. 53 shows the performance of an AI model that detects surgical instruments in JPEG images and tags corresponding stream chunks. `PUT` request latency includes interception, deserialization, inference, and tagging. Chunk size and image sampling rate significantly impact tagging performance (*e.g.*, p50 from 0.3 to 1.2 seconds). These annotations help NCT data scientists quickly locate relevant video fragments within object buckets.

4.5 Demos

Demo 1: Smart CPU & GPU Allocation for NCT Surgical AI Models (CH1) The demo showcases a real-time comparison between traditional GPU allocation and CloudSkin's intelligent bin-packing strategy for NCT's surgical video analytics workloads. Using a Kubernetes-based edge cluster with NVIDIA RTX A5000 GPUs and time-slicing capabilities, the demo processes live surgical video streams ingested via Pravega and GStreamer through three heterogeneous AI inference pipelines: instrument detection, phase recognition, and liver segmentation. In the baseline mode, each video stream is allocated to a dedicated GPU following the conventional one-stream-per-GPU approach, resulting in low GPU utilization ($\approx$20%) and limited capacity to support concurrent surgeries. The bin-packing mode leverages profiling-based CPU allocation and GPU time-slicing to consolidate multiple streams onto shared GPUs, packing 3 streams per GPU while respecting multi-

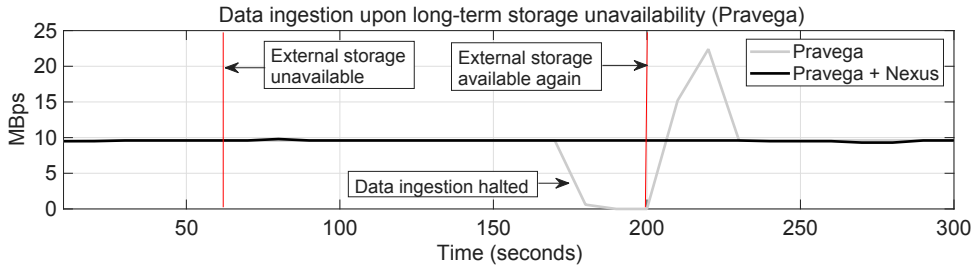


Figure 53: NCT storage buffering and annotation streamlets on Pravega streams.

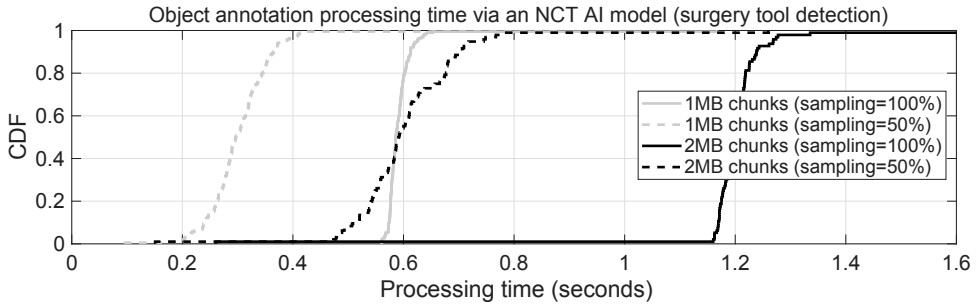


Figure 54: NCT storage buffering and annotation streamlets on Pravega streams.

dimensional constraints: CPU cores (4 for instrument detection, 4 for phase detection, 8 for segmentation), GPU memory limits, and frame rate SLOs ($\geq$30 FPS). The demo visually contrasts both approaches through live Grafana dashboards: per-stream framerate charts maintaining stable 30 FPS across all streams, GPU utilization metrics revealing $\approx$50% under bin-packing versus $\approx$20% under baseline, and resource allocation views displaying the consolidation strategy. CPU usage likewise improves by climbing from $\approx$21% to $\approx$63%. Viewers observe how intelligent bin-packing achieves 3$\times$ higher workload density (12 streams versus 4 streams on the same hardware) without compromising inference quality or latency SLOs. This demonstration highlights CloudSkin's solution to Challenge CH1, enabling hospitals to maximize utilization of scarce edge resources and support more concurrent surgical procedures with limited GPU infrastructure, reducing operational costs while ensuring uninterrupted real-time AI assistance during surgeries.

Demo 2: LSTM-based predictive streaming storage auto-scaling (CH2) The demo showcases a real-time comparison between reactive auto-scaling and an LSTM-based predictive scaling approach for streaming storage services in a surgical edge scenario. Using Pravega as the elastic streaming storage system, the demo begins by replaying anonymized operating room workload traces to emulate fluctuating video ingestion demands. In the reactive mode, scaling decisions are triggered by short-term latency thresholds, resulting in frequent oscillations and visible latency spikes during reconfigurations. The predictive mode leverages an LSTM model trained on historical workload patterns to anticipate resource needs and proactively adjust segment store instances. The demo visually contrasts both approaches through live dashboards: segment write latency charts, tail latency distributions, and the number of scaling events. Viewers observe how predictive scaling achieves smoother transitions, fewer scaling actions, and stable low-latency ingestion, ensuring uninterrupted AI video inference during surgeries. This demonstration highlights the tangible benefits of predictive elasticity for latency-sensitive edge workloads.

5 Use case: Agriculture

5.1 Overview

5.1.1 Business story

This use case addresses sectoral and technological challenges related to the analysis of sensor data, aiming to enable the development of technological ecosystems for advanced analysis of agricultural and environmental information.

On the one hand, the local use and retention of data by data providers, due to a lack of control over its use, coupled with the diversity of data origins and types, limits analytical and decision-making capabilities and hinders the development of related services and technologies at the local level.

The first experiment **deploys an agricultural dataspace that will unify and facilitate the location of datasets**, regardless of their origin (sensors, edge servers, private or public cloud). It will also clarify the semantics and units of measurement of the data through the definition of data dictionaries and ensure secure and controlled access and use of information through contracts and terms of service.

Above this data space, the second experiment demonstrates the ability to **build advanced information services from this data through a cloud-based computational analytics service** that integrates and manages complex data, such as geospatial data from satellites.

Finally, and due to market demands, to make this solution viable, artificial intelligence models are trained and analyzed to predict the optimal resources needed for data processing. This allows for reduced execution times and costs by dynamically adapting them to the needs of the datasets.

5.1.2 Why this use case needs the compute continuum?

Due to the type of information coming from agricultural sensors, and the complexity of managing geospatial information, building data analysis and management services faces a wide variety of data sources, volumes, and computing requirements.

Applying continuous computing allows this use case to benefit from task distribution across different levels of the continuum, achieving the flexibility required by computational processes.

Among the advantages of its application, we highlight the reduction in the volume of data sent to the cloud, dynamically optimizing computational resources and allowing applications to continue functioning regardless of connectivity capacity in rural environments, by enabling data preprocessing and filtering at the edge or in intermediate nodes.

From a data security perspective, the use of the continuum also allows for edge preprocessing, which facilitates compliance with local regulations (e.g., GDPR) and minimizes data exposure in the cloud. On the other hand, the scalability of continuous computing guarantees the management of necessary resources and data transfer between levels, leveraging the advantages of a distributed infrastructure that combines the local capabilities of nodes, generally connected directly to sensors, with global capabilities, enabling more advanced analysis and decision-making services.

In conclusion, and for this use case, continuous computing is perfectly suited to the origin and processing needs related to agricultural and environmental information.

5.2 Cloud-Edge continuum infrastructure for the mobility use case

5.2.1 Cloud-Edge hardware

The research project is based on a hybrid cloud and edge infrastructure that enables flexible data acquisition, processing, and storage in distributed environments. The first experiment, focusing on data space management, runs on a dedicated server within the KIO NETWORKS infrastructure. This implementation is designed to aggregate heterogeneous data streams from edge servers and IoT sensors. By integrating edge computing nodes, the system reduces latency, performs preliminary data filtering and normalization, and ensures secure transmission to the centralized data space. The architecture prioritizes scalability, interoperability, and compliance with data governance standards, enabling efficient experimentation with diverse data sources.

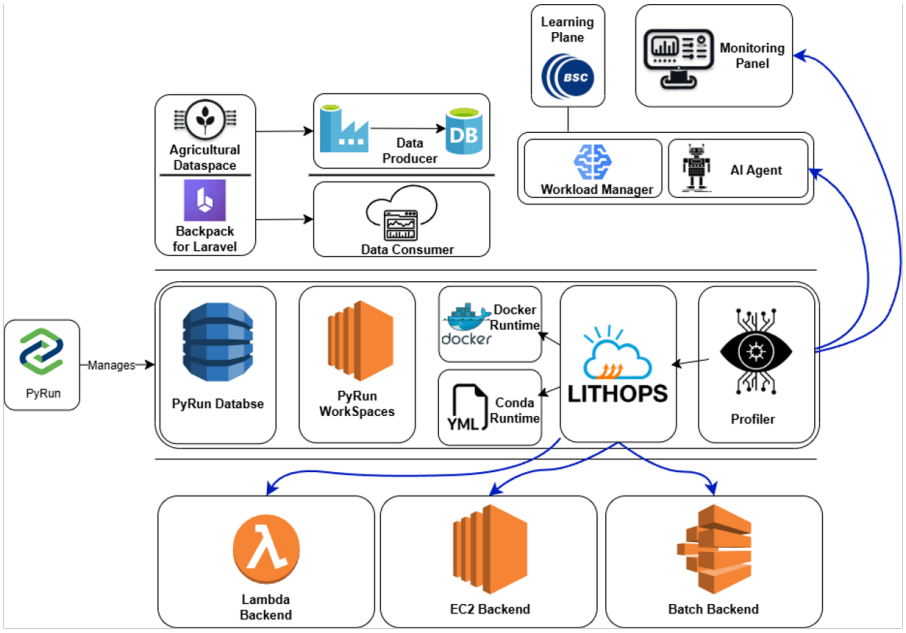


Figure 55: CloudSkin platform for the Agriculture use case

The second experiment addresses geospatial data integration and leverages native AWS services to dynamically predict computing resources, achieving flexible performance and predictable scalability. This implementation utilizes AWS capabilities for data management, processing pipelines, and advanced analytics, incorporating AI models for prediction of resource allocation. By dynamically adjusting computing resources based on anticipated workloads, the experiment optimizes cost-effectiveness and ensures high availability and responsiveness. The combination of AI-driven predictive scalability with geospatial data integration demonstrates the potential of cloud and edge synergy to support complex data-intensive treatment.

5.2.2 CloudSkin platform

The first platform developed within this research initiative is a dedicated data space focused on agricultural and environmental data. Its primary objective has been to address the legal and technical complexities inherent in data usage and sharing.

The agricultural and environmental dataspace is not only a platform for securely sharing information between data providers and consumers; in addition to the complexities of usage conditions and understanding the data's units of measurement, it represents independence of the information's origin from its computing level (sensors, mobile devices, edge servers, private or public cloud), allowing access for cloud services regardless of the volume and type of data, and the type of service requiring access.

The agricultural dataspace translates into the creation of a sector-specific channel for agriculture and the environment, enabling the standardization and universalization of data exchange practices in these fields on a single platform. This solution lays the foundation for third parties to develop new information management and analysis services, leveraging the available datasets to generate added value and foster innovation.

The second platform demonstrates the integration capabilities of the dataspace with external cloud-based services, particularly those requiring the processing of complex datasets such as geospatial information. This integration is done through RESTful APIs and standardized connectors, allowing both the ingestion of data from the data space to external services and the return of processed results back to the data space, as demonstrated by the second experiment, with the integration with a geospatial data analysis pipeline.

Beyond data integration, this platform incorporates mechanisms for predicting the computational

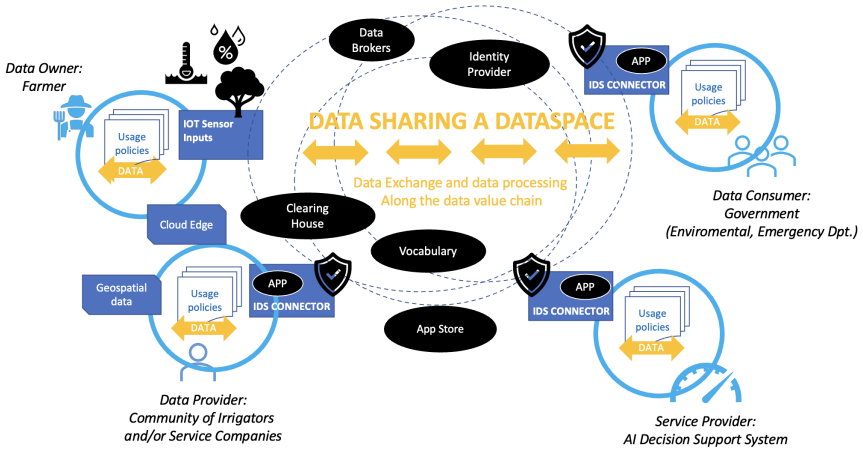


Figure 56: Dataspace relationships.

resources required for optimal performance. This functionality provides valuable insights for system dimensioning and cost forecasting, thereby supporting the efficient and sustainable deployment of advanced analytical services in cloud environments.

Complementing these efforts, a third platform has been designed to monitor and visualize resource consumption in real time. This measurement platform delivers key performance indicators that allow researchers and stakeholders to evaluate service behavior during execution in cloud infrastructures. Together, these three platforms form a unified use case that provides a comprehensive technical solution for data management, as shown in Figure 55. From sensor-based detection, through semantic harmonization using standardized data dictionaries, to complex analysis and predictive dimensioning within cloud services, the ecosystem addresses the full lifecycle of agricultural and environmental data. Data

5.3 Experiments, KPIs and benchmarks

Experiment 1: Dataspace The first experiment addresses the challenge of **creating an agricultural data space as a solution for data sharing** in the complex environment of agricultural and environmental management. This data space also serves as a practical case study to analyze the integration of cloud computing, edge computing, and Internet of Things (IoT) technologies. It includes datasets with information from sensors and other sources, that are analyzed and used by third parties.

This experiment tackles the challenge of addressing the widespread reluctance to share information in the agricultural sector, as well as the complexity of accessing data from agricultural and environmental sensors.

As a solution, a prototype cloud-based data space has been developed that ensures responsible use of information for both the data provider and the data user (as shown in Figure 56).

The first experiment, in addition to incorporating requirements for warranted use, availability, and data sharing in accordance with the conditions established by the data producers, analyses the execution impact related to the loading and sharing of information from sensors and edge servers with the dataspace's cloud platform, thereby enabling the determination of the capacity and performance of the developed platform.

Experiment 2: Water usage footprint In Experiment 2, we are developing an advanced analytics application to measure water consumption through:

- Sensor data, including humidity and well water levels,
- Sentinel-2 multispectral satellite images, and
- LiDAR data for spatial precision.

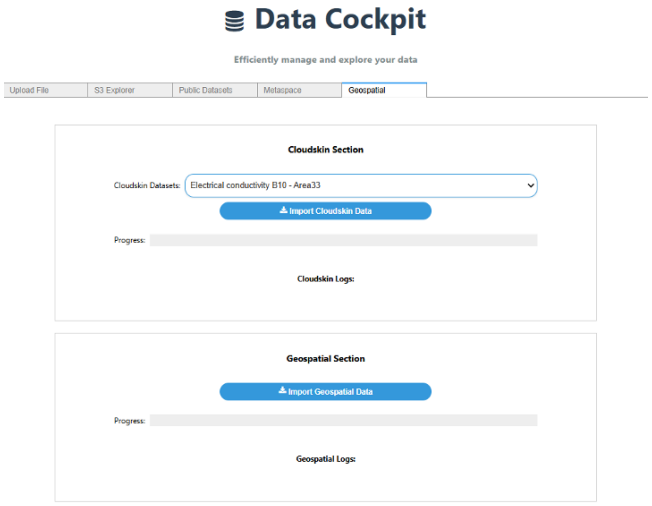


Figure 57: Data consumer interface

The experiment studies how to develop at a technical level:

- Predictive analytics for better decision-making,
- Integration of diverse data sources,
- Scalable infrastructure,
- Efficient data storage and processing,

Technically, this experiment works on:

- AI-driven crop classification using Sentinel-2 data, and
- other data from cloudskin project, enabling multi-source data integration for efficient decision-making.

The Proof of Concept has the goal to validate the technical feasibility of this project.
From the data perspective:

- Data Producers gather information and send it to the Agricultural Dataspace.
- Data Consumers analyze the data to generate actionable insights.
- The Learning Plane, powered by AI, ensures efficiency and scalability, with workloads managed by the Workload Manager.

This Proof of Concept demonstrates the integration of different technologies to address complex agricultural challenges while maintaining adaptability and efficiency.

I need to explain some context, such as LiDAR, Sentinel-2 data, how the integration is done, etc. Also, since you're talking about the monitoring system and the control panel, it would be helpful to include a screenshot.

The second experiment obtains data from the dataspace and manages satellite imagery, analyzing the different layers and dividing the data to perform the analysis pipeline coherently and efficiently.

Our monitoring system provides a real-time overview of pipeline performance. The dashboard features:

- CPU Usage for quick identification of bottlenecks,
- Disk & Network Usage, showing read/write operations and network activity,

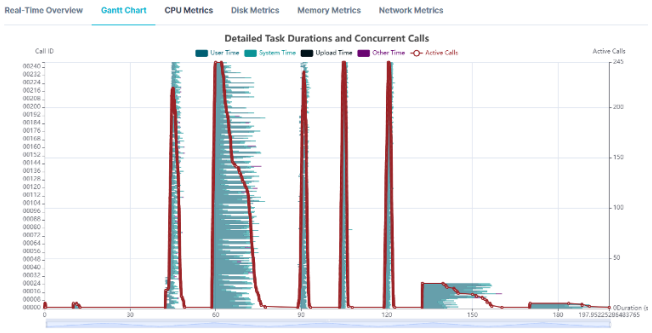


Figure 58: Execution performance with AI optimization

- Data Transfer Speed, tracking rates across storage and networks, and
- CPU Utilization Breakdown, displaying the distribution of processing tasks.

This comprehensive view ensures rapid detection of irregularities and efficient resource allocation.

5.4 Results

To achieve the experiment, multiple software components were developed and various integrations were performed, which we consider a valuable and reusable software outcome. We highlight the following:

- Profiler and monitoring API, for capturing CPU, memory, network, and storage metrics without impacting performance.
- PyRun Platform, a multi-user environment with IAM authentication, DynamoDB storage, and dynamic dashboards.
- DataPlug, a serverless library that allows partitioning and storing geospatial files in S3 with geographic consistency.
- DataCockpit, an interface for importing data, configuring parameters, and launching AI-optimized executions.
- Learning Plane, including the training of an XGBoost artificial intelligence model that learns from real-world executions and recommends optimal configurations, reducing time and costs.

From the perspective of scientific results, the work culminated in an international publication titled “Intelligent Optimization of Distributed Pipeline Execution in Serverless Platforms: A Predictive Model Approach,” presented at the 10th International Workshop on Serverless Computing (ACM WoSC10 2024, in Hong Kong). Among the most notable results:

- 79.9% reduction in execution time.
- 30% reduction in costs.
- MAE of 12.5 seconds and R^2 of 0.92, a 75% improvement over historical data.

Key contributions include:

- Effective integration of machine learning and serverless computing to optimize scientific pipelines.
- Experimental validation with a geospatial water consumption pipeline.
- A reproducible methodological framework applicable to other scientific domains.

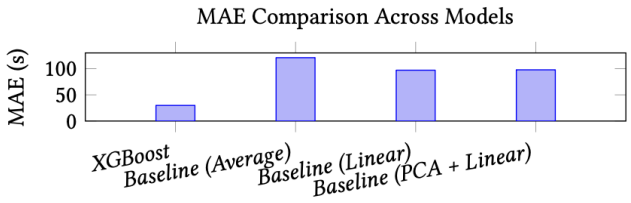


Figure 59: Mean Absolute Error (MAE) comparison across models.

We compared several methods to assess their performance in predicting and optimizing the duration of the geospatial analysis pipeline:

- XGBoost Model: Gradient boosting model that captures complex feature interactions.
- Average (Baseline): Predicts using the mean execution time from training data.
- Linear Regression (Baseline): Assumes linear relations between features and duration.
- PCA + Linear Regression (Baseline): Uses PCA for dimensionality reduction before linear regression.
- Design Space Analysis (DSA): This technique involves systematically exploring the possible configurations of a system with the goal of finding the best one, but is computationally costly.

Table 10: Comparison of Models.

Model	**MAE (s)**	**Avg. MAE (CV) (s)**	**MAPE (%)**	**R^2**
XGBoost	29.81	34.20	8.72%	0.8802
Baseline (Average)	120.90	–	–	–
Linear Regression	97.02	96.62	28.73%	0.3380
PCA + Linear Regression	97.70	92.03	29.04%	0.3240

Cost-Benefit Analysis and Efficiency

Beyond its predictive accuracy, the XGBoost model offers significant cost savings compared to the exhaustive Design Space Analysis (DSA). Table 11 compares configurations resulting in the minimum and maximum execution durations, along with their respective costs per execution.

Table 11: Minimum vs Maximum Duration Configuration and Cost.

Parameter	**Minimum Duration**	**Maximum Duration**
Number of Files	5	5
Splits	5	2
Input Size (GB)	0.25	0.25
Runtime Memory (MB)	2000	1024
Ephemeral Storage (MB)	1024	1024
vCPUs	1.13	0.58
Duration (s)	184.08	915.89
Cost per Execution (USD)	0.281	0.350
Cost Difference (USD)	0.069	

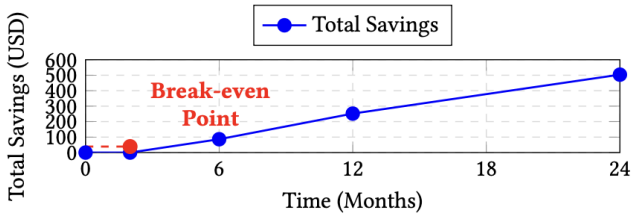


Figure 60: Projected cost savings over time, assuming 10 executions per day. The break-even point occurs at approximately 2 months.

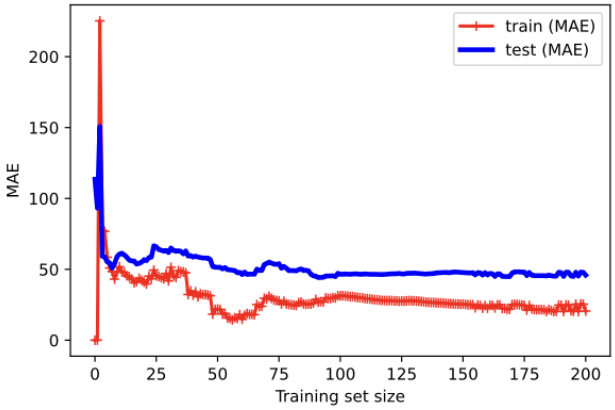


Figure 61: Learning curve of the XGBoost model

The cost difference between these configurations is $0.069 per execution, resulting in approximately 19.71% savings per run. The initial training cost of $38.75 for the XGBoost model leads to a break-even point after approximately 562 executions.

Assuming a rate of 10 executions per day, the model reaches the break-even point in about two months.

In summary, the XGBoost model delivers substantial cost savings and a rapid return on investment. By efficiently identifying optimal configurations without exhaustive testing, it proves to be an effective and practical solution for ongoing pipeline optimization.

Comparison of Real vs. Predicted Duration

To evaluate the XGBoost model's ability to identify optimal configurations, we tested it on all configurations from the Design Space Analysis (DSA), including both seen and unseen setups. The configuration with the shortest real duration, selected by the DSA, was excluded from the training set to evaluate whether the model could identify it as the most efficient in the test set. The results indicated that the configuration with the shortest predicted duration (195.26 seconds) closely matched the actual duration of 184.08 seconds. This alignment demonstrates the model's effectiveness in identifying optimal configurations without the need for exhaustive testing.

Learning Curve Next figure shows the learning curve of the XGBoost model. The graph indicates a significant reduction in error as the size of the training dataset increases. The narrow gap between training and test errors indicates minimal overfitting and strong generalization capacity. During model training, the early stopping rounds technique was implemented to prevent overfitting. This technique stops training when no significant improvements in validation error metrics are observed after a specified number of rounds, ensuring more efficient and robust training.

In summary, this Cloudskin use case demonstrates that artificial intelligence can automate decision-making in serverless environments, reducing costs, accelerating research, and improving scientific productivity. With this, we are moving towards smarter, more sustainable, and science-oriented computing for the future, capable of leveraging artificial intelligence to improve performance, re-

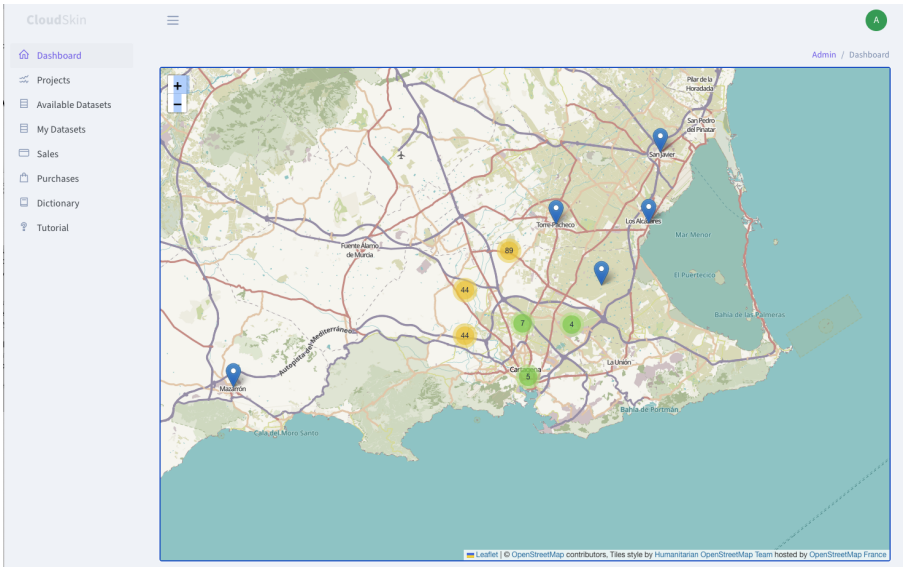


Figure 62: Geographic search functionality for datasets

duce economic and processing costs, and mitigate the technology's footprint.

5.5 Demos

The demonstrations of each experiment, in addition to validating the fulfillment of their specific objectives, are carried out in an integrated way with Cloudskin, not only through the use of the computational power of KIO and AWS, or through the integration capacity of the Learning Plane related to the applied artificial intelligence model, with the one developed by BSC, but also through the standardization of APIs and data sharing systems, which allow both the validation of the use case and the creation of interrelationships between the solutions, for the different technological environments of Cloudskin.

Demo 1: Dataspace Each data provider has a biased and very limited view of the information, preventing global data analysis that delivers added value and enables the development of advanced automation and decision-support solutions. The first experiment addresses this shortcoming by providing an environment that enables the secure sharing of information, along with the necessary selection functionalities to ensure that the analyzed information is accurate, up to date, and accessible.

The demonstration of the first experiment focuses on how this agricultural dataspace tackles data management challenges and meets the requirements for information sharing, while abstracting the data source, which may originate directly from sensors, edge servers, or private or public clouds, thanks to centralization within the agricultural and environmental dataspace.

A solution is also presented through the dataspace to overcome the main barriers identified for information sharing in the agricultural environment. On the one hand, these barriers are related to information standardization, as it is necessary to work with multiple datasets that may include, to cite some relevant examples, different units of measurement and volumes. The use of customized, platform-definable data dictionaries makes it possible to address this barrier. However, one of the main reasons for reluctance to share information is related to control over how the information is used. To this end, the dataspace provides data usage and sharing control functionalities through the creation of a system of contracts and access and usage conditions for the data.

Demo 2: Water usage footprint The second experiment builds upon the dataspace, a cloudless service for advanced data analysis integrated with geospatial information. In addition to being a practical application of the agricultural dataspace as a data source, it performs advanced analysis of complex data within a cloudless distributed environment, demonstrating the use of continuous computing for decision-making solutions and complex calculations.

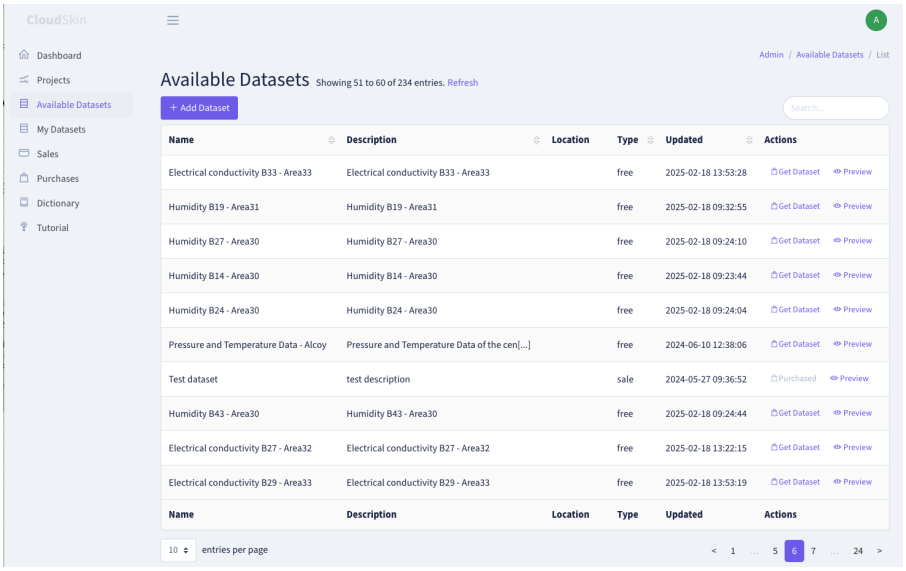


Figure 63: Dataset acquisition functionality

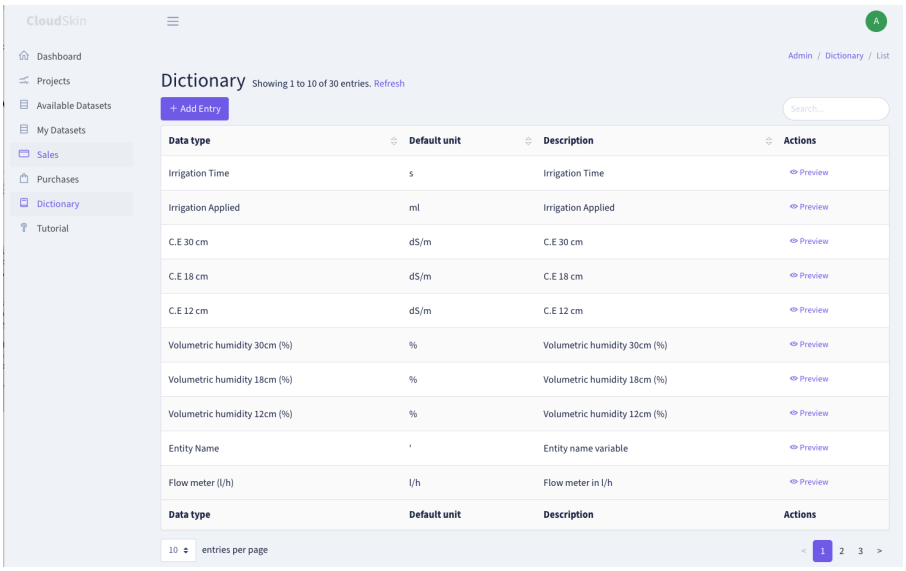


Figure 64: Data dictionary functionality

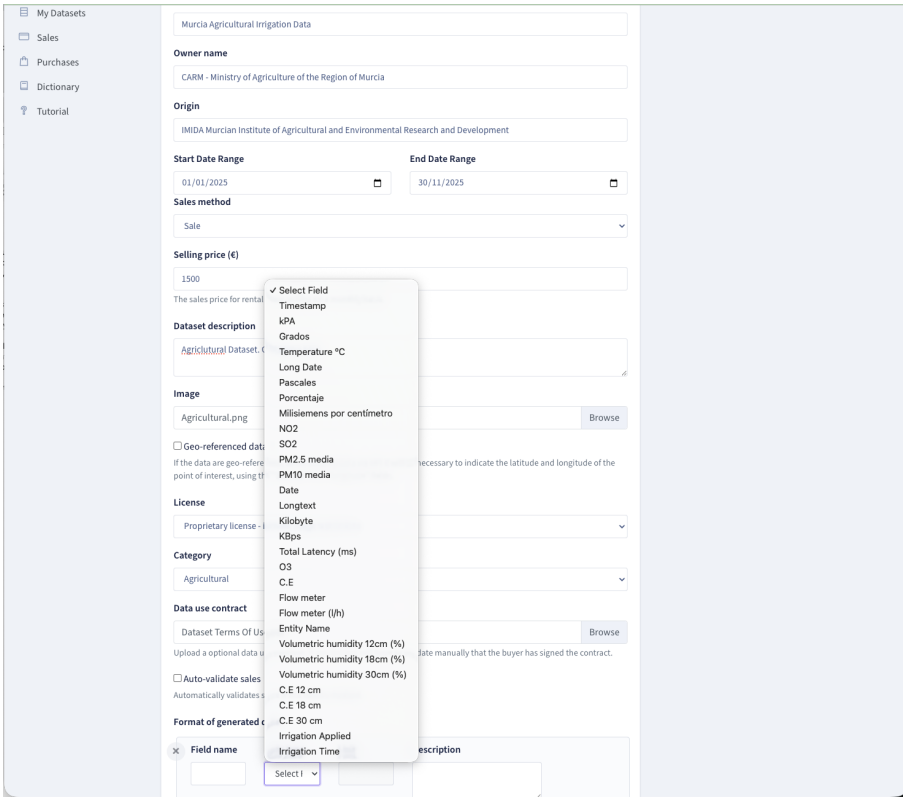


Figure 65: Functionality for creating and importing data

This experiment, besides the advanced processing of geospatial information, follows a transition process depending on the state of the information.

From a data perspective, it develops a Profiler and metric APIs, integrated with Lithops and Pyrun, allowing not only the information subject to analysis to be available but also the performance and execution metrics necessary for process optimization—metrics that must not interfere with the actual performance data during the metric collection process. From an information perspective, a multi-user control platform integrated with Pyrun is created, enabling real-time metric visualization and analysis through the development of a dashboard mechanism based on Grafana.

From an artificial intelligence perspective, a predictive model has been trained using XGBoost and Optuna, validated with over 150 real executions of agricultural and environmental pipelines, capable of predicting the optimal resources required for performing advanced calculations on geospatial and dataspace data. This optimized cloud resource configuration reduces both the execution time of processes and execution costs, offering a scalable solution with controlled costs.

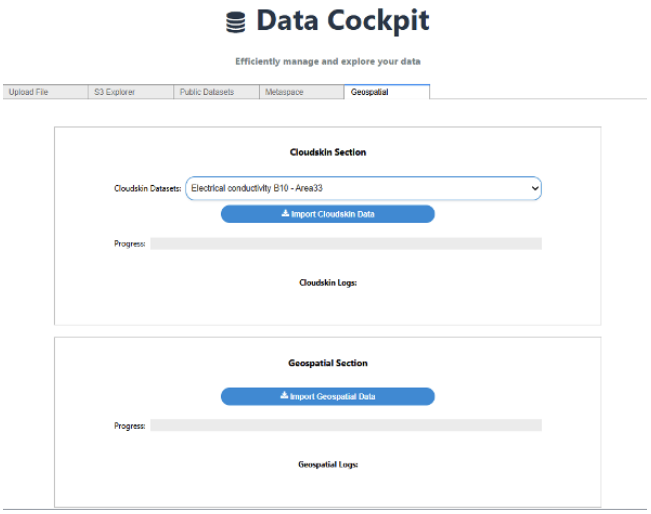


Figure 66: DataCockpit, an interface for importing data, configuring parameters, and launching AI-optimized runs

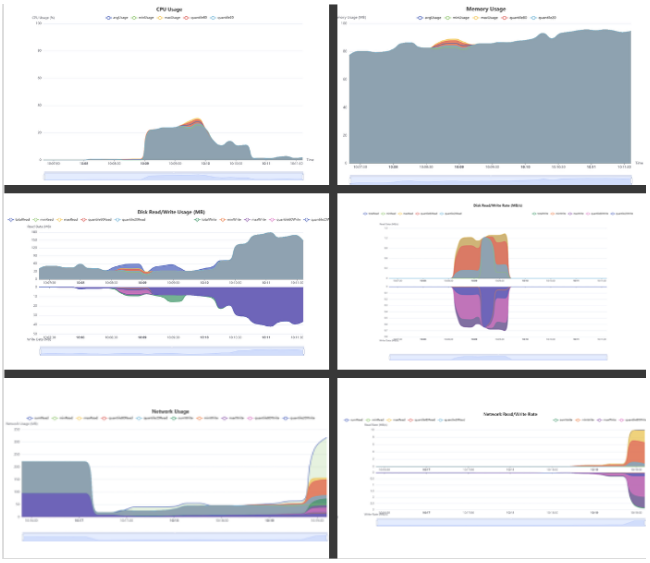


Figure 67: Data visualization and management, with advanced dashboards created in Grafana, Vue.js + Apache ECharts, integrated directly into PyRun

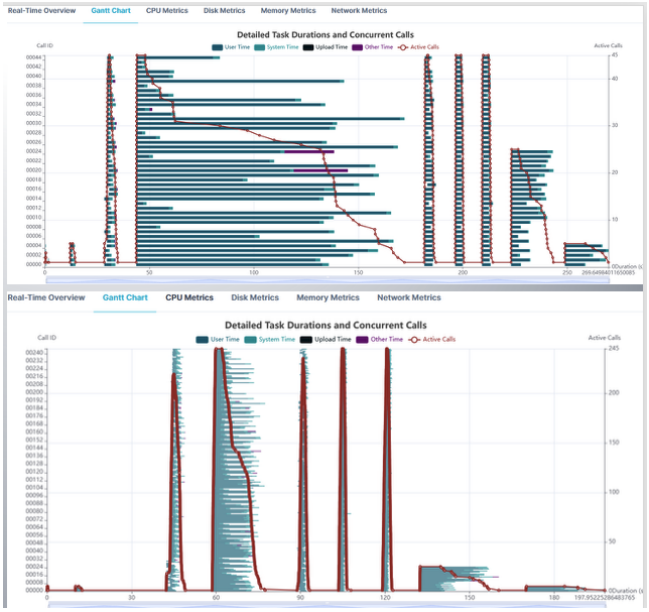


Figure 68: Approach achieves a 27% performance improvement compared to a known good baseline configuration, and up to 70% improvement when compared to suboptimal configurations, while maintaining identical output

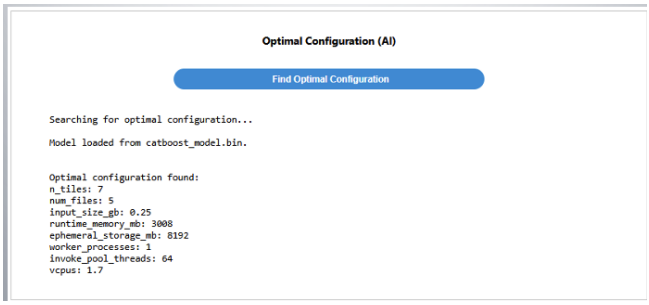


Figure 69: AI configuration proposal

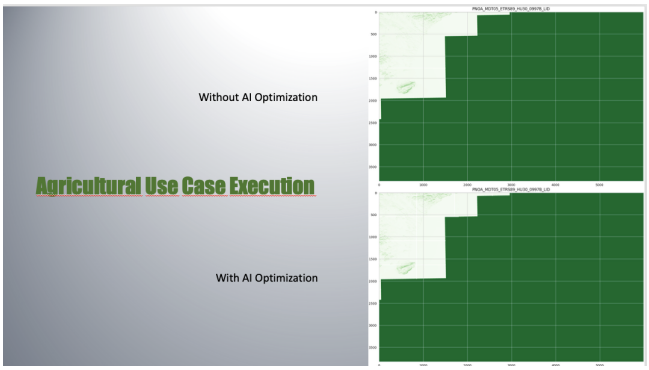


Figure 70: Comparative execution results

6 Conclusions

This deliverable shows the results of how Cloudskin technologies, in terms of platform and components, could help with different use cases. The proposed proof of concept, methodologies and demos can be reproduced for the other related use cases, empowering the business with better performance efficiency, lower cost, better reliability, scalability and confidentiality.

References

[1] CloudSkin Deliverable 2.3 Adaptive virtualization for AI-enabled Cloud-edge Continuum. CloudSkin, June 2024.

[2] P. Liu, J. O. Torra, M. Palacín, J. Guitart, J. L. Berral, and R. Nou, "Data-connector: An agent-based framework for autonomous ml-based smart management in cloud-edge continuum," in 2024 IEEE 32nd International Conference on Network Protocols (ICNP), pp. 1–6, 2024.

[3] "Amazon ec2 on-demand pricing." https://aws.amazon.com/ec2/pricing/on-demand/.

[4] "Electricity price." https://www.globalpetrolprices.com/Spain/electricity_prices/.

[5] "Amazon compute service level agreement." https://aws.amazon.com/compute/sla/.

[6] M. Macías and J. Guitart, "Sla negotiation and enforcement policies for revenue maximization and client classification in cloud providers," Future Generation Computer Systems, vol. 41, pp. 19–31, 2014.

[7] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi, "Darknetz: towards model privacy at the edge using trusted execution environments," in Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services, MobiSys '20, (New York, NY, USA), pp. 161–174, Association for Computing Machinery, 2020.

[8] H. Chang, H. Jia, Y. Yang, B. Wang, and T. Zhang, "Ginver: Training-free collaborative inversion attacks against split learning," in Proceedings of the ACM Web Conference 2023, pp. 1952–1962, 2023.

[9] "Pravega." https://cncf.pravega.io, 2025.

[10] R. Gracia-Tinedo, F. Junqueira, T. Kaitchuck, and S. Joshi, "Pravega: A tiered storage system for data streams," in ACM Middleware '23, p. 165–177, Association for Computing Machinery, 2023.

[11] "Gstreamer." https://gstreamer.freedesktop.org/, 2024.

[12] "Pravega - gstreamer connector." https://github.com/pravega/gstreamer-pravega, 2024.