CloudSkin

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

CloudSkin

(grant agreement No 101092646)

Adaptive virtualization for AI-enabled Cloud-edge Continuum

D2.5 Reference implementation of architectural building blocks

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	52
WP/Task related to this document	WP2 / T2.1
WP/Task responsible	URV
Leader	Marc Sanchez-Artigas (URV)
Technical Manager	Peini Liu (BSC)
Quality Manager	Raúl Gracia (DELL)
Author(s)	Ardhi Putra Pratama Hartono (TUD), Raúl Gracia (DELL), Hossam Elghamry (DELL), Alan Cueva (DELL), Peini Liu (BSC), Joan Oliveras Torra (BSC), Jordi Guitart (BSC), Josep Lluis Berral (BSC), Ramon Nou (BSC), Maria A. Serrano (NBC), Michalis Dalgitsis (NBC), Carlos Segarra (IMP), Peter Pietzuch (IMP), Marc Sanchez-Artigas (URV), Pablo Gimeno (URV), Radu Stoica (IBM)
Partner(s) Contributing	BSC, DELL, IBM, IMP, NBC, TUD, URV
Document ID	CloudSkin_D2.5_Public.pdf
Abstract	Public release of stable software components of the CloudSkin toolkit. Full specification of APIs. Final description and evaluation of the project results from the validation of use cases under a variety of Cloud-edge dynamic workloads.
Keywords	CloudSkin platform; reference implementation; architectural building blocks; cloud–edge continuum; cognitive cloud; Learning Plane; AI-driven orchestration; WebAssembly; Trusted Execution Environments; instrumented storage.

History of changes

Version	Date	Author	Summary of changes
0.1	01-07-2025	Marc Sanchez-Artigas (URV)	TOC and deliverable structure.
0.5	27-11-2025	Marc Sanchez-Artigas (URV)	Architecture specs, application deployment workflow, and final functional specifications.
0.5	30-11-2025	Marc Sanchez-Artigas (URV)	Lithops description.
0.6	03-12-2025	Ardhi Putra Pratama Hartono (TUD)	SCONE architecture.
0.7	03-12-2025	Peini Liu (BSC)	Add Data-connector architecture.
0.8	03-12-2025	Carlos Segarra (IMP), Peter Pietzuch (IMP)	Add C-Cells architecture.
0.85	11-12-2025	Raul Gracia (DELL)	Add Nexus and Pravega architecture.
0.9	20-12-2025	Maria A. Serrano (NBC), Michalis Dalgitsis (NBC)	Add NearbyOne architecture.
0.95	20-12-2025	Pablo Gimeno (URV)	Add GEDS-based WebAssembly Units architecture.
0.96	22-12-2025	Radu Stoica (IBM), Marc Sanchez-Artigas (URV)	Add GEDS architecture.
1.0	22-12-2025	Marc Sanchez-Artigas (URV)	Polishing and final document.

Table of Contents

List of Abbreviations and Acronyms

AEMET	Agencia Estatal de Meteorología (State Meteorological Agency (Spain))
AI	Artificial Intelligence
AoT	Ahead-of-Time
API	Application Programming Interface
AR	Augmented Reality
AWS	Amazon Web Services
C-Cell	Cloud-edge Cell
CAS	Computer-Assisted Surgery
CC	Creative Commons
CFI	Control Flow Integrity
COS	Cloud Object Storage
CPU	Central Processing Unit
CSV	Comma-Separated Values
DHCP	Dynamic Host Configuration
DL	Deep Learning
DOI	Digital Object Identifier
DRAM	Dynamic Random-Access Memory
EC2	Elastic Compute Cloud
ECS	Elastic Container Service
EKS	Elastic Kubernetes Service
FaaS	Function-as-a-Service
FDR	False Discovery Rate
FR	Functional Requirement
GEDS	Generic Ephemeral Data Storage
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDFS	Hadoop File System
HPC	High-Performance Computing
HPDA	High-Performance Distributed Analytics
JIT	Just-in-Time
JNI	Java Native Interface

JSON	JavaScript Object Notation
K8s	Kubernetes
KER	Key Exploitable Result
KPI	Key Performance Indicator
LCM	Life-Cycle Management
LTS	Long-Term Storage
ML	Machine Learning
MPI	Message Passing Interface
MS	Imaging Mass Spectrometry
NAS	Network-Attached Storage
NAT	Network Address Translation
NDVI	Normalized Difference Vegetation Index
NFS	Network File System
NVMe	Non-Volatile Memory express
OCI	Oracle Cloud Infrastructure
OLTP	Online Transactional Processing
OOM	Out Of Memory
OS	Operating System
PAT	Port Address Translation
PoC	Proof of Concept
PVA	Predictive Video Analytics
QoS	Quality of Service
RAM	Random Access Memory
ROS	Robot Operating System
RTSP	Real-Time Streaming Protocol
S3	Simple Storage Service
SDK	Software Development Kit
SFI	Software Fault Isolation
SGX	Software Guard Extensions
SIAM	Sistema de Información Agraria de Murcia (Murcia Agricultural Information System)
SLA	Service Level Agreements
SLO	Service Level Objectives

SSD	Solid-State Drive
TCB	Trusted Computing Base
TEE	Trusted Execution Environment
TTFB	Time To First Byte
VA	Video Analytics
VM	Virtual Machine
WAL	Write-Ahead Log
Wasm, or WASM	WebAssembly
XLSX, or XLS	Excel Spreadsheet

1 Executive summary

This deliverable presents an updated and consolidated overview of the CloudSkin platform, focusing on the reference implementation of its architectural building blocks within **WP2: Architecture and Software Validation**. It provides a coherent description of the whole platform architecture and the interplay among its software components, all contributing to the realization of a **cognitive computing continuum** across heterogeneous Cloud–edge environments.

A key objective of this deliverable is to position CloudSkin within the broader European cloud–edge ecosystem. To this end, the platform architecture and its core technologies are explicitly aligned with the **EUCloudEdgeIoT** (EU-CEI) reference architecture. Specifically, CloudSkin maps its components to the **EU-CEI Building Blocks**, ensuring consistency with emerging European guidelines, fostering interoperability with other continuum platforms, and contributing validated implementations. This alignment strengthens CloudSkin sustainability beyond the project lifetime and reinforces its role as a practical instantiation of EU-CEI architectural principles.

At its core, CloudSkin implements a **composable**, **three-layer architecture** that cleanly separates concerns while enabling tight cross-layer optimization:

- The Infrastructure layer (L1) provides high-performance, tiered data management through components such as GEDS, Pravega, MinIO, and Nexus, supporting both batch-oriented and streaming workloads with low latency, elasticity, and resilience across cloud and edge sites.
- **The Execution layer (L2)** delivers a universal execution abstraction, allowing legacy code to run anywhere in the continuum. This layer combines WebAssembly-based C-Cells, Kubernetes pods, with optional confidential computing enabled through Trusted Execution Environments (TEEs) and SCONE. Advanced features like adaptive virtualization and live migration enable workloads to scale and move dynamically across heterogeneous resources.
- **The Orchestration layer (L3)** hosts the Learning Plane, the cognitive backbone of the platform. By continuously collecting telemetry and system state, training models and issuing predictions, the Learning Plane enables AI-driven orchestration, proactive resource provisioning, smart placement, and adaptive service migration across the cloud–edge continuum.

A defining characteristic of CloudSkin is its emphasis on **modularity**. All platform component are deployable on **Kubernetes**, forming a uniform substrate across cloud, edge, and IoT environments. This approach avoids monolithic designs and instead promotes independently deployable, reusable software components that can be composed into end-to-end workflows.

The deliverable also elaborates on the core integration points that unify the platform, including Kubernetes-based orchestration, object storage as a shared data layer, and a robust observability stack using `Prometheus`, `Grafana`, and `NearbyOne`. Together, these integration points enable a closed-loop control system where telemetry-driven insights inform the Learning Plane, orchestration decisions propagate across multiple sites, and workloads continuously adapt to evolving conditions

Finally, this document outlines the application deployment workflow supported by CloudSkin, demonstrating how diverse workloads—ranging from MPI applications and video analytics pipelines to batch inference workflows—are packaged, deployed, monitored, and dynamically optimized across the cloud-edge continuum. By combining multi-orchestration support, portable execution formats, unified storage abstractions, and AI-driven decision-making, CloudSkin delivers an end-to-end, smart platform capable of sustaining performance, security, and efficiency in highly dynamic cloud–edge environments.

In summary, this deliverable consolidates the architectural vision, reference implementation, and integration strategy of CloudSkin, providing a mature and standards-aligned foundation for future extensions, large-scale adoption, and long-term impact within the European cloud–edge computing landscape.

2 Summary of D2.3 CloudSkin: Architecture Specs and Early Prototypes

This section provides a concise overview of the architectural foundations and early platform insights previously delivered in the earlier architecture deliverable. Since "D2.5 Reference Implementation of Architectural Building Blocks" builds upon that work, this summary revisits the essential elements that shaped the design and validation strategy of the CloudSkin platform.

In short, the earlier deliverable provided the first complete description of the CloudSkin platform architecture and the interactions among its major software components. Also, it introduced the vision of a cognitive Cloud–edge continuum capable of adapting to dynamic conditions through AI-driven orchestration, unified execution environments, and instrumented storage technologies.

Core Innovations

The following innovations formed the conceptual basis for the platform:

- **IN1 – Cognitive Learning Plane.** Introduction of novel AI/ML-based techniques to optimize workloads, resources, energy, and network traffic in a "holistic" manner for a rapid adaptation to changes in application behavior and data variability. This materialized in the definition of a "Learning Plane" that, in cooperation with the application execution framework [IN2] and the Cloud continuum infrastructure [IN3], can enhance the overall orchestration of Cloud-edge resources. This plane represents the incarnation of the cognitive cloud, where decisions on the cloud and the edge are driven by the continuously obtained knowledge and awareness of the computing environment through AI, and particularly, neural networks and statistical learning, taking the challenge of enabling these methods into low-power edge devices.
- **IN2 – Universal Execution Abstraction.** A unified and secure execution layer enabling "stack identicality" across the continuum. By leveraging WebAssembly (Wasm) [1] along with Trusted Execution Environments (TEEs), the platform allows traditional HPC and Cloud software stacks (e.g., MPI, OpenMP) to run seamlessly on remote edges with near-native performance and strong data-in-use protection.
- **IN3 – Instrumented Storage Infrastructure.** Definition and development of a first set of storage-level abstractions, observability and dynamic configuration hooks to support complex Cloud-edge scenarios that require real-time performance, adaptive data-tiering, and improved fault-tolerance, all driven by the Learning Plane [IN1].

Platform Validation and Use Cases

The former deliverable also described the initial validation strategy, including:

- Early Proof-of-Concept (PoC) prototypes for the four representative use cases: **5G automotive**, **metabolomics**, **computer-assisted surgery**, and **agriculture**;
- Cloud-edge experimentation environments and first testbed results;
- A reference set of Key Performance Indicator (KPIs) targeting performance equivalence with Cloud-only execution, important reduction of Cloud offloading, real-time analytics, accelerated serverless processing, secure automated TEE deployment, microsecond-scale data access, and low-overhead data-tiering.

Role of this Summary in D2.5

The insights and architectural principles summarized here establish the foundation upon which the present deliverable D2.5 builds. While the earlier deliverable report focused on conceptual, partially developed architecture, preliminary prototypes, and initial validation criteria, D2.5 moves toward the final realization of these architectural building blocks, providing their reference implementation and demonstrating their maturity within the CloudSkin platform.

This summary therefore serves as a bridge between the early architectural vision and the fully implemented components documented in this deliverable.

3 Architecture specifications

3.1 Overview

The current computing paradigm is characterized by a significant imbalance, with an estimated 80% of data processing and analysis centralized in Cloud data centers, and only 20% using edge resources. This model reinforces a strategic dependency on non-EU Cloud providers and inhibits the European digital economy from fully capitalizing on the agility, low-latency, as well as data sovereignty benefits offered by edge computing.

In response to this challenge, the CloudSkin project seeks to develop a cognitive platform for the cloud-edge continuum that intelligently orchestrates heterogeneous resources. By leveraging AI, the platform will autonomously optimize the allocation of workloads, ensuring they are processed in the most efficient and context-appropriate location, whether in the core cloud or at the network edge.

Though promising, this approach introduces unique challenges for developers and infrastructure providers. Unlike clouds, edge computing lacks standardized development guidelines and essential services, such as resource managers, universal execution abstractions, scalable storage, and Cloud-edge workflow management. Consequently, developers must independently decide how to manage resources, split workloads, and offload tasks from cloud to edge, a complex design space with many options. This leads to the following key architectural insight:

> Traditional task-offloading models are often treated in **isolation**, even though they address overlapping challenges. Viewing cloud and edge computing as part of a **unified** compute continuum frees developers from the constraints of siloed paradigms.

Achieving this objective requires a concerted, multidisciplinary effort. The CloudSkin platform aims to **break the silos** of isolated computing models by providing a reference implementation of the **cognitive continuum**.

3.2 Means of verification

The three innovations in the project will be validated in real settings using the general KPIs defined in Table 1. Each use case contribute other specific KPIs, but the above KPIs constitute a representative reference validation platform (RVP) for the project.

3.3 Global Architecture

The EU-CEI (`EUCloudEdgeIoT.eu`) initiative defines a reference architecture intended to serve as a standard for the computing continuum. Specifically, EU-CEI identifies eight categories referred to as **Building Blocks** (BBs), which represent the fundamental technical capabilities required to operate applications across the continuum [2].

CloudSkin, as an active member of the EU-CEI initiative, shares its core motivations and goals. In the final stages of the project, it has become increasingly important to align and map CloudSkin architectural components and technologies onto the EU-CEI reference architecture. The main reason is that this mapping ensures coherence with European guidelines, strengthens interoperability and compatibility across European continuum platforms, and contributes to the broader standardization efforts within the EU computing ecosystem.

The aim of this mapping is twofold. First, it represents a solid step toward positioning CloudSkin within emerging European standardization efforts after the project ends. Second, CloudSkin aims to contribute directly to the EU-CEI initiative by providing:

- Concrete implementations of the Building Blocks (BB) based on real testbeds that demonstrate their use; and
- Refinements of these BBs informed by the technological advances in the project.

Table 2 presents a summary of this mapping effort. In CloudSkin, infrastructure, orchestration, and intelligence are designed to be loosely integrated, minimizing the overlap of EU-CEI Building

Table 1: Primary KPIs for validating the CloudSkin platform.

Means of verification & KPIs
• KPI1: Delivering equivalent performance of instrumented Cloud-edge programs compared with centralized Cloud ($\approx$ 1X performance). • KPI2: Reduction of cloud offloading ($> 50\%$), while amortizing edge resources and saving communication bandwidth. • KPI3: Achieving real-time processing in edge data analytics, at least in one use case. • KPI4: Cloud-edge cells startup times at least 10% faster than containers. • KPI5: Execution of complex software stacks such as MPI, and OpenMP "as is" with Cloud-edge cells at close to native speeds despite virtualization ($\approx$ 1X performance). • KPI6: Automatic conversion of legacy applications into confidential TEE-enabled Cloud-edge cells (zero development effort). • KPI7: Microsecond data access latency despite virtualization and adaptive scaling. • KPI8: Automated data-tiering and allocation with very low impact on performance (<1%). • KPI9: At least 2x acceleration of workload processing with serverless computing. • KPI10: Transparent migration of execution contexts and application state across Cloud-edge sites with $< 10\%$ service disruption. • KPI11: Sustained low-latency streaming (sub-50 ms) with high-throughput ingestion for edge video/IoT pipelines. • KPI12: Successful orchestration and monitoring of heterogeneous serverless, containerized, and WebAssembly workloads via a unified interface. • KPI13: Coordinated multi-site orchestration with consistent lifecycle management across clusters, including deployment, migration, and policy enforcement. • KPI14: In-transit data transformations, buffering, semantic annotation, and intelligent routing with $< 5\%$ added latency to the original storage operations.

Blocks (BBs) across multiple technological layers. Nonetheless, some BBs inherently span technological layers. For instance, resource management and application orchestration operate across levels. Typically, Kubernetes manages low-level container orchestration, while the Learning Plane, together with AI-based BBs, drives high-level decisions on task placement, workload migration, and resource allocation across the Cloud-edge continuum.

This design pattern is consistently applied in other platform components. For instance, Lithops Serve leverages Kubernetes to orchestrate the deployment of confidential inference containers, while the Learning Plane dynamically determines the appropriate number of containers to provision. By combining low-level container orchestration with high-level AI-driven resource provisioning, Lithops Serve can efficiently scale batch inference workloads across the Cloud-edge continuum, optimizing resource utilization and maintaining service quality.

Table 2: Mapping of EU-CEI Building Blocks to CloudSkin implementation

EU-CEI Building Blocks	**CloudSkin Implementation**
Security and Privacy	Built-in infrastructure mechanisms. CloudSkin leverages confidential containers through SCONE to allow standard Docker applications (Python, Java, etc.) to execute in SGX enclaves without source-code modifications, providing hardware-enforced isolation and per-container memory encryption keys. As part of this BB, the project combines confidential containers with WebAssembly to implement secure C-Cells, memory-safe, lightweight, and portable sandboxed applications with a controlled interface to OS services, allowing unmodified or legacy applications to run securely while supporting encryption, attestation, and filtered communications.
Trust and Reputation	CloudSkin enhances trust and reputation in the continuum by leveraging confidential containers through SCONE and secure C-Cells based on WebAssembly. This ensures that users and infrastructure providers can trust that applications execute securely and reliably, while sensitive data remains protected, supporting accountability and reputation management across federated and multi-stakeholder environments.
Data Management	Adaptive and optimized management of ephemeral and persistent data across the continuum. **L1 [Infrastructure layer]** provides efficient storage abstractions to support diverse workloads, including monolithic applications, microservices, and streaming computations, ensuring low-latency I/O for performance-critical tasks.
Resource Management	Kubernetes manages the low-level orchestration of standard containerized workloads, while CloudSkin leverages additional fine-grained runtime orchestrators for C-Cells to efficiently handle parallel applications implemented with multi-threading (e.g., OpenMP) or multi-processing (e.g., MPI). These low-level orchestrators address limitations of traditional cloud schedulers, which cannot dynamically scale multi-threaded applications or manage fragmentation caused by statically allocating multi-process applications to virtual machines.
Orchestration	The AI-enabled orchestration layer (L3) identifies optimal provisioning, placement, and partitioning strategies between Cloud and edge. At its core, the **Learning Plane** extracts knowledge from continuum components to provide recommendations, predictions, and inferred information for global system optimization, including task scheduling, resource allocation, and storage orchestration. By combining Kubernetes, specialized AI and low-level WebAssembly orchestrators, and the Learning Plane, CloudSkin delivers a multi-layer orchestration framework capable of managing heterogeneous resources and diverse workload types, from containerized applications to parallel HPC-style tasks, with fine-grained elasticity and global system optimization. The project leverages multi-cluster orchestration platforms, combining Kubernetes with high-level service orchestrators (e.g., NearbyOne Orchestrator) and monitoring stacks for closed-loop automation.

Continued on next page

Table 2 – continued from previous page

EU-CEI Building Blocks	CloudSkin Implementation
Network	CloudSkin adopts a multi-layer infrastructure with uniform interfaces and protocols to enable seamless workload balancing at runtime. Further, CloudSkin enhances network support by providing dynamic service discoverability across multi-site infrastructures. Services are registered with a DNS-based solution, enabling seamless migration between cloud and edge nodes without manual reconfiguration. Routing rules and service endpoints are automatically updated by the orchestrator during migrations, ensuring low-latency connectivity. The system supports mobility and dynamic deployment scenarios, allowing applications to move across heterogeneous environments while maintaining availability and network efficiency. This approach reduces service downtime and improves user experience during runtime migrations.
Monitoring and Observability	CloudSkin implements a multi-layer observability stack to monitor containerized services across the Cloud-Edge Continuum. The stack integrates Prometheus, Thanos, Grafana, and MinIO for scalable multi-cluster monitoring. Observer and Observee blocks collect and aggregate metrics from edge and cloud clusters, supporting long-term storage and global queries. This enables real-time insight into system health, workload performance, and connectivity. Observability data feeds the **Learning Plane** for intelligent decision-making, enabling closed-loop automation, anomaly detection, and predictive management of service migrations.
Artificial Intelligence (AI)	The **Learning Plane** embeds AI-driven intelligence across L3 [**orchestration**], L2 [**execution**], and L1 [**infrastructure**] layers to enable proactive and adaptive management of the Cloud-Edge Continuum. It continuously collects telemetry data from edge and cloud nodes, including application performance, resource usage, and network metrics, and feeds this into ML-based models for workload characterization, anomaly detection, and QoS prediction. Leveraging predictive analytics, the engine supports proactive service migration, dynamic scaling, and workload redistribution to optimize latency, throughput, and resource utilization. By integrating with orchestrators and the Learning Plane, it enables closed-loop automation and decision-making, ensuring that services are deployed, migrated, and managed efficiently across heterogeneous environments while maintaining security, reliability, and QoS.
Execution[1]	This new building block highlights the novelty of **C-Cells**, a lightweight, portable, and adaptive WebAssembly-based virtualization environment. C-Cells enable secure execution of unmodified applications across heterogeneous Cloud and edge nodes, transparently leveraging hardware isolation (e.g., Intel SGX) or acceleration where available, and supporting rapid scale-up/down of compute units.

Additional BBs. Importantly, CloudSkin adds a crucial missing Building Block: **Execution**. Any computing-continuum platform must provide a flexible execution environment that allows applications to run across the continuum with minimal modification. This is highlighted in italics in Table 2.

In CloudSkin, this is exemplified by two **WebAssembly**-powered platforms: C-Cells, providing lightweight execution for continuum applications, and GEDS, enabling an independent computing layer for specialized workloads:

- **Secure C-Cells** provides a protected runtime for Wasm applications using Intel SGX TEEs. The two-way sandbox ensures that: 1. Wasm protects co-resident applications from unauthorized code or data access;

[1]Indicates a new BB introduced in CloudSkin.

and 2. SGX enclaves protect applications from tampering or data leaks. The OS is considered potentially hostile, while enclave memory is encrypted and isolated.

- **GEDS** provides an integration layer that enables Wasm code to leverage advanced storage features. By extending the runtime interface beyond the minimal WebAssembly System Interface (WASI) capabilities, GEDS allows applications to access persistent volumes, cloud storage, and OS-level I/O transparently, without requiring modifications to the Wasm code. This ensures legacy and new applications can utilize rich storage services across the continuum.

We remember here that WASI provides a **uniform API** across OSs, delivering virtualization, sandboxing, and access control, with capabilities for file systems, networking, time, event polling, and randomness. To turn Wasm into a truly seamless, cross-platform runtime spanning cloud and edge servers, and IoT devices, extra enhancements are needed, **which is exactly the gap that CloudSkin fills**. Fig. 1 provides a mapping between BBs to CloudSkin layers (L1–L3).

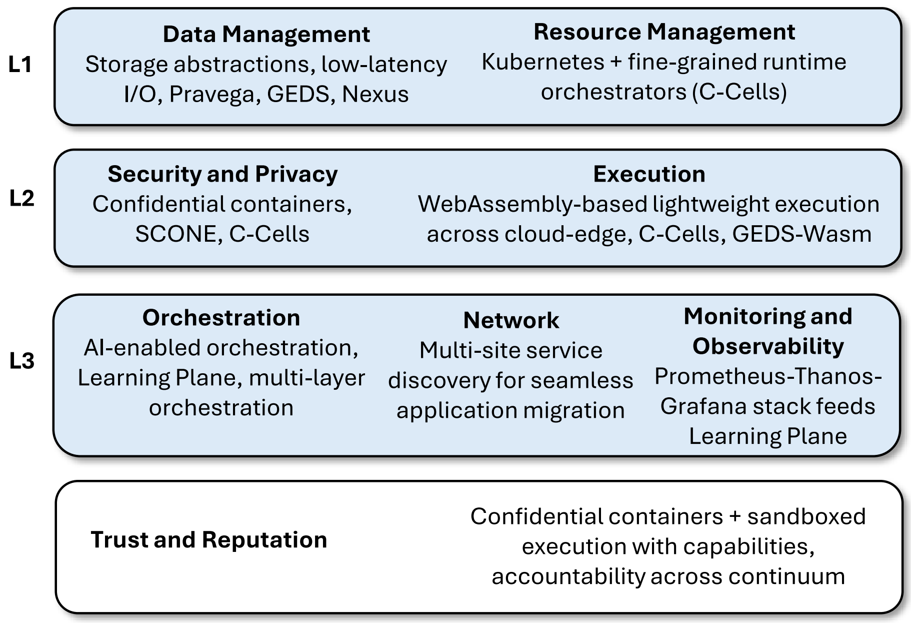
L1
Data Management
Storage abstractions, low-latency I/O, Pravega, GEDS, Nexus
Resource Management
Kubernetes + fine-grained runtime orchestrators (C-Cells)
L2
Security and Privacy
Confidential containers, SCONE, C-Cells
Execution
WebAssembly-based lightweight execution across cloud-edge, C-Cells, GEDS-Wasm
L3
Orchestration
AI-enabled orchestration, Learning Plane, multi-layer orchestration
Network
Multi-site service discovery for seamless application migration
Monitoring and Observability
Prometheus-Thanos-Grafana stack feeds Learning Plane
Trust and Reputation
Confidential containers + sandboxed execution with capabilities, accountability across continuum

Figure 1: Mapping of building blocks (BBs) to CloudSkin layers and key functions across the cloud-edge continuum, highlighting infrastructure, execution, orchestration, and cross-layer capabilities.

3.3.1 Architecture overview.

As introduced in D2.3, the CloudSkin platform implements a **composable** and **three-layered**cloud continuum platform, integrating heterogeneous, federated, and collaborative computing resources. Its final architecture is illustrated in Figure 2, highlighting how infrastructure, execution, and orchestration layers interact to support dynamic service deployment and management. We provide a short recap of their responsibilities:

- **L3. Orchestration layer (AI-enabled Orchestration BB).** CloudSkin orchestration layer leverages the "Learning Plane" to make AI-driven decisions on resource provisioning, placement, and partitioning across Cloud and edge. It extracts insights from system components to provide recommendations and predictions for global optimization, including resource allocation, containerization, virtualization, and storage management.
- **L2. Execution layer (Execution & Security BBs).** This tier includes a universal execution environment, enabling unmodified applications to run anywhere in the Cloud-edge continuum. WebAssembly-based C-Cells and hardware support such as TEEs (e.g., Intel SGX) enable confidential processing, hardware acceleration, and fast, lightweight execution for diverse workloads.

Figure 2: Layered Architecture for the CloudSkin platform.

- L1. **Infrastructure layer (Infrastructure & Data Management BBs).** The infrastructure layer delivers high-performance storage abstractions, supporting monolithic applications, microservices, as well as streaming tasks. Efficient low-latency I/O ensures that fast-executing tasks in L2 are not bottlenecked, preserving performance across the Cloud-edge continuum.

As one of the most common software architectures, a layered approach introduces many benefits. First, it provides a clear separation of concerns, where each software layer performs a specific role. It also promotes the isolation of changes, meaning that future modifications to the implementation of a layer will not impact other layers, as long as the APIs between them are adhered to. For instance, Learning Plane agents can directly call the Kubernetes API to launch pods with specific configurations, enabling intelligent, automated deployment and scaling of workloads without affecting the underlying infrastructure or orchestration logic.

3.3.2 Orchestration (layer L3)

As illustrated in Figure 2, at the top lies the **L3. Orchestration layer**, which contains the **Learning Plane**: the intelligence core of the CloudSkin platform. This layer is responsible for a range of AI-driven functions:

- Collecting state and telemetry data,
- Modeling the system,
- Managing a model catalog, and
- Delivering recommendations, predictions, and forecasts.

The **Learning Plane** employs **sensors**, **actuators**, and predictive pipelines to anticipate changes in service QoS, trigger proactive application migrations, and optimize task placement and resource allocation. By combining continuous monitoring, intelligence, and multi-site orchestration, the L3 layer delivers a holistic, adaptive, and autonomous control plane capable of orchestrating diverse workloads across the CloudSkin ecosystem.

The complete list of components in this layer is:

- **Kubernetes** (Open-source, CNCF): A popular container orchestration platform designed to automate deployment, scaling, and lifecycle management of containerized applications. It provides abstractions for pods, services, and workloads. Kubernetes is foundational for hybrid and multi-cloud deployments, offering extensibility via custom resource definitions and operators for domain-specific automation.

- **NearbyOne** (Open-source, ETSI; Proprietary, Nearby Computing): An orchestration, automation, and lifecycle management platform purpose-built for distributed edge-to-cloud environments. It delivers unified orchestration capabilities spanning infrastructure, connectivity, and application layers, operating over specialized functional components. By creating tight integration between domains that are typically siloed, NearbyOne enables advanced cross-layer orchestration, improving performance, efficiency, and service innovation. Its single-pane-of-glass interface allows operators to rapidly provision edge nodes, allocate resources, and manage their complete lifecycle as a service.

- **Lithops** (Open-source, Apache): Python-based multi-cloud serverless computing framework enabling transparent execution of massively parallel functions and data analytics across diverse backends. `Lithops` abstracts cloud compute (FaaS, VMs, containers) and storage systems (object stores, file systems) into a unified programming model, allowing developers to write regular Python code that executes remotely at scale. Its modular architecture supports major clouds and container platforms, low operational costs, and efficient parallel data processing.

3.3.3 Execution (layer L2)

In the **L2. Execution layer** lies the worker machines that execute the (Wasm-)containerized applications in forms of **C-Cells**, serverless functions, and **Kubernetes** pods. One important feature of the virtualization layer is ① **Live migration** support: C-Cell execution must be able to be interrupted and transferred from one host to another across the heterogeneous Cloud-edge continuum with no (or little) disruption to the application execution. Another key feature is ② **Adaptive virtualization**, which means that optionally and transparently, depending on the data being processed, C-Cells must support confidential execution with TEEs [3] (e.g., Intel SGX [4]). The CloudSkin platform uses **SCONE** [5] for this aim.

The complete list of components in this layer is:

- **SCONE** (Community Edition, SCONTAIN): Open-source confidential computing platform that enables secure execution of sensitive applications inside containers using Trusted Execution Environments (TEEs). SCONE provides transparent encryption, attestation, and isolated runtime environments that protect data and code throughout their lifecycle, even in untrusted cloud infrastructures.
- **C-Cells** (Open-source, Apache): Distributed WebAssembly-based execution units designed to support elastic and portable execution of parallel applications across cloud and edge clusters. More specifically, C-Cells units can be dynamically created, scaled, and migrated at runtime. They enable vertical scaling by adding more execution units to multi-threaded workloads within a node, and horizontal scaling by migrating multi-process workloads across nodes without restarting the application. Fast snapshotting and WebAssembly-based isolation allow new C-Cells to be launched efficiently, enabling responsive adaptation to resource availability and minimizing fragmentation.
- **GEDS WebAssembly-based Units** (Open-source, Apache): WebAssembly execution runtime integrated within the Generic Ephemeral Data Store (GEDS) in layer L1 to enable high-performance, sandboxed data processing close to the storage layer. While WebAssembly ensures portability and minimal memory overhead, its standard I/O interface is limited to basic POSIX-like operations, restricting I/O to the local file system. GEDS WebAssembly-based units address this limitation by interposing on standard I/O hostcalls, directing read and write operations to GEDS for in-memory storage, with automatic tiering to long-term storage.

3.3.4 Infrastructure (layer L1)

In the **L1. infrastructure layer** can be found a number of storage services to keep up with I/O-intensive applications (e.g., Computer-Assisted Surgery). This includes ① **Streaming** workloads, where data streams must be durable, consistent, and elastic, but also ② **Batch** jobs with performance critical I/O operations, such as data shuffling or sharing of data between tasks. As depicted in Fig. 2, CloudSkin handles both types of workloads with the help of **Pravega** streams, IBM **GEDS**, **MinIO** storage services.

The complete list of components in this layer is:

- **GEDS** (Open-source, Apache): A fast, distributed, and multi-tiered data store specifically designed for managing ephemeral and intermediate data across cloud and edge environments. GEDS provides low-latency in-memory storage for transient datasets, while supporting automatic tiering to durable object storage for longer-term retention. Its architecture enables efficient sharing of data between containers, WebAssembly modules, and serverless functions, making it a key enabler for in-situ data processing and ephemeral workflows.
- **Pravega** (Open-source, CNCF): A distributed streaming storage system that allows applications to store unbounded sequences of bytes durably and elastically. Pravega streams provide high-throughput, low-latency access to continuously generated data, while interfacing with GEDS to tier data to long-term storage (LTE). Its design supports streaming analytics, real-time event processing, and video pipelines, ensuring both scalability and durability across heterogeneous cloud and edge sites.
- **Nexus** (Open.source): Nexus is a programmable, policy-driven framework that enhances tiered streaming storage in the continuum by adding value to data in transit. Classical tiered storage offloads cold data as opaque chunks. However, Nexus introduces buffering at the edge to mask network failures, semantic annotation, data transformations, and intelligent routing based on performance, privacy, or cost policies.

  By exploiting the computational slack at chunk boundaries, Nexus performs these operations without affecting real-time ingestion, enabling more reliable, efficient, and insightful data pipelines, particularly for latency-sensitive, data-intensive applications such as surgical analytics and AI model training.

3.4 Key Platform Integration Points

This section defines the **minimal integration points** that enable seamless interoperability across the CloudSkin platform. It focuses on the essential connections between the core components, including the **orchestration** layer, **execution** environments, **Learning Plane**, and underlying **infrastructure**, without prescribing detailed implementation specifics.

By establishing clear interfaces and standardized communication patterns, these integration points ensure that diverse technologies, ranging from containerized workloads and WebAssembly-based execution units to AI-driven orchestration agents can operate cohesively. The goal is to provide a lightweight yet robust blueprint that allows the platform to coordinate tasks, manage resources, and enable intelligent service migration across the cloud-edge continuum while preserving modularity and extensibility.

The integration points are the following:

1. Kubernetes. CloudSkin leverages Kubernetes as the central container management platform, orchestrating the deployment, scaling, and operation of both traditional and **confidential containers**. For C-Cells, we have ensured that all system components for running them are fully deployable with Kubernetes. The reason is that C-Cells has a proprietary **live migration protocol** that is not compatible with the Kubernetes API. Further, the **Learning Plane** and and **storage** integrations have been made fully deployable with Kubernetes in order to make the entire platform consistently manageable across cloud, edge, and IoT nodes.

As part of ongoing development, we are integrating GEDS Wasm-based containers with Kubernetes as a replacement of traditional containers. In this setup, `containerd` serves as the high-level runtime, leveraging lightweight shim processes to decouple container execution from the daemon, thereby ensuring reliability and seamless upgrades. The ultimate goal of this integration is to enable WebAssembly applications to fully benefit from the same lifecycle management, scheduling, and orchestration capabilities as traditional containers. This effort includes the packaging of WebAssembly applications as **OCI-compliant images**[2] in a container registry, which allows Kubernetes to pull, instantiate, and scale them across the continuum.

2. Object Storage. An essential integration point within the CloudSkin platform is the use of object storage as a universal communication and data exchange layer. Both C-Cells and Wasm-based execution units leverage object storage to interact with the rest of the platform components and with each other. This is achieved either directly via MinIO[3], which provides a lightweight, S3-compatible, and Kubernetes-friendly storage backend, or through GEDS integration for WebAssembly units, enabling advanced storage features transparently.

The same pattern extends to streaming applications: Pravega, integrated with GEDS, supports automatic data tiering to S3 for cloud tasks that require exchanging large datasets or transient results. By centralizing both batch and streaming data interactions through object storage, the platform ensures efficient, consistent, and portable access to information across the cloud-edge continuum, while simplifying workflow orchestration and data sharing between heterogeneous execution environments.

For instance, the GEDS S3/MinIO Endpoint provides essential primitives to interact with object storage in CloudSkin. These operations are used by GEDS WebAssembly-based units to communicate and exchange data:

- **PUT Object** Uploads data to a specified bucket and key. Supports both in-memory buffers and input streams, with optional length specification:
  - `putObject(bucket, key, const uint8_t *bytes, size_t length)`
  - `putObject(bucket, key, std::shared_ptr<std::iostream> stream, std::optional<size_t> length)`
- **GET Object** Reads data from a bucket and key. Supports direct memory access or streaming to an output stream, with optional position and length for partial reads:
  - `readBytes(bucket, key, uint8_t *bytes, size_t position, size_t length)`
  - `read(bucket, key, std::iostream &outputStream, std::optional<size_t> position, std::optional<size_t> length)`

These primitives provide the foundational interface for object storage operations, enabling persistent and streaming data exchanges across the CloudSkin platform.

[2] https://tag-runtime.cncf.io/wgs/wasm/deliverables/wasm-oci-artifact/
[3] https://www.min.io/

3. Observability. Following modern DevOps principles, observability is a **first-class aspect** of the platform, providing continuous, actionable insights into system behavior, and enabling the **Learning Plane** to respond rapidly and intelligently. In particular, `Prometheus`[4] acts as the primary time-series engine, collecting metrics from all the CloudSkin services, orchestration components, and Kubernetes workloads. `Thanos`[5] extends these capabilities with global querying and durable storage, ensuring a consistent view of the entire platform.

On top of this, CloudSkin incorporates specialized `Grafana`[6] **dashboards** tailored to each subsystem. These dashboards visualize the health, performance, and runtime dynamics of applications, resource provisioning policies, and WebAssembly-based execution environments in real time. To put it in a nutshell, the combination of `Prometheus` metrics, `Thanos` federated storage, and `Grafana` analytics provides a comprehensive end-to-end observability stack that supports operators, enables debugging and optimization, and empowers the **Learning Plane** to perform intelligent, data-driven orchestration across the cloud–edge continuum.

NearbyOne Observability Stack. CloudSkin also leverages the NearbyOne Observability Stack to provide seamless monitoring across multiple clusters and deployment tiers. This stack brings together `Prometheus`, `Thanos`, `Grafana`, and MinIO into a scalable and fault-tolerant observability layer designed for heterogeneous cloud–edge clusters. To achieve this, Observer and Observee Blocks are deployed across the continuum: Observee Blocks expose per-cluster metrics, whereas the Observer Block aggregates the metrics, ensures long-term retention, and enables unified cross-site queries. This architecture ensures efficient monitoring even under tight edge-resource constraints and provides the telemetry backbone required for proactive decision-making in the Learning Plane, as demonstrated in the automotive use case.

Table 3: CloudSkin Key Integration Points.

Integration Point	**Description**
Kubernetes	CloudSkin leverages Kubernetes as the central container management platform to orchestrate deployment, scaling, and operation of traditional containers. All system components, including the Learning Plane, C-Cells and storage integration, are fully deployable with Kubernetes, ensuring consistent management across cloud, edge, and IoT nodes. GEDS Wasm-based containers are integrated to benefit from container lifecycle management, scheduling, and orchestration capabilities, packaged as OCI-compliant images for deployment across the continuum.
Object Storage	Object storage acts as a universal communication and data exchange layer. C-Cells and Wasm-based execution units interact with each other and platform components via MinIO or GEDS integration. Streaming applications, such as Pravega, leverage automatic object storage data tiering for large datasets. GEDS S3 and MinIO endpoint primitives, including `putObject` and `read/readBytes`, provide foundational interfaces for persistent and streaming data exchanges across the CloudSkin platform.
Observability	CloudSkin employs the NearbyOne Observability Stack integrating `Prometheus`, `Thanos`, and `Grafana` for scalable, multi-cluster monitoring. Observee Blocks expose per-cluster metrics while the Observer Block aggregates metrics, stores long-term data, and enables unified cross-site queries. Specialized `Grafana` dashboards visualize the health and performance of each subsystem. This design ensures efficient monitoring under constrained edge resources and provides a telemetry backbone for proactive Learning Plane-driven orchestration.

3.4.1 Composable Software Platform.

With all system elements, from C-Cells to Learning Plane services and GEDS-backed storage adapters, being deployable on Kubernetes, CloudSkin evolves into a **fully composable software platform**. Instead of relying

[4]`https://prometheus.io/`
[5]`https://thanos.io/`
[6]`https://grafana.com/`

on monolithic deployments or tight integration boundaries, CloudSkin structures its core functionality around **modular software components** that can be independently packaged, versioned, deployed, and orchestrated. Kubernetes provides the uniform substrate where these components coexist, interact, and scale, whether they are traditional containers, confidential containers, or WebAssembly-based execution units.

By enabling components to be reused and composed across different data flows and computational paths, this architecture ensures that teams can innovate independently while still benefiting from shared abstractions and consistent implementations. The platform thereby achieves a balance between autonomy and coherence: individual components can evolve at their own pace, yet they integrate seamlessly into higher-level workflows. This promotes development efficiency, simplifies system evolution, and increases agility when adapting to new operational requirements or deploying the platform across diverse environments.

In this model, composability becomes a foundational principle. Each software component can be combined into end-to-end services without altering the underlying deployment or operational model. Kubernetes acts as the unifying layer that coordinates these components across cloud, edge, and IoT devices, ensuring reliable scheduling, scaling, and lifecycle management.

As GEDS WebAssembly containers become fully integrated into this ecosystem, CloudSkin expands its composability even further, enabling lightweight, portable execution units to participate alongside container-based components within a single, cohesive platform.

4 Application Deployment Workflow in CloudSkin

The CloudSkin platform must support a highly diverse compute and orchestration landscape, spanning from large centralized clouds to mobile edge locations and embedded devices. The mobility use case exemplifies this heterogeneity: services must execute across edge sites, and private clouds, all while maintaining strict QoS guarantees for video analytics and RAN control loops.

In the continuum, depending on a single orchestrator or a single execution model is neither feasible nor desirable. Instead, CloudSkin embraces a multi-orchestrator, multi-runtime approach. Despite CloudSkin uses Kubernetes as its central container management platform, it often composes a hierarchy of orchestrators, with Kubernetes serving as the foundational layer beneath higher-level workflow and multi-site orchestrators. As discussed earlier, these are:

- `Lithops`[7] lets developers deploy tasks as serverless functions in the cloud, or run the same code inside containers on Kubernetes clusters when working on-prem or at the edge.
- `NearbyOne` [6] acts as a multi-site orchestration layer: it manages application and service lifecycles across many edge and cloud locations, relying beneath the hood on Kubernetes to deploy and run workloads.

In this way, CloudSkin combines the strengths of container orchestration through Kubernetes with the flexibility of serverless and edge-aware multi-site orchestration via `Lithops` and `NearbyOne`, enabling a unified, scalable, and portable deployment framework.This is shown in Figure 3.

Despite the diversity of orchestrators and runtimes, the CloudSkin platform is able to adapt to different execution workflows by enabling developrs to compose its system components. This composable architecture ensures that tasks can be deployed, managed, and optimized seamlessly across heterogeneous Cloud-edge environments. A typical workflow consists of several major phases, which we describe in detail below. A rapid overview of a typical execution workflow is given in Figure 4.

4.1 Application Packaging Into Containers, WebAssembly Modules, or Functions

The first phase involves transforming an application into an artifact that can be deployed anywhere across the continuum. CloudSkin supports three primary packaging formats:

- **WebAssembly modules:** For lightweight execution or sandboxed environments, CloudSkin supports WebAssembly, ideal for C++/Rust workloads or scenarios where portability and isolation are of paramount importance. Through the C-Cells subsystem, WebAssembly runtimes support monolithic binaries (e.g., OpenMP and MPI programs), enabling HPC-style code to run safely at the edge.
- **OCI-compliant containers:** Containers encapsulate all runtime dependencies, such as libraries and VMs for Python, Java, etc., providing a stable execution environment across heterogeneous infrastructures. For instance, NCT surgical models run as containers.
- **Serverless functions:** Lightweight tasks packaged as functions (e.g., Python or JavaScript) are supported through FaaS frameworks such as `Lithops`, which are especially relevant for event-driven and batch pipelines.

[7] `https://github.com/lithops-cloud/lithops`

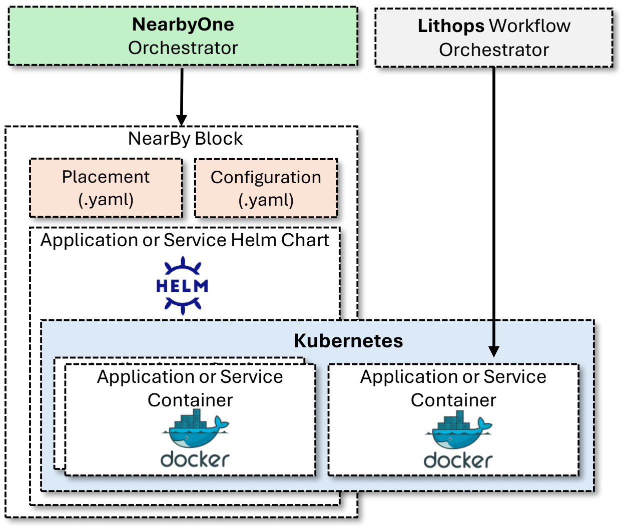
NearbyOne Orchestrator
Lithops Workflow Orchestrator
NearBy Block
Placement (.yaml)
Configuration (.yaml)
Application or Service Helm Chart
HELM
Kubernetes
Application or Service Container
docker
Application or Service Container
docker

Figure 3: CloudSkin orchestration: Kubernetes as the runtime substrate; `Lithops` and `NearbyOne` as higher-level orchestration layers for serverless, edge, and multi-site deployments.

Each packaging format has their corresponding runtimes provided by CloudSkin, ensuring safe execution regardless of where the workload is ultimately deployed. The use of open standards such as OCI for containers, WASI for WebAssembly modules, and cloud-agnostic APIs for serverless functions, guarantees portability across all orchestrators in the platform.

Alternatively, WebAssembly modules can be packaged as **OCI-compliant artifacts**[8] in a container registry. These artifacts abide by the standard OCI image layout and consist of multiple layers: a configuration layer specifying the entrypoint (typically a `.wasm` file), the Wasm module layer itself, and optional additional layers containing configuration files, libraries, or other resources. Unlike traditional Docker images, the `manifest` file is primarily used to reference these layers and ensure compatibility with OCI-compliant tooling. By pushing these OCI artifacts in a registry, Kubernetes or other OCI-aware orchestrators can pull, instantiate, and scale WebAssembly modules consistently across the continuum. This approach is the one adopted to manage GEDS WebAssembly containers. An example of `manifest` is:

Listing 1: Example of OCI-compliant Wasm image

```json
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.oci.image.index.v1+json",
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "digest": "sha256:485ee0f459ed27244372efd50eadf92011201f5a8ae5daf72cc0f829f2cb8a90",
      "size": 707,
      "annotations": {
        "io.containerd.image.name": "ghcr.io/cloudskin/wasi-helloworld:latest",
        "org.opencontainers.image.ref.name": "latest"
      },
      "platform": {
        "architecture": "wasm",
        "os": "wasip1"
      }
    }
  ]
}
```

This also encompasses SCONE-enabled confidential containers, allowing seamless deployment using pod

[8]`https://tag-runtime.cncf.io/wgs/wasm/deliverables/wasm-oci-artifact/`

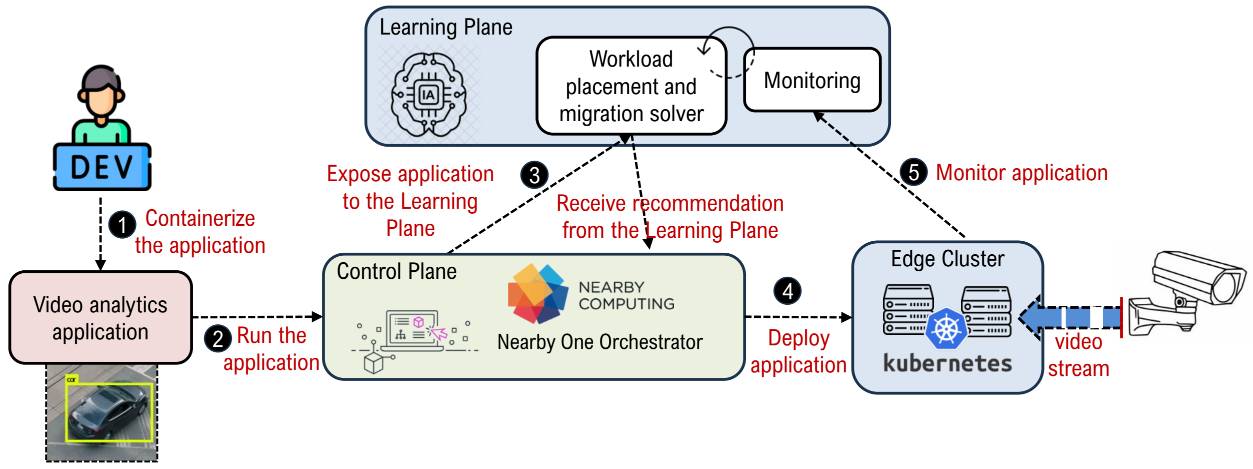
Learning Plane
Workload placement and migration solver
Monitoring
DEV
1 Containerize the application
Video analytics application
2 Run the application
Expose application to the Learning Plane
3
Receive recommendation from the Learning Plane
Control Plane
NEARBY COMPUTING
Nearby One Orchestrator
4 Deploy application
5 Monitor application
Edge Cluster
kubernetes
video stream

Figure 4: Execution workflow for a video analytics application on CloudSkin.

`.yaml` specifications:

Listing 2: SGX-enabled test pod

```
apiVersion: v1
kind: Pod
metadata:
  name: testpod
spec:
  imagePullSecrets:
    - name: sconeapps
  containers:
    - name: testcon
      image: registry.scontain.com/user/cloudskin-images/test
      env:
        - name: SCONE_MODE
          value: SIM
      securityContext:
        capabilities:
          add: ["SYS_RAWIO"]
```

This pod leverages the SGX simulation mode (`SCONE_MODE: SIM`) and requires elevated capabilities (`SYS_RAWIO`) to properly emulate trusted execution environments. The `imagePullSecrets` entry ensures access to private SGX-enabled container images stored in the secure registry.

4.2 Submission to a CloudSkin-Compatible Orchestrator

After packaging, the developer delegates code execution to the appropriate orchestration layer or service. For instance, the developer can rely on `Lithops` to run a sequence of tasks, whether packaged as container images or deployed as serverless functions in AWS Lambda, or interact directly with the C-Cells distributed runtime through its native interfaces such as `faasmctl` to upload the `.wasm` files from Docker container images. Each orchestrator exposes its own execution abstraction: Kubernetes manages containerized workloads through declarative manifests, AWS-style FaaS platforms execute function artifacts, etc.

Although highly valuable, providing a unified interface for all type of computations would be extremely challenging, as it requires reconciling fundamentally different execution models, native APIs, and scheduling semantics. Managing serverless functions, containerized workloads, and WebAssembly modules into a single unified interface can introduce complexity, increase the risk of misconfigurations, and make debugging more difficult across cloud, edge, and IoT environments. It should be noted that `Lithops` can leverage Kubernetes as a compute backend, while `Faasm`[9], the WebAssembly-based system underpinning C-Cells, alrady provides a minimal-effort path to enable `Lithops` to orchestrate complex WebAssembly workflows.

As an illustrative example, Lithops Serve, a novel serverless batch inference engine, uses `Lithops` to launch multiple specialized instances of a Python runtime. This fleet of instances perform batch inference in parallel

[9] `https://github.com/faasm/python`

or other AI-related tasks, running as serverless functions or as containers on-premises without requiring code refactoring of any type. In this setup, `Lithops` is leveraged by a higher-level AI orchestrator to execute the functions. Interaction with Lithops Serve is provided through a simple RESTful interface, which exposes an authenticated `POST` endpoint for submitting tasks. Developers only need to code a Python script and invoke the `POST` endpoint to submit a bach inference job. An example of inference code is provided below:

Listing 3: Example of task workflow using TaskManager

```python
config_dict = {
    'load': {'batch_size': 1, 'max_concurrency': 32},
    'preprocess': {'batch_size': 32, 'num_cpus': 4},
}

manager = TaskManager(config_dict=config_dict, logging_level=logging.INFO)

@manager.task(mode="threading")
def load(image_dict):
    result_dict = {}
    for key in image_dict:
        print(f"Downloading image {key} from S3")
        image_data = s3_client.download(key, s3_bucket=BUCKET)
        print("Downloading image finished")
        result_dict.update({key: image_data})
    return result_dict

@manager.task(mode="multiprocessing", previous=load, batch_format="bytes")
def preprocess(image_dict):
    composed_transforms = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
        ),
    ])
    result_dict = {}
    for key, image_data in image_dict.items():
        print("Transformation started", key)
        image = Image.open(io.BytesIO(image_data)).convert('RGB')
        tensor = composed_transforms(image)
        result_dict.update({key: tensor})
        print("Transformation finished", key)
    return result_dict
```

Multi-Site orchestration. A major strength of the `NearbyOne` orchestrator lies in its advanced multi-site orchestration capabilities. Critical application lifecycle operations such as deployment, migration, and service discoverability can be coordinated by the `NearbyOne` service orchestrator. Deployed in a separate Kubernetes cluster, `NearbyOne` operates outside the edge–cloud continuum and functions as a multi-site, cloud-native orchestrator that manages distributed continuum applications without directly hosting workloads. To this end, it exposes a RESTful **Northbound Interface** (NBI) for external triggers, allowing systems and AIOps agents to initiate migrations and orchestration workflows.

Communication with managed clusters is handled by lightweight `NearbyOne` agents deployed within each site. These agents expose cluster capabilities, accept orchestration instructions, and enable declarative control of workloads and infrastructure components, such as containers and `.wasm` modules. This design guarantees consistent lifecycle management while preserving the autonomy of individual clusters.

These capabilities have been key to deploy the Learning Plane agents across cloud and edge sites. These agents continuously monitor telemetry, evaluate policies, make AI-driven decisions, and trigger actions via the NBI, such as migrating applications between clusters. By separating responsibilities between the orchestrator, Learning Plane agents, and individual clusters, `NearbyOne` enables modular, event-driven, and scalable multi-site orchestration across heterogeneous edge and cloud environments.

To wrap up, the fully-fledged orchestration capabilities of CloudSkin are:

- **Orchestrator Abstractions:**
  - **Kubernetes:** manages containerized workloads via declarative manifests.

  - **FaaS platforms:** execute serverless function artifacts.
  - **Lithops:** supports both cloud functions and containerized execution on Kubernetes, and optionally on other compute backends, such as IBM Cloud Code Engine, AWS Batch, Google Cloud Run, and Knative.

- **Multi-Site Orchestration with NearbyOne:**
  - Coordinates deployment, migration, and service discoverability across clusters.
  - Operates as a cloud-native orchestrator outside the edge–cloud continuum.
  - Exposes a RESTful **Northbound Interface (NBI)** for external triggers.
  - Uses lightweight agents at each site to control workloads and infrastructure declaratively.

- **Learning Plane Integration:**
  - Agents monitor telemetry, evaluate policies, and trigger orchestration actions via the NBI.
  - Supports AI-driven decisions such as dynamic application migration.
  - Ensures modular, event-driven, and scalable multi-site orchestration.

- **Unified Compatibility Layer:** Minimal northbound API interfaces across orchestrators allow distributed applications and the Learning Plane to operate across heterogeneous execution environments.

4.3 Integration with Object Storage, GEDS, and Pravega

A key enabler of portability and inter-component communication in CloudSkin is the use of object storage as a universal data abstraction. Applications, including WebAssembly modules and containers, can read and write data to object storage either directly using MinIO or through a WebAssembly-specific integration with GEDS. This provides a consistent mechanism for sharing files and datasets across edge and cloud sites.

For streaming workloads, such as video analytics pipelines, CloudSkin taps into Pravega integrated with GEDS. Pravega streams are automatically tiered to S3-compatible object storage when required, enabling large-scale, durable data exchange without blocking the application.

The Pravega-GEDS integration uses the Hadoop File Sysetm (HDFS) API as the common interface: GEDS already provides a Java library integrtaing HDFS, while Pravega supports HDFS as a long-term storage option. To enable the integration, Pravega was extended with a new `fs.hdfs.impl` parameter, allowing administrators to specify the HDFS binding class. For example, setting `fs.hdfs.impl= com.ibm.geds.hdfs.GEDSHadoopFileSystem` enables Pravega to interact with GEDS directly. A Pravega Docker container including the GEDS libraries was created for straightforward deployment in Kubernetes clusters. Detailed deployment instructions are available at `https://github.com/cloudskin-eu/pravega-geds`. Table 4 summarizes the characteristics of GEDS and Pravega, along with a comparison to object storage and in-memory key-value stores.

4.4 Exposure to the Learning Plane, Telemetry and Migrations

Once deployed, applications become observable to the Learning Plane through telemetry aggregation. Metrics include:

- Workload metrics (CPU/GPU utilization, memory footprint, I/O activity),
- Infrastructure state (network latency, bandwidth, node availability),
- Application-specific KPIs (frame rate, inference latency, throughput),
- Contextual information (mobility patterns, geographic placement).

The Learning Plane uses these inputs to produce **QoS predictions** for potential execution sites across the continuum. These predictions feed resource provisioning and migration policies, either heuristic or machine-learning–driven, that determine the most suitable place to run the workload. As historical traces accumulate, predictions become more accurate, improving resource utilization and reducing SLA violations. For instance, a QoS-aware migration policy for application migration can be described as:

Table 4: Comparison of GEDS, Redis/Memcached, Object Storage, and Pravega.
Legend: KV = Key-Value; WORM = Write Once, Read Many; COS = Cloud Object Storage; NVMe = Non-Volatile Memory Express; DRAM = Dynamic Random Access Memory.

Property	**GEDS**	**Redis / Memcached**	**Object Storage (S3, Ceph, COS)**	**Pravega**
Latency	Microseconds (DRAM), low-ms for remote reads	Microseconds–1 ms	10–100 ms	1–10 ms
Throughput	20–30 GB/s reads, 10–15 GB/s writes (NVMe); DRAM-speed local I/O	Millions of ops/s per node	High capacity but lower throughput for small objects	High-throughput append-only streaming
Scalability Model	Elastic DRAM → NVMe → Object Store; metadata-driven discovery; no static pod allocation	Horizontal scaling via sharding and replicas	Virtually unlimited horizontal scalability	Elastic segments with automatic scaling
Data Model / Semantics	file + S3-like; WORM; 1:1 mapping across DRAM, NVMe, backend; supports caching mode	Ephemeral key-value store	Object-based (buckets, blobs)	Append-only streams with strong ordering and transactions
Durability	Write-back persistence to S3/COS	Optional durability (Redis AOF/RDB)	Strong durability with multi-zone replication	Durable log storage with retention; Write-back persistence to S3
Access Model	Local file paths + buffers; remote reads via network buffers; automatic S3 import; S3-cached mode with chunking and eviction	Network-based KV operations	REST/HTTP-based object operations	Client libraries for stream readers/writers
Special Features	DRAM-speed I/O; NVMe spillover; sharable with apps; no data fragmentation; allows non-GEDS consumers	Low-latency caching; optional pub/sub (Redis)	Lifecycle policies; optimal for large objects; extreme durability	Exactly-once semantics; unified streaming + storage

Listing 4: Learning Plane Policy for QoS Prediction and Migration Decisions

```
# Agent: Analyzing and Planning Strategy
# Components: Data-connector agent, NearbyOne Actuator

# --- Analyzing Strategy: QoS Predictions ---

WHEN intervalTrigger(5min, QoS_prediction(data))
IF successful_call
THEN runWorkflow("prediction-pipeline", data)

# --- Planning Strategy: Migration Decision ---

apiRequest("/analyze_qos")

IF app_cluster == "edge"
   AND current_qos > 200ms
THEN Call(NearbyOneActuator : migrate_service)

ELSEIF app_cluster == "cloud"
      AND current_qos < 45ms
THEN Call(NearbyOneActuator : migrate_service)

ELSE
THEN Nothing_to_do
```

Based on the Learning Plane recommendation, the orchestrator deploys or redeploys the application to the selected node or platform. The granularity of this step varies:

- **Coarse-grained migration:** Moving a service from the public cloud to an edge cluster to reduce latency and egress costs.
- **Fine-grained node selection:** Choosing between nodes with different accelerators (CPU vs GPU) or secure execution enclaves (e.g., Intel SGX).
- **Hybrid deployments:** Splitting components across cloud and ege, for example, decoding on the edge and inference in the cloud.

C-Cell migration. One important assess in the project is the ability to migrate C-Cells across the compute continuum. The C-Cell runtime manages scaling and migration by temporarily interrupting execution at well-defined control points. Two types of control points are defined: regular control points, which allow lightweight coordination such as message handling or shared-memory updates, and barrier control points, which ensure that the C-Cell has reached a consistent state with no in-flight messages or pending memory synchronizations. Because barrier control points provide a safe global view of the application state, allmigration operations are performed cooperatively when C-Cells reach these barriers. C-Cells supports two types of operations:

- **Vertical scaling** is achieved by launching additional C-Cells that share the same WebAssembly linear memory, differing only in their stack allocations and entry points. This design enables the runtime to expand or contract parallelism efficiently while maintaining consistency. Because vertical scaling only occurs at barrier control points, the runtime can distribute work among C-Cells according to backend semantics such as OpenMP threading or MPI ranks.
- **Horizontal migration** follows a similar principle. At a barrier, the runtime consults the Learning Plane and applies a migration plan if required. Migration is performed by snapshotting the state of a C-Cell (WebAssembly linear memory plus runtime metadata such as file descriptors) and restoring it in the destination machine. Once all runtimes have prepared for the migration, the barrier is released and execution resumes consistently across sites. This cooperative, barrier-driven mechanism ensures safe and transparent movement of C-Cells across machines with minimal disruption to application execution.

4.5 Continuous Monitoring and Re-Evaluation

Once the application is running, CloudSkin enters a continuous control loop:

1. Monitor the workload and infrastructure state,

2. Evaluate whether the current placement still satisfies QoS objectives,
3. Predict impending violations,
4. Assess placement alternatives,
5. Trigger a new deployment plan if beneficial.

This feedback loop is essential for dynamic environments such as mobility scenarios, where workloads and network conditions constantly fluctuate. Proactive migrations enabled by the Learning Plane significantly reduce SLA breaches, optimize resource usage, and improve overall system resilience.

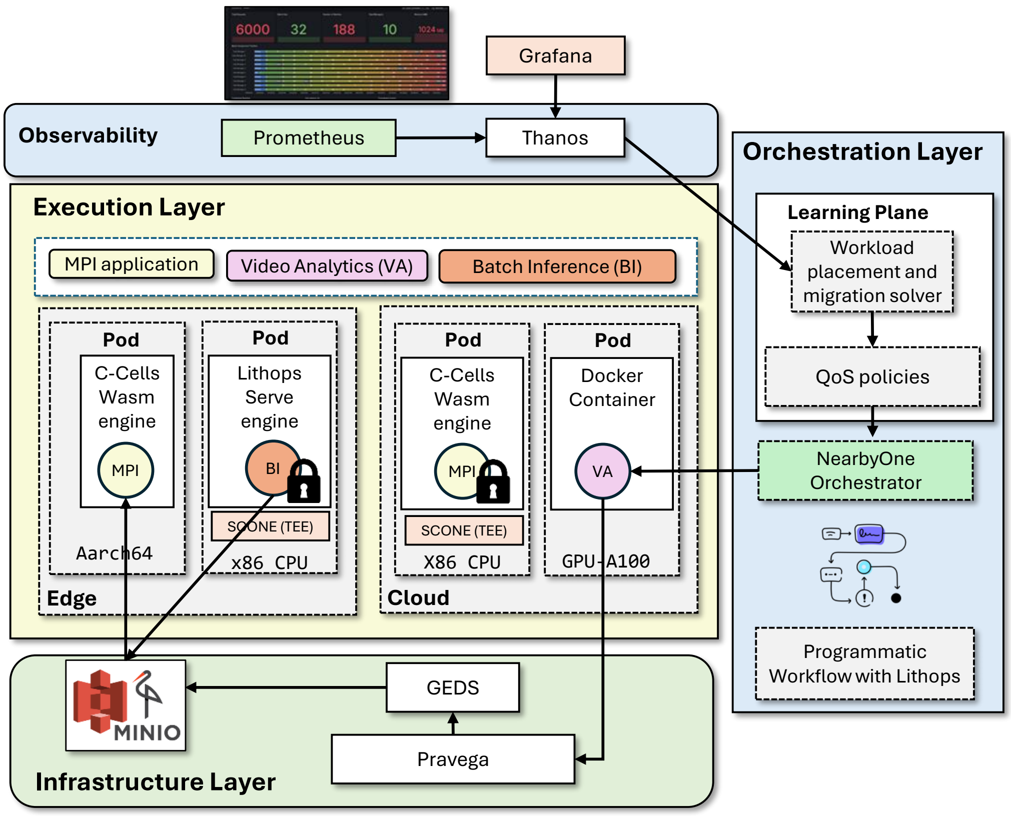

Figure 5: Interaction workflow for the different system components.

4.6 Summary

To conclude, the CloudSkin platform provides an end-to-end workflow that spans packaging, orchestration, data integration, and adaptive execution across the cloud–edge continuum. The points below summarize the essential mechanisms and capabilities presented in this section.

- **Multi-orchestrator integration:** CloudSkin seamlessly combines Kubernetes, `Lithops`, and `NearbyOne` to support containers, functions, and distributed multi-site deployments.
- **Portable packaging formats:** Workloads can be packaged as OCI containers, WebAssembly modules, or serverless functions, ensuring portability across heterogeneous environments.
- **Unified storage abstraction:** MinIO, GEDS, and Pravega provide consistent object and streaming storage interfaces for both cloud and edge sites.
- **GEDS–Pravega interoperability:** Integration via the HDFS API enables Pravega to use GEDS as an LTS backend, with configuration-driven HDFS bindings for flexible deployment.
- **Learning Plane connectivity:** Applications expose telemetry that feeds QoS prediction algorithms and AI-based decision-making for provisioning and migration actions.

- **C-Cell mobility and scaling:** C-Cells enable both vertical scaling and safe horizontal migration at barrier control points through cooperative snapshot-and-restore mechanisms.
- **Multi-site orchestration with NBI:** `NearbyOne` coordinates deployments and migrations across clusters via a RESTful Northbound Interface and lightweight per-site agents.
- **Continuous optimisation loop:** CloudSkin continuously monitors execution, predicts potential QoS violations, and triggers redeployments to maintain service guarantees across dynamic environments.

Figure 5 summarizes these interactions across the deployment of three applications: a MPI program (MPI), a video analytics (VA) pipeline, and a batch inference (BI) workflow:

- Applications running in pods interact with MinIO to store and retrieve datasets, intermediate outputs, and model artifacts. This includes `torch` tensors and images, telemetry logs, and video frames. MinIO acts as a unified object-storage interface used by C-Cells, enabling transparent data sharing across cloud and edge sites. Pravega supports streaming workflows and is integrated with GEDS for tiered, durable storage. Data flows from Pravega → GEDS → MinIO when stream tiering or archival is required.
- On the edge, Lithops Serve serverless instances run inside SCONE TEEs to process a batch inference job. The serverless instances use MinIO to read job inputs and persist inference results. The Video Analytics (VA) container running on a GPU-enabled node pushes vide frames to Pravega→GEDS.
- Prometheus collects telemetry metrics from workloads, such as latency, throughput, and resource usage. These metrics are forwarded to `Thanos` for long-term retention and remote querying. `Grafana` accesses `Prometheus-Thanos` metrics to visualize application health, performance, and QoS trends.
- The Learning Plane consumes telemetry and storage metadata to predict QoS violations, and decide whether to migrate or scale applications.
- Based on these decisions, the Learning Plane sends recommendations to the `NearbyOne` orchestrator, which then triggers horizontal migrations by moving a pod between cloud ↔ edge, new deployments, or resource adjustments.

In summary, the arrows depict a closed-loop system where:

- Data flows from apps → storage → compute.
- Telemetry flows from compute → `Prometheus/Thanos` → Learning Plane.
- Decisions flow from Learning Plane → `NearbyOne` → deployments/migrations.
- Workflows flow from developers → `Lithops` → compute nodes.

This forms an autonomous, AI-driven, multi-site orchestration cycle spanning the cloud–edge continuum.

5 Final Functional Specifications

Here, we provide the final functional requirements (FRs) for the CloudSkin platform. We summarize them in Table 6. We also provide the FRs associated with each use case. Some of them have varied in the second half of the project as early prototypes evolved to MVP (**Minimum Viable Product**) and include more features. Table 6 reflects the final and intended associations between the FRs and use cases as of M36 of the project.

To the very best of our knowledge, we see this set of FRs enough to accomplish the mission of CloudSkin, while not falling in the trap of over-engineering the solution with unnecessary features that make the system lose generality.

Further contextualization. Some FRs in Table 6 need further contextualization within the scope of the project:

- **FR1 – Flexible Orchestration:** Orchestration and management strategies vary significantly depending on the use case, making a "one size fits all" approach impractical. For example, Kubernetes is ideal for cloud-native containerized applications, `Lithops` [7] excels in serverless multi-cloud orchestration, and `NearbyOne` [6] is specifically tailored for 5G and edge computing scenarios. Selecting the appropriate orchestrator per workload ensures optimal performance and resource utilization.

- **FR5 – Heterogeneous Execution Support:** Some applications do not require migration across the cloud–edge continuum. In these cases, complex software stacks that are difficult to compile to WebAssembly, such as PyTorch can be deployed as-is to maximize performance. Supporting FR5 also requires heterogeneous architecture compatibility, including varying instruction sets and node types. Currently, WebAssembly execution modes: interpreted, Ahead-of-Time (AoT) compiled, and Just-in-Time (JIT) compiled, are not universally supported across all architectures, so careful selection per platform is necessary.

- **FR6 – Sandboxed Multi-Tenant Execution:** FR6 ensures that CloudSkin can execute sandboxed code from different tenants within the same container, providing execution semantics equivalent to threads and processes. This capability, in combination with FR4 (mobility and migration), enables both secure and transparent movement of workloads across the continuum. WebAssembly is a key enabler here: it provides sufficient generality to support continuum applications, lightweight deployment compared to containers [8], small memory footprints, and minimal performance overhead, making it ideal for multi-platform execution.

Table 6: **Final functional requirements for the CloudSkin platform**.
In this table, the four use cases are labeled as follows: 1. **Mobility**, for orchestrating applications in cloud-edge and mobile environments; 2. **Metabolomics**, for spatial metabolomics analysis; 3. **CAS**, for computer-assisted, edge-driven surgery; and 4. **Agriculture**, for managing and leveraging the agriculture dataspace.

No.	Functional requirement	Software layer(s)	Associated use case(s)	Further context and implications
FR1	Respond and adapt intelligently to changes in application behavior and data variability to optimize where data is being processed (e.g., very close to the user at the edge, or in centralized capacities in the Cloud).	L3	Mobility, Metabolomics, CAS	This will require interfacing with orchestrators to offer automatic deployment, mobility, and secure adaptability of services from Cloud to edge.
FR2	Ensure extensibility of the AI-enabled control plane with new Machine and Deep Learning models to expand the reach of the CloudSkin platform to other use cases.	L3	All	Interoperability challenges may arise between computing providers, orchestrators, and the Learning Plane. Open standards, interoperability models, and open platforms should be considered where appropriate.
FR3	Collect and manage metrics and telemetry to extract knowledge from both the underlying infrastructure and the decision-making systems.	All	All	This will require developing an interface to push telemetry data to the Learning Plane. Standard open-source monitoring and alerting systems (e.g., `Prometheus`) should be considered where appropriate.
FR4	Enable migration of execution contexts and data in order to facilitate cross-Cloud, Cloud-edge and cross-edge workflow execution to transparently integrate the diverse compute continuum resources.	All	All	Migration of execution contexts and data needs to be lightweight to make relocation transparent to users. Services may be self-migratable, require independence from the provider (FR5), and demand trusted execution (FR7).

No.	Functional requirement	Software layer(s)	Associated use case(s)	Further context and implications
FR5	Provide an adaptive virtualization layer that enables the seamless execution of the same legacy code across the whole continuum (e.g., both in an HPC cluster or at an edge server).	L2	Metabolomics	This requires the use of portable super-lightweight containers that can run anywhere from the edge to the Cloud, e.g., based on WebAssembly.
FR6	Virtualize execution memory such that code from different suppliers can safely execute side-by-side in the same physical machine.	L2	Metabolomics	This requirement calls for safety guarantees such as Software Fault Isolation (SFI) to protect computations from security breaches and other types of failures, enforcing strict boundaries between collocated processes.
FR7	Ensure confidential processing of data to make users confident that their sensitive data will stay private and encrypted even while being processed in the Cloud and edge.	L2	All	This not only requires confidential execution of native code, but also of lightweight WebAssembly containers to comply with FR5.
FR8	Develop efficient Cloud and edge storage services for efficiently managing ephemeral data.	L1	CAS	To improve I/O performance, the storage service must also support multi-tiering to achieve the targeted performance at the lowest possible cost, and make data survive temporary failures at the edge.
FR9	Integrate an elastic streaming storage fabric to enable edge use cases with stringent low-latency streaming requirements, such as real-time video analytics.	L1	CAS	While auto-scaling mechanisms for stream processing engines exist, elasticity for data streams in the storage is challenging, but it is crucial to adapt to changing data rates.
FR10	Provide a unified interface for managing heterogeneous serverless and containerized workloads, enabling deployment, execution, and monitoring of distributed functions across cloud, edge, and IoT environments.	L3	Metabolomics, Agriculture	The system should support diverse execution models and APIs, allow parallel execution, and minimize operational overhead while preserving security and portability.
FR11	Enable modular multi-site orchestration of distributed applications across cloud and edge sites, coordinating deployment, migration, and lifecycle operations while preserving site autonomy.	L3	Mobility	Orchestration should allow external triggers for initiating workflow actions, expose clear APIs, and ensure consistent application lifecycle management across heterogeneous and geographically distributed sites.

No.	Functional requirement	Software layer(s)	Associated use case(s)	Further context and implications
FR12	Treat storage chunks as first-class data units rather than raw bytes, enabling runtime transformations, buffering, routing, and semantic annotation for enhanced data management.	L1	CAS, Metabolomics	Storage services should allow in-transit processing of data chunks, including compression, encryption, metadata annotation, and intelligent routing to different storage systems or compute nodes. This enables more efficient, privacy-aware, and semantically rich data pipelines while supporting real-time and batch analytics.

6 Reference implementation

Here, we provide the full description of the reference implementation of each layer in the project.

Table 7: List of platform components.

Name	Layer	Functional Require-ments	KPIs	Role
Learning Plane	L3	FR1, FR2, FR3	KPI1, KPI2	Learning Plane data-connector agent has capabilities to connect orchestrators by writing customized actuator (FR1), to call different ML inference pipelines generated by different use cases (FR2), and to connect telemetry and save predictions data to the database (FR3).
C-Cells	L2	FR4, FR5, FR6	KPI1, KPI4, KPI5, KPI10	Distributed WebAssembly-based execution units designed to provide an adaptive virtualization layer that enables the seamless execution of the same legacy code across the whole continuum (FR5), supporting elastic and portable execution of parallel applications across cloud and edge clusters. Execution memory is virtualized to allow code from different suppliers to safely run side-by-side on the same physical machine (FR6).
GEDS	L1	FR8	KPI7, KPI8	The Generic Ephemeral Data Store (GEDS) excels at the efficient handling of temporary data created, exchanged, and consumed by compute tasks of complex, potentially multi-staged workloads. Efficiency is achieved by direct integration of application buffer management with the lowest tier (Tier 0) of the multi-tiered GEDS.
`NearbyOne`	L3	FR11	KPI10, KPI3	`NearbyOne` is a cloud-native edge computing platform designed explicitly for multi-site orchestration, aligning directly with FR11. It provides a unified, single-pane-of-glass control layer that manages distributed edge and cloud platforms while preserving the autonomy of each site. Through its inter-node orchestration capabilities, `NearbyOne` coordinates applications across multiple edge nodes, allowing them to scale, migrate, or adapt based on the Learning Plane. Distributed "Nearby Blocks" extend functionality to each edge platform, allowing new capabilities to be introduced seamlessly and orchestrated at scale. Because Nearby One relies on standard Kubernetes-based orchestration and supports cloud-to-edge workload portability, it satisfies FR11 by enabling coherent, modular, and scalable orchestration of distributed applications across geographically separated cloud and edge sites.

Continued on next page

Table 7 continued from previous page

Name	**Layer**	**Functional Require-ments**	**KPIs**	**Role**
`Lithops`	L3	FR10	KPI9, KPI12	Python-based multi-cloud serverless framework enabling transparent execution of massively parallel functions and data analytics. It abstracts cloud compute (FaaS, VMs, containers) and storage systems into a unified programming model, supporting major clouds and container platforms. In line with FR10, `Lithops` provides a flexible execution layer capable of orchestrating heterogeneous workloads, ranging from serverless functions to containerized applications, while offering a consistent API for deployment, execution, and monitoring. This enables scalable, parallel data processing pipelines across cloud and edge environments without requiring code refactoring.
SCONE	L2	FR7	KPI6	Confidential computing platform enabling secure execution of sensitive applications inside containers using TEEs (FR7). SCONE provides transparent encryption, attestation, and isolated runtime environments to protect data and code, even in untrusted cloud infrastructures. It has been used to create confidential C-Cells with a two-level sandboxing approach: an internal sandbox using WebAssembly for memory safety, and an external sandbox leveraging the enclave for strong hardware isolation. Additionally, SCONE has been employed to implement confidential workers for Lithops Serve and Pravega, enabling secure execution of serverless functions and streaming workloads across heterogeneous cloud and edge environments.
GEDS WebAssembly-based Units	L1	FR5, FR6, FR12	KPI14	W ebAssembly execution runtime integrated with GEDS to enable high-performance, sandboxed data processing near the storage layer. It interposes on standard I/O hostcalls to redirect reads and writes into GEDS in-memory, tiered storage, while providing a unified abstraction for computational storage across the continuum (FR5). The lightweight WebAssembly sandbox enables near-data execution (FR6), supporting fine-grained, low-latency compute tasks directly within the storage path. In line with FR12, these units transparently intercept local file-system reads and writes, applying on-the-fly transformations before routing the resulting data into GEDS distributed storage, thus enabling true computational storage workflows across the continuum.

Continued on next page

Table 7 continued from previous page

Name	**Layer**	**Functional Require-ments**	**KPIs**	**Role**
Pravega	L1	FR8, FR9	KPI7, KPI8, KPI11	Distributed streaming storage system that stores unbounded sequences durably and elastically, offering high-throughput, low-latency access. Integrates with GEDS for tiered long-term storage while supporting streaming analytics, real-time event processing, and video pipelines across cloud and edge sites. Its elastic storage fabric adapts dynamically to varying data rates, automatically scaling and partitioning streams to maintain consistent performance under changing workloads, thereby enabling latency-sensitive edge applications such as live video analytics and real-time AI inference.
Nexus	L1	FR12	KPI14	Nexus is a programmable, policy-driven framework that transforms tiered streaming storage into an intelligent data management layer, treating storage chunks as first-class units rather than raw bytes. It provides in-transit processing capabilities including dynamic buffering to handle network variability, semantic annotation for downstream analytics, on-the-fly data transformations such as compression, encryption, and format conversion, and intelligent routing based on performance, privacy, or cost policies. These operations occur without impacting real-time ingestion, enabling reliable, efficient, and semantically rich pipelines for latency-sensitive, data-intensive applications such as video analytics, AI model training, and edge-cloud workflows.

Table 8: Mapping of CloudSkin components to functional requirements (FRs).

System Component	**FR1**	**FR2**	**FR3**	**FR4**	**FR5**	**FR6**	**FR7**	**FR8**	**FR9**	**FR10**	**FR11**	**FR12**
Learning Plane	✓	✓	✓									
C-Cells				✓	✓	✓						
GEDS								✓				
NearbyOne											✓	
Lithops										✓		
SCONE							✓					
GEDS WebAssembly-based Units					✓	✓						✓
Pravega								✓	✓			
Nexus												✓

As shown in Table 8, each functional requirement (FR1–FR12) of the CloudSkin platform is addressed by at least one component. This mapping demonstrates that the platform comprehensively covers all required capabilities across the continuum, from AI-enabled orchestration to confidential execution, storage, streaming, and computational data processing. By linking each FR to one or more components, the table shows how the integrated software stack collectively fulfills the functional objectives of the platform.

Table 10 summarizes the key deliverables of the CloudSkin project, detailing each software layer and the responsible work package. Each deliverable represents a concrete outcome of the project, ranging from the integration of the AI-enabled Learning Plane (D5.3) to reference implementations of the cloud-edge platform (D4.3) and the underlying cloud-native infrastructure (D3.4).

Table 10: List of deliverables with full details of reference components.

Deliverable no.	**Deliverable name**	**Software layer**	**Work Package**
D5.3	Integration of the Learning Plane	L3	WP5
D4.3	Reference implementation of Cloud-edge platform	L2	WP4
D3.4	Reference implementation of Cloud native Infrastructure	L1	WP3

Table 12: Reference Architecture Components with URLs

Component	**URL**
Learning Plane	`https://github.com/cloudskin-eu/scanflow-data-connector`
C-Cells	`https://github.com/cloudskin-eu/granny`
SCONE	`https://github.com/scontain/scone`
GEDS	`https://github.com/cloudskin-eu/GEDS`
GEDS WebAssembly-based Units	`https://github.com/cloudskin-eu/GEDS_Wasm_Units`
Pravega	`https://github.com/pravega/pravega`
Nexus	`https://github.com/cloudskin-eu/nexus-nct-streamlets`
`NearbyOne`	`https://github.com/cloudskin-eu/Mobility-use-case`
`Lithops`	`https://github.com/lithops-cloud/lithops.`

7 Reference Architecture Components

This section provides a more detailed description of the key components of the CloudSkin platform. For each system component, we outline its overview, internal architecture, key features, and interfaces for integration with other layers of the platform.

7.1 Learning Plane

7.1.1 Overview

The Learning Plane (LP) orchestrates workloads across the cloud-edge continuum using AI-driven decision-making. It collects telemetry via sensors, applies ML inference pipelines for real-time reasoning, and leverages actuators to implement adaptive actions. Serving as the central intelligence in the platform, the LP ensures efficient, autonomous, and optimized resource management.

7.1.2 Architecture

The implementation architecture of the data-connector is shown in Figure 6. While the Learning Plane can be viewed as an abstract entity encompassing all the CloudSkin predictive and decision-making capabilities, it also includes a concrete reference implementation in the form of the data-connector. Acting as an LP agent, the data-connector provides QoS predictions for application workloads and generates placement, migration, as well as scaling recommendations for the orchestrator.

Its implementation builds on `Scanflow-k8s` [9][10], offering pre-configured deployments of prediction ML models, metadata management, parameter tracking, a model registry, and an agent framework for autonomic management. Developers only need to supply scenario-specific sensors and actuators. Listing 5 shows the agent is watching the QoS predictions and having a policy of performance cost trade-off (i.e., performance/-cost). If QoS constraints are satisfied, chooses the node with the best placement recommendation.

Listing 5: Custom sensor to get app QoS predictions and enable recommendation policy

```python
#example 1: watch app QoS predictions
@sensor(nodes=["predictor"])
async def watch_qos(runs: List[mlflow.entities.Run], args, kwargs):
    qos = 0
    input_data = get_qos()
    if input_data:
        qos, node_index = choose_better_nodes(input_data)
        if qos_constraints(qos):
            await call_migrate_app(max_qos_index, "icresnet", "torch-deployment")
        else:
            logging.info("all machine can not achive qos sla, no actions")
    else:
        logging.info("no data in last check")
    return max_qos
```

A detailed explanation of the LP and its integration with the use cases is provided in D5.3.

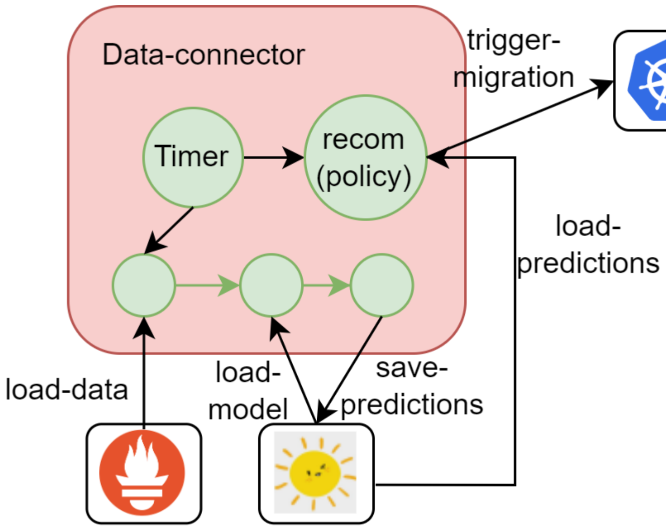

Figure 6: Implementation architecture of the data connector for the Learning Plane.

7.1.3 Key Features

- Learning Plane agents.
- Integration with ML inference pipelines.
- Telemetry collection and predictions storage.
- Support for multi-use case deployments (Mobility, Metabolomics, CAS).

7.1.4 Interfaces and Integration

The LP data-connector framework provides flexible sensors and actuators to interface with various CloudSkin data plane components (e.g., Prometheus, MLflow, Nearby observability stack) and control plane components (e.g., Kubernetes, NearbyOne, and Lithops). Depending on the scenario, the LP integrates different ML models to generate predictions, including existing models such as Long Short-Term Memory (LSTM), as described in D5.2, as well as new models introduced in D5.3.

In terms of use case integration, we highlight:

- In the mobility use case, the LP connects with the Nearby observability stack and NearbyOne, leveraging regression and time-series models to enable intelligent, QoS-aware service migration.
- In the Metabolomics use case, the LP does not drive real-time decisions. Instead, it plays a critical role behind the scenes, training two regression models: one to predict the aggregated initialization time for

w serverless functions, and the other to estimate the total execution time required to process a job of r metabolomics images using n serverless workers. Leveraging these predictions, the use case implements a lightweight scheduler that selects the optimal number of workers to minimize overall job completion time while staying within a predefined cost budget. This approach enables efficient, cost-aware scaling of large-scale inference workloads, turning predictive insights into actionable resource optimization.

- In the CAS use case, the LP consumes latency metrics published by Pravega via Prometheus, along with other relevant sys tem metrics, to drive auto-scaling decisions (e.g., adjusting the number of segment stores). Actuation is straightforward, as Pravega Segment Store instances can be horizontally scaled through Kubernetes APIs, such as via Custom Resource Definitions (CRDs).

7.2 C-Cells

7.2.1 Overview

C-Cells are a WebAssembly-based execution abstraction for CloudSkin's cloud-edge continuum. The choice of WebAssembly as intermediate representation makes C-Cells, by default, language- and hardware-independent. C-Cells can execute code written in any programming language that supports cross-compilation to Wasm, and can be executed in any architecture that can be targeted as an LLVM back-end.

C-Cells support communication via message passing and synchronization via shared memory. This makes it possible for multiple C-Cells to co-operatively run distributed multi-process and multi-threaded programs. C-Cells currently support the execution of applications written using the MPI and OpenMP programming models, but this support could be extended in the future.

7.2.2 Architecture

C-Cells are executed as different OS threads in the same Linux process. C-Cell's memory space is isolated using Wasm's sandbox, thus benefiting from Wasm's spatial memory safety guarantees. C-Cells are further isolated with a combination of control groups and network namespaces (see Figure 7). Wasm-sandboxed does not have access to any resources of the host system, this prevents code executing in a Wasm module to make any system calls to access the filesystem or the network. CloudSkin's C-Cell runtime interposes on any such call, including MPI and OpenMP calls, using control-points.

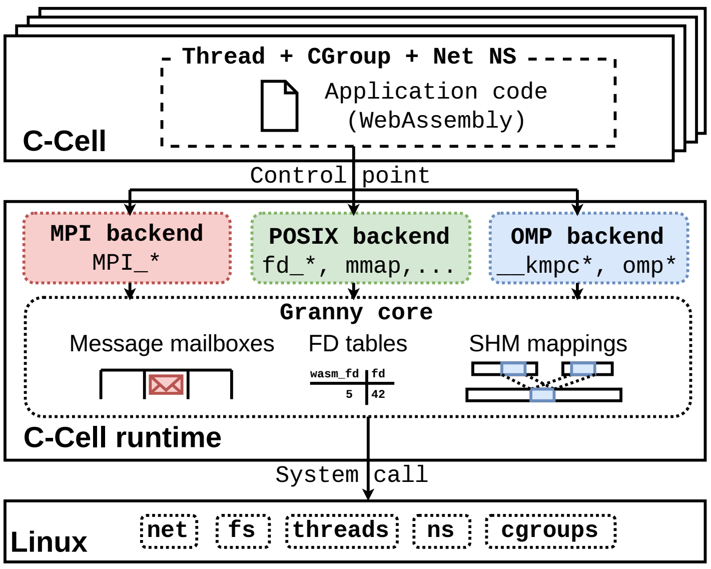

Figure 7: C-Cell overview.

A control-point is a lightweight software trap that allows the runtime to take over C-Cell execution with minimal overhead. The context switch introduced by a control-point is that of a Wasm context switch, which is comparable to a regular function call. Using control-points the host C-Cell runtime can provide its own implementation of the system, MPI, or OpenMP call. We use this strategy to change the parallelism or the distribution of OpenMP or MPI applications, respectively.

Using control-points a C-Cell runtime can interpose on OpenMP calls to fork into a parallel execution, and decide, at runtime, how many threads to use in this particular computation. We call this process vertical scaling and it can be used to elastically scale a computation up or down depending on the system's load. Similarly, we can interpose on MPI calls and migrate MPI processes without losing in-flight messages, thus reducing fragmentation and improving locality. We call this process horizontal migration.

7.2.3 Key Features

- **Adaptive Virtualization.** C-Cells support executing code written in a variety of programming languages and targeting a variety of architectures. Most of these features come from WebAssembly itself, and we extend them to, for example, support executing C-Cells inside Intel SGX enclaves.
- **Secure Isolation.** C-Cells offer lightweight virtualization, yet solid isolation guarantees. WebAssembly provides memory safety, and resource isolation is achieved with a combination of control-groups and network namespaces. This combination of isolation mechanisms allow C-Cells to boot fast, in less than 10 ms.
- **Lightweight Interruption.** C-Cells provide a lightweight interrupt mechanism in the form of control-points. Control-points are triggered when application code makes a call to any supported API that requires exiting the WebAssembly sandbox. These APIs include the system call API, the MPI API, and the OpenMP API, but can also be extended with arbitrary APIs.
- **Vertical Scaling.** Our C-Cell runtime uses control-points to interrupt OpenMP calls that fork to multi-threaded execution to elastically decide how many threads to use, based on input from the control-plane.
- **Horizontal Migration.** Our C-Cell runtime uses control-points to interrupt MPI calls that send messages to each other in order to, when no more messages are in-flight, migrate MPI processes across VMs.

7.2.4 Interfaces and Integration

- **Smart Orchestration.** Our C-Cell runtime relies on input from the control-plane to decide when to vertically scale and when to horizontally migrate. By leveraging these two new mechanisms, we can also inform new scheduling and orchestration policies for the learning-plane.
- **Ephemeral Storage.** Whenever C-Cells need to access ephemeral storage, they can do so using GEDS and, in particular, GEDS-Wasm extension, which can be directly linked to application code.
- **Confidential Execution.** For C-Cells that need to process private data, we support transparently running C-Cells inside SGX enclaves using a lift-and-shift approach as provided by SCONE.

7.3 GEDS

The **Generic Ephemeral Data Store (GEDS)** is a core CloudSkin component designed to efficiently manage temporary data produced, exchanged, and consumed by complex, multi-stage workflows deployed across the cloud–edge continuum. GEDS places particular emphasis on supporting serverless execution environments, where the number of compute elements may vary significantly over time and data locality plays a critical role in overall performance.

7.3.1 Overview

GEDS implements a high-performance, multi-tier ephemeral storage abstraction that supports fast data access, transparent data movement, and near-data computation using WebAssembly, as described in §7.7. By tightly integrating storage management with application execution, GEDS enables workloads to operate efficiently under highly dynamic resource conditions, making it particularly well suited for serverless, streaming, and data-intensive applications.

7.3.2 Architecture

GEDS follows a **multi-tier architecture** centered around a tightly integrated **Tier 0** buffer management layer. Tier 0 interfaces directly with local resources through a filesystem abstraction, allowing seamless integration of DRAM and fast local storage such as NVMe devices. All local GEDS objects are represented as files, enabling memory-mapped access for low-latency data operations and implicit acceleration through the OS page cache.

Beyond Tier 0, GEDS supports a **Persistency Tier** that provides node-independent, disaggregated storage and object persistence. This tier is integrated through an **S3-compatible API** and can be backed up by object storage systems such as IBM COS or AWS S3. GEDS supports both application-initiated and automatic object spilling from Tier 0 to the Persistency Tier, triggered when configurable local resource limits are reached. At present, object relocation is governed by an LRU policy.

With the exception of a lightweight **Metadata Server**, GEDS is implemented as an application-loadable library rather than a standalone storage service. This design minimizes deployment overhead and permits GEDS to be natively integrated into existing applications, with bindings currently available for C++, Python, and Java. Fig. 8 exemplifies a deployment of GEDS in a Kubernetes environment running a Python workload with the corresponding code in Listing 6.

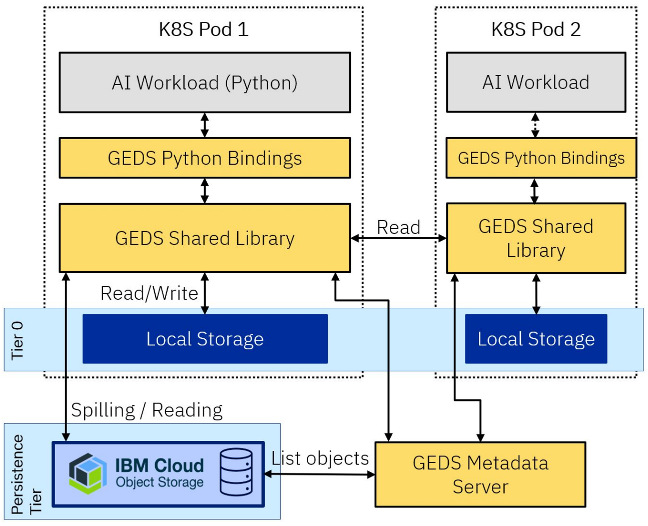

Figure 8: GEDS example deployment in Kubernetes.

```
import os
import sys
import time
import io
import numpy as np
from threading import Thread

from smart_open import open
import geds_smart_open
from geds_smart_open import GEDS

bucket = 'geds-test'

AWS_SECRET_ACCESS_KEY = os.getenv('AWS_SECRET_ACCESS_KEY')
AWS_ACCESS_KEY_ID = os.getenv('AWS_ACCESS_KEY_ID')
AWS_ENDPOINT_URL = os.getenv('AWS_ENDPOINT_URL')

geds_smart_open.register_object_store(
    bucket,
    AWS_ENDPOINT_URL,
    AWS_ACCESS_KEY_ID,
    AWS_SECRET_ACCESS_KEY
)

output_csv = open('file://ai_training.csv', 'w')
output_csv.write("Threads,Time,Data Read,Data Written,Data Spilled\n")

def read_file(tid, buffers):
    path = f'geds://{bucket}/ml_model_{tid}.bin'
    with open(path, 'rb') as f:
        buffers[tid] = f.read()

```

```
def write_checkpoints(tid, buffers):
    path = f'geds://{bucket}/checkpoint/checkpoint_{tid}.bin'
    with open(path, 'wb') as f:
        f.write(buffers[tid])

def persist():
    geds_smart_open.relocate()

def benchmark_threads(num_threads, csv):
    buffers = [None] * num_threads

    # Read checkpoints
    start_time = time.time_ns()
    threads = [
        Thread(target=read_file, args=(i, buffers))
        for i in range(num_threads)
    ]
    [t.start() for t in threads]
    [t.join() for t in threads]

    duration = (time.time_ns() - start_time) / (1000**3)
    length = sum(len(b) for b in buffers)
    csv.write(f'{num_threads},{duration},{length},0,0\n')

    # Write checkpoints
    start_time = time.time_ns()
    threads = [
        Thread(target=write_checkpoints, args=(i, buffers))
        for i in range(num_threads)
    ]
    [t.start() for t in threads]
    [t.join() for t in threads]

    duration = (time.time_ns() - start_time) / (1000**3)
    csv.write(f'{num_threads},{duration},0,{length},0\n')

    buffers = None

    # Persist data
    start_time = time.time_ns()
    persist()
    duration = (time.time_ns() - start_time) / (1000**3)
    csv.write(f'{num_threads},{duration},0,0,{length}\n')

for t in [1, 2, 4, 6]:
    benchmark_threads(t, output_csv)
    output_csv.flush()

output_csv.close()
```

Listing 6: Multi-threaded GEDS I/O benchmark used for AI training workload characterization.

7.3.3 Key Features

- **High-performance ephemeral data management:** GEDS has been specifically optimized for short-lived, intermediate data generated by multi-stage and data-intensive workloads. By prioritizing low-latency and high-throughput access over strict durability guarantees, GEDS reduces I/O overhead compared to traditional persistent object stores, enabling efficient exchange of intermediate results across cloud–edge resources.

- **Multi-tiered storage architecture for performance and resilience:** GEDS transparently manages ephemeral data across a hierarchical storage stack comprising DRAM (Tier 0), local or disaggregated block storage such as NVMe (Tier 1), and S3-compatible object storage (Tier 2). This tiered design allows hot data to remain in high-performance memory while automatically spilling colder data to lower tiers based on the LRU policy, ensuring elastic operation under memory pressure without requiring application-level changes.

- **Near-data computation via GEDS-based WebAssembly Units:** To support data transformation pipelines, GEDS integrates a novel WebAssembly runtime that extends standard WASI with native, GEDS-aware I/O primitives. These **GEDS-based WebAssembly Units** enable lightweight, isolated execution of data preprocessing and transformation tasks directly close to the storage tiers, minimizing data movement and improving end-to-end pipeline efficiency across the cloud–edge continuum.

7.3.4 Interfaces and Integration

- **Unified storage substrate for CloudSkin runtime components:** GEDS acts as the ephemeral backbone for multiple CloudSkin platform components, including **C-Cells**, **Nexus**, and **Pravega**. By providing a shared namespace, GEDS enables efficient data exchange between distributed batch stages, serverless and edge-based workloads.

- **Seamless interoperability with distributed and near-data runtimes:** GEDS inter-operates with both the C-Cells distributed runtime and the GEDS-based WebAssembly runtime. While C-Cells focuses on compute-intensive, MPI/OpenMP-style parallelism, GEDS-based WebAssembly Units target fine-grained, I/O-bound transformations executed close to data. Both Wasm runtimes can share the same Wasm code, allowing portable binaries to be executed either as part of distributed workflows or as standalone near-storage tasks, enabling flexible and transparent cloud–edge integration.

7.4 NearbyOne

7.4.1 Overview

`NearbyOne` is a powerful, cloud-native orchestration and automation platform designed to streamline and simplify the management of complex cloud, edge, and private network infrastructures. By providing a unified operational layer, NearbyOne enables businesses and service providers to automate deployments, manage infrastructure, and orchestrate services efficiently across multi-cloud, edge computing, and telecom environments.

The NearbyOne orchestrator is responsible for the service onboarding and life-cycle management of cloud-native applications and infrastructure at a global scale, and across the continuum. NearbyOne enables dynamic automation, ensuring that services are instantiated, monitored, and optimized across diverse infrastructure components.

7.4.2 Architecture

NearbyOne architecture, depicted in Figure 9, presents a unified control and management plane, i.e., the core orchestration platform while allows to span multiple sites (public cloud, private cloud, and edge resources). In CloudSkin, each site is based on Kubernetes, which serves as the common abstraction layer for managing containerized workloads.

This multi-cluster setup allows CloudSkin to seamlessly deploy and operate services (e.g., the mobility use case) across cloud and edge, while abstracting infrastructure heterogeneity from application developers. The NearbyOne controller itself is a cloud-native platform that can be deployed on Kubernetes. As a result, it operates using containerized services, each with specific resource requirements in terms of CPU, memory, and storage.

The NearbyOne solution used in CloudSkin is composed of four main building blocks (see Figure 9):

- **Management dashboard and Orchestration Engine**: This is the central component responsible for end-to-end orchestration of applications and infrastructure. It automates provisioning, deployment, scaling, migration, and configuration management using declarative specifications (YAML), without requiring any modification to application code.

- **Nearby Blocks**: Nearby Blocks are reusable, higher-level components that encapsulate application logic together with orchestration metadata and management capabilities. Each block references Helm charts and container images, enabling standardized deployment and lifecycle management of both core services (e.g., observability) and mobility use case applications.

- **Northbound Interface (NBI)** Orchestration API: A RESTful API that exposes orchestration capabilities to external entities, notably the Learning Plane, enabling AI-driven automation such as service migration and optimization decisions.

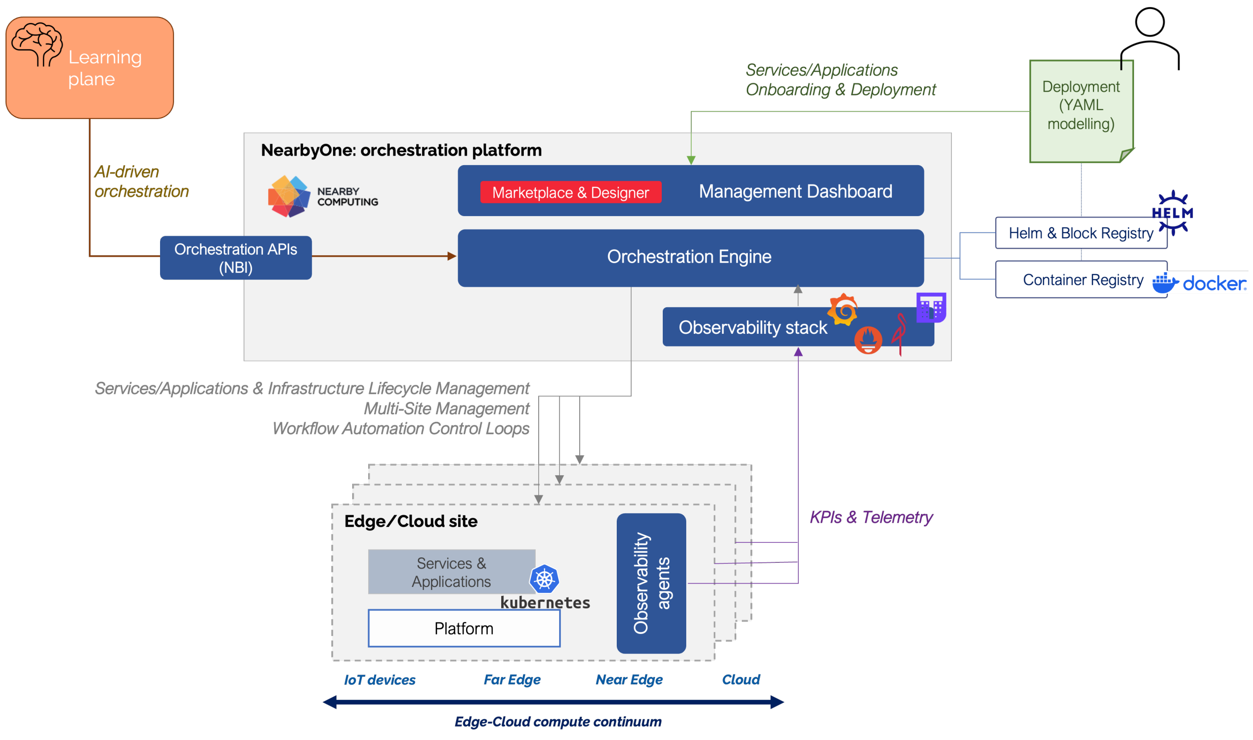

Figure 9: `NearbyOne` architecture overview.

- **Observability Stack**: Built on cloud-native open-source technologies, the observability stack enables efficient collection, aggregation, storage, and visualization of telemetry data from all managed clusters.

As shown in Figure 9, the NearbyOne orchestration platform provides a single pane of glass through the NearbyOne Management Dashboard. This GUI enables users to: define service deployment chains, specify placement, configuration, and policy constraints, monitor system state, and deploy applications with minimal effort. At the core of the platform lies the Orchestration Engine, which follows self-healing reconciliation loops and enforces eventual consistency rather than transactional execution. This design allows the platform to manage complex, distributed operations and dynamically adapt to infrastructure or service changes. A built-in Marketplace further simplifies operations by offering one-click deployment of pre-packaged solutions. For CloudSkin, the marketplace includes: observability stack blocks, the mobility use case DL Streamer video analytics application, a dynamic DNS service and example applications such as NGINX and TorchServe.

Nearby blocks are the high-level, versioned entities that package and compose orchestration resources for deployment, reuse, and lifecycle management of services. NearbyOne is not limited to Kubernetes-native constructs or node management. Instead, it provides a unified, declarative resource model that extends well beyond Kubernetes, enabling consistent management of clusters, edge nodes, workloads, and infrastructure at scale. Listing 7 shows a component of the *DL Streamer Pipeline Server* Nearby Block from the mobility use case. A Nearby Block specifies the entire desired state of the DL Streamer component, from image version and placement configuration to ingress and DNS configuration, without modifying any of the application's internal code.

- `kind:ChartDeployment`, NearbyOne's custom object representing a reusable DL Streamer workload deployment.
- `connectionListSelectors`, a workflow coordination primitive, in this case used to automatically discover the Prometheus server where the DL Streamer will report metrics.
- `k8sClusterSelector`, used to determine the placement of the Helm chart to be deployed. This selection can be driven by labels or by Key Performance Indicators (KPIs).
- `values`, overrides DL Streamer Helm defaults (such as enabling Ingress, or setting hostnames).

Listing 7: A representative Nearby Block snippet of the DL Stramer Pipeline Server (mobility use case)

```yaml
# Copyright 2025 Nearby Computing S.L.
apiVersion: blocks/v1beta1
kind: ChartDeployment
metadata:
  name: dlstreamer-deployment
  namespace: {{ .Values.Block.InstanceId }}
spec:
  connectionListSelectors:
    promRelease:
      matchLabels:
        application: releases-monitoring-prometheus
        site.nbycomp.com/{{ .Values.placement.site.label }}: "true"
      required: 1
      limit: 1
  template:
    metadata:
      labels:
        application: dlstreamer-{{ .Values.Block.InstanceId }}
        app: dlstreamer-cloudskin
        id: dns-{{ .Values.Block.InstanceId }}
    spec:
      k8sClusterSelector:
        matchLabels:
          site.nbycomp.com/{{ .Values.placement.site.label }}: "true"
      chart: dlstreamer-pipeline-server
      version: "0.3.0"
      repo:
        url: dlstreamer.cloudskin.repo
        username: cloudskin-user
        password:
          secretKeyRef:
            name: cloudskin.name
            key: cloudskin.key
      values: |
        imagePullSecrets:
          - name: cloudskin-image-pull-secret-name
        pipelineServerName:  &pipeline-server-name "dlstreamer-pipeline-server"
        pipelineServerStatusURL: &pipeline-server-status-url "http://dlstreamer-
            pipeline-server:8080/pipelines/status"
        pipelineServerMqttName: &pipeline-server-mqtt-name "dlstreamer-pipeline-
            server-mosquitto"
        fullnameOverride: *pipeline-server-name
        prometheus-json-exporter:
          enabled: true
          ...
        mqtt-exporter:
          enabled: true
          ...
        ingress:
          enabled: true
          ...
```

The NearbyOne NBI Orchestration API, enables programmatic interaction with the orchestrator. It allows the Learning Plane to trigger automated actions—such as service migration between cloud and edge—based on AI-driven insights, closing the loop between monitoring, learning, and orchestration.

The NearbyOne observability stack is deployed on top of NearbyOne as a service. Therefore, this is subject to be adapted to the specific needs of each deployment. It is composed of standard observability components commonly used across multiple domains to collect and aggregate telemetry. The observability stack is used to collect, transport, and aggregate telemetry. The consolidated telemetry is available internally to the orchestration engine to influence policies and configurations, or externally to users for visualization purposes or the Learning Plane for the AI-driven orchestration.

- *Prometheus* is the main telemetry collection component, extensively used to collect telemetry from multiple services at a cluster level.
- *Thanos* is designed to aggregate and consolidate short-lived data stored in Prometheus cluster-stores, as well as to expose a long-term Prometheus-compatible interface to the data.
- *MinIO* is a high performance, distributed object storage system that acts as the telemetry repository for Thanos.
- *Grafana* for visualization and analytics.

All observability components are deployed and configured through NearbyOne Blocks, making monitoring setup fully automated and consistent across clusters.

Moreover, NearbyOne relies on several external registries to support flexible and secure deployment workflows:

- *Container Registries*, for storing Docker images (public or private),
- *Helm Chart Registries*, for packaging and deploying complex Kubernetes applications, and
- *Nearby Block Registries*, for storing Nearby Blocks, which reference Helm charts and container images.

7.4.3 Key Features

- Unified multi-site orchestration, manages and automates application and infrastructure lifecycle across public cloud, private cloud, and edge sites through a single control plane.
- Single-pane-of-glass management interface, a graphical interface and one-click deployment of services.
- Resource allocation and service lifecycle management, based on self-healing reconciliation loops and policy-driven behavior to manage complex, distributed operations with eventual consistency.
- Multi-cluster observability stack, based on open-source components, enabling scalable monitoring across the cloud-edge continuum.
- AI-driven automation via NBI: Exposes a northbound REST API that enables the Learning Plane to trigger automated orchestration actions such as scaling and service migration.

7.4.4 Interfaces and Integration

NearbyOne integrates seamlessly with the CloudSkin reference architecture by providing the orchestration, automation, and control backbone required to manage distributed cloud–edge infrastructures and services, directly supporting FR11. Through its Northbound Interface (NBI) Orchestration API, NearbyOne exposes programmatic control of orchestration workflows to external CloudSkin components, most notably the Learning Plane. This interface enables AI-driven decision-making loops, where insights derived from monitoring data and analytics can trigger automated actions such as service (re)deployment, scaling, or migration between cloud and edge sites. The NBI abstracts the internal complexity of the orchestration platform, allowing the Learning Plane to interact with NearbyOne through well-defined, technology-agnostic RESTful endpoints.

At the infrastructure level, NearbyOne interfaces with Kubernetes clusters via its southbound integration, while at the service level it relies on declarative specifications (YAML, Helm charts, and Nearby Blocks) to ensure consistent and reproducible deployments. This approach allows application developers and platform services to integrate without modifying application code, fostering interoperability and rapid onboarding of third-party components within the CloudSkin ecosystem.

7.5 Lithops

7.5.1 Overview

`Lithops` is a serverless, multi-cloud framework that enables the orchestration and execution of data analytics workflows across heterogeneous cloud and edge environments. In addition to Function-as-a-Service (FaaS) platforms, `Lithops` can also launch functions on containers and virtual machines, making it suitable for edge deployments and hybrid infrastructures. It addresses intrinsic challenges of large-scale computation, including the lack of direct communication between functions, absence of native synchronization between computational stages, and general portability limitations due to proprietary APIs. `Lithops` lets developers port single-process Python code to a fully distributed execution model, while abstracting the complexities of IaaS management.

By orchestrating hundreds of concurrent function instances, `Lithops` delivers substantial performance gains for workloads that require short-lived, highly parallel computations. This capability is critical for FR10, as `Lithops` provides a unified API to manage heterogeneous serverless functions, containerized applications, and distributed tasks across cloud-edge environments. Simply put, developers can deploy, run, and monitor[10] parallel workflows through a consistent programming model.

`Lithops` has served as the foundation for the development of **Lithops Serve**, a distributed batch inference engine that operates seamlessly across both cloud and edge environments. Building on `Lithops` serverless and multi-cloud capabilities, Lithops Serve orchestrates multiple parallel function instances to execute AI inference tasks at scale, enabling efficient, low-latency processing in heterogeneous deployment scenarios.

7.5.2 Architecture

Lithops follows a layered design that separates the concerns of local job management and remote serverless execution. Its architecture can be divided into two main categories:

- **Local components:** These run on the client machine. The primary interfaces are the `Futures API` for job submission and the `Executor`, which partitions jobs, uploads input data and dependencies to storage back-ends, orchestrates the scheduling of remote workers, and collects results.
- **Remote components:** These execute user code in diverse environments, including FaaS platforms, on-premises Kubernetes clusters, OpenWhisk, and virtual machines. The `Invoker` manages the deployment of remote workers, which fetch user code and dependencies from object storage, execute the tasks, and store results back. Each worker is `Lithops`-aware, enabling seamless, scalable, and transparent execution of user-defined code across heterogeneous infrastructures.

The architecture abstracts away cloud-provider specifics, allowing `Lithops` to operate across AWS Lambda, Google Cloud Run, and other FaaS platforms. It also provides storage-agnostic access through its Storage API, which supports object storage and memory-based systems such as Redis. This multi-cloud and multi-backend support ensures portability and mitigates vendor lock-in while preserving high-performance execution. Fig. 10 depicts the `Lithops` architecture.

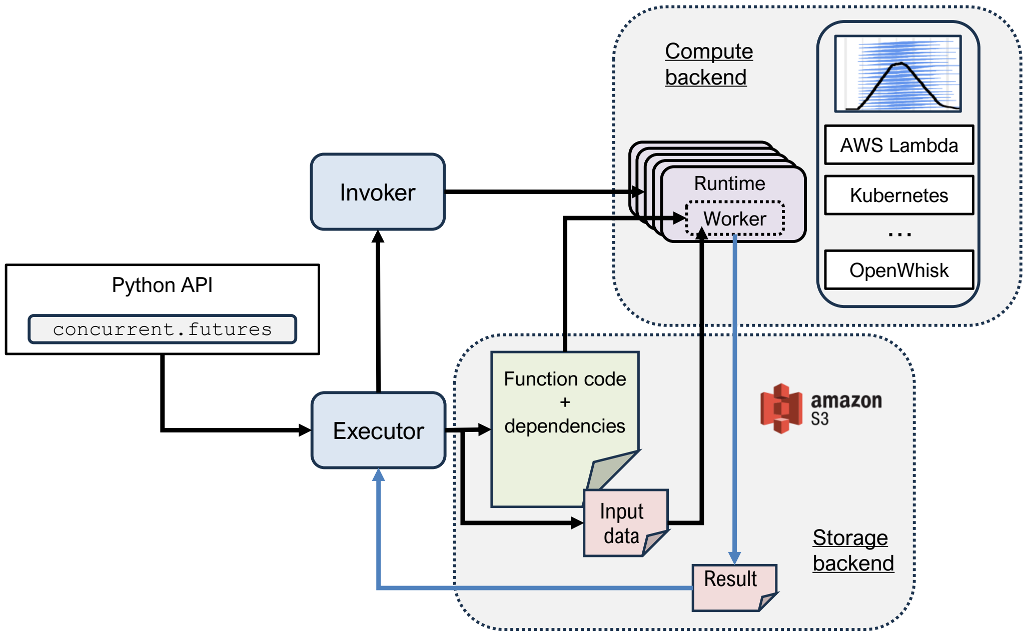

Figure 10: `Lithops` architecture overview.

7.5.3 Programming Model

Lithops exposes high-level APIs that simplify parallel programming on cloud functions:

[10]https://lithops-cloud.github.io/docs/source/monitoring.html

- **Futures API:** Enables developers to submit jobs as single-machine Python functions. The `map()` function executes a given function over multiple data partitions concurrently, while `map_reduce()` extends this capability by collecting intermediate results and applying a reduction step. This interface allows users to write code that resembles sequential execution while `Lithops` handles distributed orchestration.

- **Storage API:** Abstracts access to cloud storage and in-memory storage, enabling functions to read and write data without concern for backend differences. This ensures that workflows can seamlessly interact with heterogeneous storage backends, including S3, MinIO, GEDS, or Redis, while remaining portable across cloud providers.

`Lithops` automatically handles task partitioning, function invocation, dependency distribution, and result aggregation. Synchronization between workflow stages, including nested function compositions, is implicit, reducing the programming burden and avoiding errors associated with distributed systems. In what follows, you can find a snippet of code for counting words in parallel:

Listing 8: Distributed word count using Lithops map-reduce

```python
"""
Simple Lithops example using map() to count word occurrences in text chunks
"""
import lithops
from collections import Counter

# Function to count words in a single chunk
def count_words(text_chunk):
    words = text_chunk.split()
    return Counter(words)

if __name__ == '__main__':
    # Example: split text into chunks
    texts = [
        "lithops is a serverless framework for parallel computing",
        "it can run code on cloud functions, containers, or vms",
        "map and reduce primitives enable large scale data analytics",
        "users can process many text chunks concurrently"
    ]

    # Create a Lithops executor
    fexec = lithops.FunctionExecutor()

    # Map phase: run word count on each chunk
    fexec.map(count_words, texts)

    # Collect results
    results = fexec.get_result()

    # Reduce phase: combine word counts
    total_count = Counter()
    for r in results:
        total_count.update(r)

    print("Aggregated word counts:", dict(total_count))
```

7.5.4 Key Features

- Transparent orchestration of hundreds to thousands of concurrent serverless functions.
- Unified interface for serverless, containerized, and edge workloads.
- Abstracted storage API ensuring multi-cloud portability and seamless interaction with object or memory-based storage systems.
- MapReduce-style and Futures API for simplified parallel programming.

- Automatic dependency management and implicit synchronization between computation stages.
- High elasticity: functions can scale from zero to thousands of concurrent executions in seconds.
- Integrated fault tolerance and retry mechanisms to ensure reliable execution.

7.5.5 Integration with the Reference Architecture

`Lithops` integrates seamlessly with other CloudSkin components to support FR10 by providing a high-level execution layer for distributed batch inference workloads, particularly in the **Metabolomics** use case in the project. It leverages object storage as an intermediate exchange data layer, allowing it to interface with virtually any component in the platform. Consequently, `Lithops` can consume data from GEDS and Pravega, tiered to S3 or other S3-compatible object stores such as MinIO. Further, it can interact directly with GEDS and Pravega via their native Python APIs.

This architecture enables fully distributed and parallelizable data pipelines across cloud and edge settings. Its API abstractions also allow `Lithops` to coordinate execution with orchestrators or Learning Plane agents, responding dynamically to telemetry data or policy-driven triggers.

7.6 SCONE

7.6.1 Overview

SCONE is a confidential computing platform that provides a lift-and-shift approach for native applications to leverage TEEs (Trusted Execution Environments) without many modifications to enable secure execution of sensitive applications. In a nutshell, SCONE has been applied in several key contexts throughout the project:

- In the context of C-Cells, SCONE is able to provide a further level of **sandboxing** by running it inside an Intel SGX enclave, complementing the existing WebAssembly sandbox for memory safety. By running a C-Cell inside an enclave, not only is **hardware-level isolation** guaranteed, but also **memory integrity** from powerful potential outsider attacks.
- SCONE also brings confidential computing to `Lithops` workers and Pravega servers. As C-Cells, both are core components of the reference implementation.
- SCONE is also employed to address computational confidentiality and integrity in CloudSkin use cases, namely Metabolomics and CAS.

7.6.2 Architecture

In general, SCONE leverages the fact that an application requires a C library to function. SCONE implemented a customized C library, replacing the one the application is compiled against. This substitution is transparent, meaning that, in most cases, no changes are needed in the application. If the application is executed on top of a specific sandbox such as Python or Faasm, then both the application and the sandbox are protected. Looking at current research, many extensions to SCONE are available, such as protecting application integrity [11] or preventing rollback attacks on storage [12].

When the application loads into memory, SCONE can distinguish this process and place the necessary parts of the application into the TEE. Although system calls are still executed outside the TEE, SCONE provides a shield to filter and secure system call execution. Additionally, SCONE provides advanced features such as a file system and a network shield to protect the integrity and confidentiality of storage and network interaction, respectively.

Interestingly, performance evaluations from several project partners show that the SCONE-powered variant can, in some scenarios, outperform native execution. This is largely attributed to SCONE asynchronous system-call mechanism and its optimized thread-management model. When an enclave is instantiated, SCONE allocates two types of threads: an internal enclave thread (`ethread`) and a system-call handling thread (`sthread`). These correspond to the *M:N threading* and *asynchronous system-call interface* components depicted in Fig. 11.

7.6.3 Key Features

- Transparent secure execution of applications and functions with TEE support.
- Integration with C-Cells for confidential workloads.
- Provide confidential computing support to support serverless (`Lithops`) and streaming processing (Pravega).

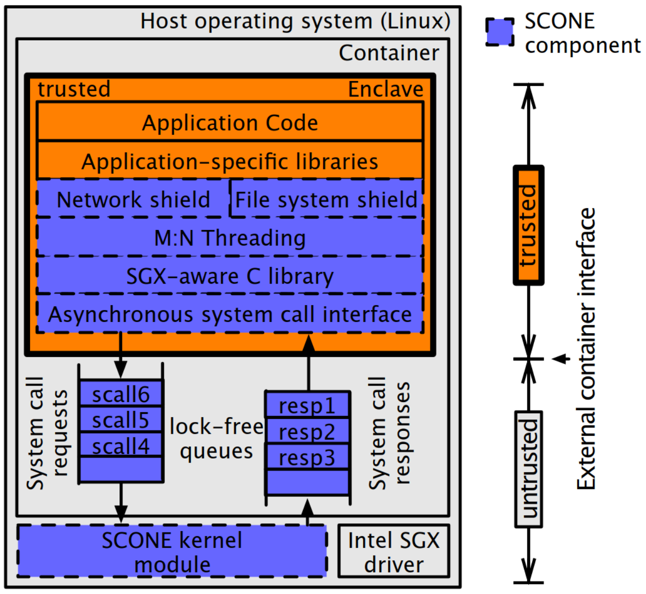

Figure 11: SCONE architecture [5].

7.6.4 Interfaces and Integration

SCONE enables transparent conversion of native applications to run inside a TEE, ensuring both integrity and confidentiality of code and data within the secure memory region. This lift-and-shift capability has been applied to the C-Cell runtime, allowing C-Cell workloads to execute inside an Intel SGX enclave and remain protected from a powerful external adversary.

Beyond the application layer, SCONE has also been integrated into several components of the supporting platform across the architecture. In collaboration with URV and DELL, we enabled confidential execution of Lithops Serve workers and Pravega servers. In the Metabolomics use case, SCONE secures the execution of Lithops Serve workloads, which includes `PyTorch` dependencies. The same applies to Pravega Controller and Segment store, which are core parts of their architecture.

In the surgery use case, SCONE has been applied to the two frameworks in use in data pipeline: ROS (Robot Operating System) and GStreamer[11], a multimedia streaming framework. This guarantees that model inference is run confidentially, safeguarding both data-in-use and the processing pipeline. We also conducted a controlled, Python-only experiment focused exclusively on inference to assess performance overhead. These results demonstrate that integrating SCONE into the computer-assisted surgery workflow is both feasible and flexible, enabling protection at multiple layers of the application stack.

7.7 GEDS WebAssembly-based Units

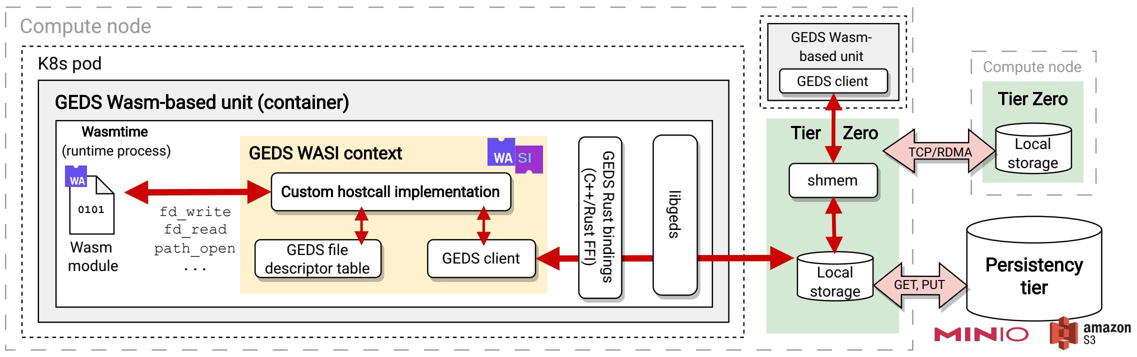

Figure 12: GEDS WebAssembly-based Units architecture.

[11] https://gstreamer.freedesktop.org/

7.7.1 Overview

WebAssembly [1] is a bytecode format that promises **high performance**, **portability**, and **lightweight isolation**. These properties are delivered thanks to a simplified memory management, a close-to-native bytecode and a minimalist interface for the interaction with system resources (WASI, short for WebAssembly System Interface). While these properties help shape Wasm's strengths, they also hinder legacy application extensibility and I/O-heavy use cases. To overcome these limitations and provide Wasm modules with access to GEDS, we propose **GEDS WebAssembly-based Units**.

GEDS WebAssembly-based Units (GEDS Wasm Units) are a custom Wasm execution environment that empowers legacy Wasm applications transparently with GEDS' storage tiering and fast data migration capabilities for near-data and fast sandboxed computations. To do so, GEDS Wasm Units leverage a novel approach, **WASI hostcall interposition**, to intercept standard I/O hostcalls from Wasm and redirect them transparently to GEDS.

GEDS Wasm units are a deep integration that takes place between Wasm modules and the compute node: in the Wasm runtime process. By working at the runtime level, this integration achieves better performance than external approaches (e.g., FUSE filesystems), and overcomes Wasm's extensibility limitations, allowing existing applications to utilize GEDS without rewriting large portions of code.

GEDS Wasm Units not only cover the integration of GEDS with Wasm modules, but also provides significant benefits to legacy Wasm applications. Firstly, they extend WASI's minimalist I/O interface with advanced storage capabilities. Secondly, they allow bypassing legacy Wasm modules' 4 GB memory limit. Finally, they allow for efficient communication of Wasm applications through shared memory and RDMA, two capabilities that are not available in Wasm.

7.7.2 Architecture

GEDS Wasm units are implemented on top of Wasmtime [13], an industry-standard Wasm runtime written in Rust. To allow the interaction of GEDS, implemented in C++, with Wasmtime, we provide a set of Rust bindings for GEDS.

Figure 12 depicts the architecture of GEDS Wasm Units, which are made up of two main components:

- **Runtime process**. The runtime process is responsible for executing Wasm applications. For portability reasons, Wasm-compiled applications do not contain any logic referring to system calls (hostcalls in WASI). Instead, when executing a Wasm application in a runtime, the runtime provides this implementation in the form of a WASI context. By supplying a custom WASI context that replaces the implementation of specific hostcalls, we can extend Wasm transparently. We have named this mechanism **WASI hostcall interposition**, and is also performed by the runtime process.

- **GEDS WASI context**. GEDS Wasm units implement a GEDS WASI context that modifies file I/O hostcalls (i.e., `fd_write`, `fd_read`, `path_open`, `fd_close`, etc.) to achieve the integration of GEDS with Wasm. An example of this is found in Listing 9, which depicts our custom implementation of the `fd_write` hostcall, similar to the `write` syscall. The GEDS WASI context is not destructive; the original implementation of the WASI hostcall can also be used if the interaction with the module's local storage, instead of GEDS, is required.

GEDS Wasm Units are the perfect match for computational storage: they combine the lightweight isolation of WebAssembly with a tiered storage layer closely located to each other. Wasm modules gain access to GEDS' fast in-memory Tier Zero storage layer, efficient communication between module instances through RDMA and shared memory, and persistence of the data, an important feature in containerized environments where local storage is often ephemeral. Wasm's low startup latency allows for on-the-fly data transformations, the results of which can be quickly cached and stored in GEDS.

Listing 9: A code snippet of the custom GEDS context which showcases the modified `fd_write` to allow the interaction of the Wasm runtime with GEDS.

```
async fn fd_write(
        &mut self,
        mem: &mut GuestMemory<'_>,
        fd: Fd,
        iovs: CiovecArray,
    ) -> Result<Size, Error> {
    // Check if the file descriptor corresponds to a GEDS object
    if self.geds_descriptors.contains_key(&u32::from(fd)) {
        let geds_file = self.geds_descriptors.get(&u32::from(fd)).unwrap();

        // Initialize memory buffer
        let buf = first_non_empty_ciovec(mem, iovs)?;
        let buf = mem.to_vec(buf)?;

        // Write to GEDS
        return match geds_file.write(&buf, 0, buf.len()) {
            Ok(()) => Ok(u32::try_from(buf.len())?),
            Err(_) => {
                Err(Errno::Fault.into())
            }
        };
    }
    // Fallback to original WASI context for non-GEDS files (i.e, local filesystem)
    self.inner.fd_write(mem, fd, iovs).await
}
```

7.7.3 Key Features

- On-the-fly data transformations.
- Near-data execution and low-latency compute.
- Virtualization of memory for secure multi-tenant execution

7.7.4 Interfaces and Integration

In the framework of the project, GEDS Wasm Units bridge the gap between Wasm runtimes and GEDS, enabling existing, pre-compiled legacy Wasm applications to interact with GEDS transparently. With this, we provide applications access to tiered storage (in-memory, local disk, persistent storage) in the form of a modified Wasm runtime. On the other hand, GEDS Wasm units represent a complementary compute layer abstraction that can coexist and cooperate with others, such as C-Cells and Lithops executors.

From a developer's perspective, GEDS Wasm Units do not provide an interface per se. A Wasm application running in an unmodified Wasm runtime (e.g., Wasmtime) that performs file I/O operations will interact with local storage, while the same application running in a GEDS Wasm Unit will automatically interact with GEDS without requiring any changes in the codebase.

7.8 Pravega

7.8.1 Overview

Pravega is an open-source, distributed storage system designed specifically for *data streams*. Unlike traditional event streaming systems that primarily buffer events, Pravega introduces a tiered storage architecture that combines a low-latency write-ahead log (WAL) with scalable long-term storage (LTS), enabling applications to retain streaming data for extended periods in a cost-effective way. Streams in Pravega are durable, elastic, and append-only sequences of bytes, internally divided into *segments* for parallelism. This architecture supports high ingestion throughput, strong consistency guarantees (no duplicates or missing events, per-key ordering), and efficient historical reads, making Pravega a robust substrate for modern streaming pipelines.

A distinguishing feature of Pravega is *elastic streams*, which automatically adjust their parallelism degree based on workload fluctuations. This auto-scaling capability reduces operational complexity and ensures balanced resource utilization across compute and storage layers. Pravega also integrates storage tiering into the ingestion path, unlike systems that rely on best-effort offloading, guaranteeing predictable performance and durability. Furthermore, Pravega supports multiple APIs on top of data streams beyond event streaming, including byte-oriented streams, key-value tables, and state synchronization primitives. These APIs enable heterogeneous workloads such as large file transfers, multimedia pipelines (e.g., GStreamer video analytics), and distributed coordination, all within a unified streaming framework.

7.8.2 Architecture

The architecture of Pravega is shown in Fig. 13. First, Pravega offers client libraries implementing the server APIs, such as *writers* and *readers*, which interact with Pravega server instances either within the same cluster or externally. These client libraries allow stream processing engines to manage *data events* from Pravega.

On the server side, we find the Pravega *control plane* formed by *controller instances*. The control plane is primarily responsible for orchestrating all stream lifecycle operations, like creating, updating, scaling, and

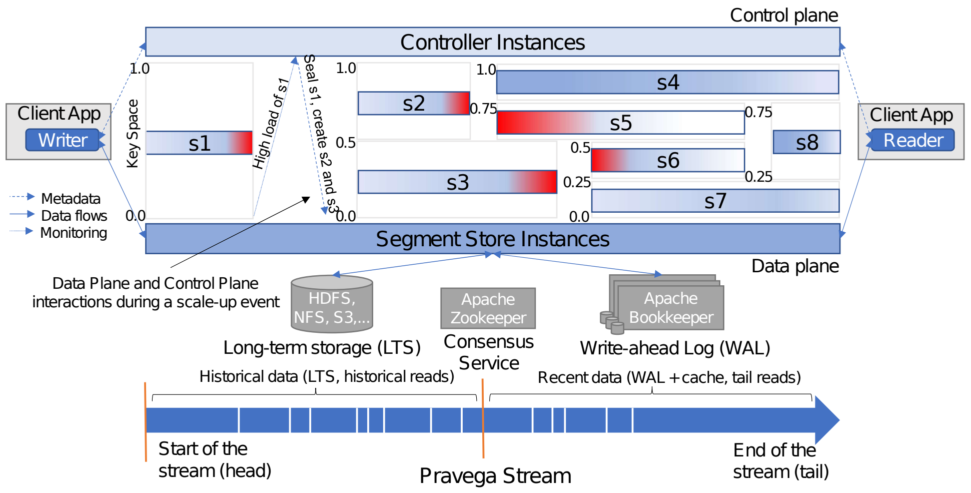

Figure 13: Architecture of Pravega consisting of clients, controllers, and segment store instances. Segment stores temporarily write data to WAL and then move the data to LTS. The figure also shows an example of stream auto-scaling.

deleting streams. Pravega streams are policy-driven. Currently, the system offers two types of *stream policies*: *retention policies*, which automatically truncate a stream based on size or time bounds; and *auto-scaling policies*, which allow the system to automatically change the segment parallelism of a stream based on the ingestion workload (events/bytes per second). The control plane takes care of enforcing such stream policies. For stream auto-scaling policies, Pravega builds a feedback loop between the control and data planes, so the control plane can react to the load monitored by the data plane.

The *data plane* in Pravega handles data requests from clients and is formed by *segment store instances*. Segment stores play a critical role in making segment data durable and serving it efficiently. Note that segment stores only work with segments and are agnostic to the concept of stream, which is an abstraction of the control plane. The data plane distributes the segment-related load based on *segment containers*. Segment containers perform the heavy lifting on segments and the main role of segment store instances is to host segment containers. A segment is mapped during its entire life to a segment container using a stateless, uniform hash function that is known by the control plane. Thus, segment ids belong to a *key-space* that is partitioned across the available segment containers.

The segment store has two storage tiers: *Write-Ahead Log (WAL)* and *Long-Term Storage (LTS)*. The main goal of WAL (implemented via Apache Bookkeeper [14, 15]) is to guarantee durability and low latency of incoming writes and keep that data temporarily for recovery purposes. Segment stores asynchronously move data to LTS. Once some data is stored in LTS, the corresponding log file from WAL is truncated. Pravega has an LTS tier for a couple of key assumptions that determined its design: data streams are potentially *unbounded* and the system should be able to store a large number of segments in a *cost-effective manner*. Pravega achieves both goals by storing historical stream data in a scalable storage service.

Finally, Pravega uses a consensus service (Apache Zookeeper [16, 17]) for leader election and general cluster management purposes.

7.8.3 Key Features

- Unified real-time streaming and batch analytics support.
- Elastic streaming storage fabric adapting to workload changes.
- Low-latency edge-cloud data access.

Listing 10: GStreamer pipeline for phase detection NCT AI model

```
"""
GStreamer pipeline to read video from a Pravega stream and perform inference via
    the "phase detection" NCT AI model
"""
gst-launch-1.0 \
-v \
pravegasrc \
  allow-create-scope=${ALLOW_CREATE_SCOPE} \
  buffer-size=1024 \
  controller=${PRAVEGA_CONTROLLER_URI} \
  keycloak-file=\"${KEYCLOAK_SERVICE_ACCOUNT_FILE}\" \
  stream=${PRAVEGA_SCOPE}/${PRAVEGA_STREAM_1} \
  $* \
! decodebin \
! videoconvert \
! NN_phase_plugin \
! videoconvert \
! x264enc tune=zerolatency key-int-max=30 speed-preset=medium \
! h264parse \
! video/x-h264,alignment=au \
! mpegtsmux \
! pravegasink \
  stream=${PRAVEGA_SCOPE}/${PRAVEGA_STREAM_2} \
  allow-create-scope=${ALLOW_CREATE_SCOPE} \
  controller=${PRAVEGA_CONTROLLER_URI} \
  keycloak-file=\"${KEYCLOAK_SERVICE_ACCOUNT_FILE}\" \
  seal=false sync=false timestamp-mode=realtime-clock
```

7.8.4 Interfaces and Integration

In CloudSkin, Pravega is positioned as a storage substrate for streaming analytics and event-driven applications across the cloud–edge continuum.

Technology wise, we have demonstrated **integrations of Pravega** with various CloudSkin components. First, we integrated Pravega and GEDS to mitigate the impact of long-term storage outages in streaming ingestion process of Pravega (see D3.3). Second, we have "sconified" Pravega clients and evaluated the performance impact of running streaming clients on TEEs for latency-sensitive applications (see D4.3). Moreover, we demonstrate how we can auto-scale Pravega instances with predictive methods in D5.4, which paves the way for integration with the CloudSkin Learning Plane. Finally, in D3.4, we present Nexus, a novel data management system for tiered data streams, and how it can add value via data management functions to the storage tiering process of Pravega across the cloud-edge contiuum.

For use cases, Pravega is the key streaming storage foundation supporting the **Computer-Assisted Surgery (CAS)** use case. We have built a streaming platform PoC for ingesting and surgical video data at scale (see D2.3). Pravega guarantees sub-10ms video frame IO latency and automatic management of historical surgical video data. Moreover, we augmented this streaming platform with predictive auto-scaling models that handle the fluctuating demands of NCT surgery room usage patterns while minimizing impact to NCT's real-time AI inference jobs (see D5.2).

In practice, NCT data scientists use Pravega streams in their pipelines as shown in Listings 10. In this GStreamer pipeline, we can see how the input video is read from `PRAVEGA_SCOPE/PRAVEGA_STREAM_1`. This input stream may be receiving real-time video from an endoscopic camera, or already contain historical video data. In any case, the pipeline defines that the video is processed by the `NN_phase_plugin`, which executes inference on video frames using the NCT "phase detection" model via Pytorch. Finally, the pipeline defines that the output of the AI model is stored on another Pravega stream named `PRAVEGA_SCOPE/PRAVEGA_STREAM_1`. In addition to providing a simple programming model for NCT data scientists to perform AI video inference on real time and in batch, these pipelines have been containerized. This fosters code re-use and facilitates deployment of NCT AI models across Cloud-edge environments.

7.9 Nexus

7.9.1 Overview

Nexus is a data management mesh designed to bridge the gap between event streaming systems and external storage services during tiered storage operations. Modern streaming platforms such as Apache Kafka, Pulsar, and Pravega support tiered storage to offload cold data chunks to cost-efficient object stores. However, these mechanisms typically perform simple data transfers without enabling advanced data management. Nexus introduces a programmable layer that transparently intercepts storage operations and applies user-defined functions—called *streamlets*—on chunks of tiered stream data across the cloud-edge continuum.

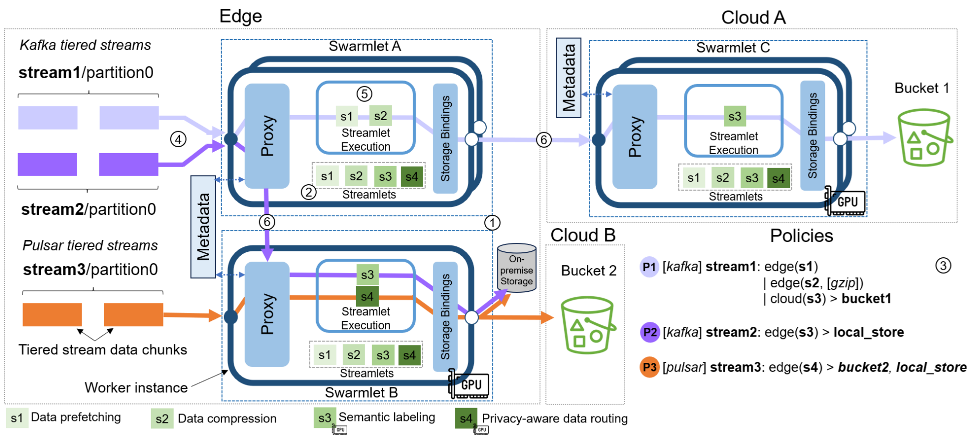

Figure 14: Architecture overview and operation of Nexus.

The core idea behind Nexus is to decouple data management from the streaming hot path while maintaining transparency and extensibility. Streamlets can perform diverse tasks such as compression, encryption, caching, semantic annotation, and privacy-aware routing. These functions are orchestrated through policies and executed across heterogeneous infrastructures (edge and cloud) using clusters of worker nodes called *swarmlets*. This design allows administrators to build rich, location-aware data management pipelines without modifying the underlying streaming system.

7.9.2 Architecture

The architecture of Nexus is organized around three key abstractions: *streamlets*, *swarmlets*, and *policies* (see Fig. 14). A streamlet is a reactive function executed inline on intercepted storage requests for tiered stream data chunks. Streamlets are categorized into four types: (i) *transformers* for content modification (e.g., compression, encryption), (ii) *performance-oriented* for caching and prefetching, (iii) *routing* for multi-cloud replication or privacy-aware placement, and (iv) *semantic* for AI-based content analysis. Streamlets can be stateless or stateful, with Nexus providing primitives for persistent state management via metadata services.

Swarmlets represent clusters of homogeneous worker instances deployed across edge or cloud environments. Each swarmlet exposes a standard API endpoint (*e.g.*, S3-compatible) to intercept storage operations transparently. Worker instances execute streamlets in containerized environments, supporting isolation and dynamic loading of binaries. Policies define the orchestration logic for streamlet pipelines, specifying execution order, location constraints, and hardware requirements (*e.g.*, GPU for AI inference). Policies also determine the final storage destination after processing.

Nexus employs a mesh-like routing mechanism to enforce policy constraints and optimize execution. Partition-aware routing ensures deterministic assignment of stream partitions to worker nodes, enabling efficient stateful streamlet execution without global synchronization. Additionally, Nexus supports location- and hardware-based routing to satisfy policy requirements across heterogeneous infrastructures. Fault tolerance is achieved by acknowledging storage requests only after successful pipeline execution, preserving end-to-end data integrity. Overall, this architecture enables Nexus to deliver programmable, in-transit data management for tiered streams while maintaining transparency and scalability.

7.9.3 Key Features

- Dynamic buffering for network variability
- Semantic annotation of storage chunks
- On-the-fly transformations (compression, encryption)
- Intelligent routing based on policies

7.9.4 Interfaces and Integration

In CloudSkin, Nexus stands as data management mesh to allow executing advanced functions over chunks of tiered stream data across the cloud–edge continuum. In this sense, while Nexus primary use case is Pravega, it can benefit any streaming system (*e.g.*, Kafka, Pulsar, RedPanda) that implements S3 bindings for offloading

Table 14: Summary of Nexus Streamlet APIs (developer-facing).

API	Purpose (examples)	Key methods
`ByteStreamlet`	Raw-byte processing at tiering boundary. Examples: compression (GZip), encryption, format conversion.	`processPutBytes`, `processGetBytes`
`DataSourceStreamlet`	Intercept/serve the data source on `GET`. Examples: buffering under outages, prefetching, cache redirect.	`handlePreGet`
`EventStreamlet<T>`	Record-level processing after deserialization. Examples: AI-driven annotation, content filtering, semantic indexing.	`processPutRecord`, `processGetRecord`
`Deserializer<T>`	Convert chunk bytes into typed records. Examples: JPEG frames, FASTQ reads.	`deserializeChunk`
`@Persistent`	Annotate state to persist across executions. Examples: per-partition indexes, routing maps.	(annotation on data structures)

stream data to external storage (see D3.4). In this sense, we demonstrate the value that Nexus can add in terms of data management across the cloud-edge continuum via the **Computer-Assisted Surgery (CAS)** use case (see D5.4). In particular, we implemented a "storage buffering streamlet" to provide long-term storage disconnection resilience in Pravega, achieving results comparable to those previously obtained with GEDS. Additionally, we developed a "surgical annotation" streamlet that leverages NCT AI models to annotate data objects based on the content of surgical images.

Listing 11: Compression streamlet implementation example

```
"""
Streamlet implementation performing compression in processPutBytes method
"""
@Override
protected void processPutBytes(StreamletIO dataStreams, StreamletContext context)
    {
    try (InputStream input = dataStreams.input();
         OutputStream output = new GZIPOutputStream(dataStreams.output())) {
        byte[] buffer = new byte[8192];
        int bytesRead;
        while ((bytesRead = input.read(buffer)) != -1) {
            output.write(buffer, 0, bytesRead);
        }
    } catch (IOException e) {
        throw new RuntimeException("Error compressing data", e);
    }
}
```

From a developer viewpoint, Nexus API is shown in Table 14. This API allows developers to create streamlets that work on raw streams of bytes (`ByteStreamlet`) or on the actual event contents by deserializing (`Deserializer`) them via the `EventStreamlet`. A simple code example of a Nexus streamlet that performs data compression can be seen in Listings 11. Visibly, this streamlet implements `processPutBytes` defined in the `ByteStreamlet` and uses the `GZIPOutputStream` library to write compressed data to the `OutputStream`. As a result, chunks of stream data intercepted by Nexus will be compressed and stored transparently in external storage.

8 Conclusions

The reference architecture presented provides a practical overview of the implemented CloudSkin platform, detailing the key components, their roles, and interactions across the cloud-edge continuum. It demonstrates how the integrated system meets all functional requirements, including secure execution, elastic and portable compute, multi-site orchestration, and advanced data management capabilities. By documenting component architectures, interfaces, and integration patterns, this reference architecture validates the design decisions made during implementation and serves as a blueprint for understanding, maintaining, and extending the platform. It also highlights the alignment between functional requirements, software layers, and real-world use cases, confirming that the platform operates as intended across diverse cloud and edge environments.

References

[1] A. Zakai, A. Haas, A. Rossberg, B. Titzer, D. Gohman, D. Schuff, J. Bastien, L. Wagner, and M. Holman, "Bringing the web up to speed with webassembly," in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), (Barcelona, Madrid), 2017.

[2] T. F. . Architecture, "Developing a reference architecture for the continuum - concept, taxonomy and building blocks," Oct. 2023.

[3] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in 33rd ACM Transactions on Computer Systems (TOCS), ACM, 2015.

[4] "Intel software guard extensions," 2022.

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure linux containers with intel SGX," in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), (Savannah, GA), pp. 689–703, USENIX Association, 2016.

[6] N. Computing, "Nearbyone edge orchestrator." `https://www.nearbycomputing.com/wp-content/uploads/2021/08/NearbyOne-Product-Data-Sheet-v2.0.pdf`, 2021.

[7] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona, "Toward multicloud access transparency in serverless computing," IEEE Software, vol. 38, no. 1, pp. 68–74, 2021.

[8] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in 2020 USENIX Annual Technical Conference (USENIX ATC 20), pp. 419–433, USENIX Association, July 2020.

[9] P. Liu, G. Bravo-Rocca, J. Guitart, A. Dholakia, D. Ellison, and M. Hodak, "Scanflow: An end-to-end agent-based autonomic ml workflow manager for clusters," in Proceedings of the 22nd International Middleware Conference: Demos and Posters, Middleware '21, (New York, NY, USA), p. 1–2, Association for Computing Machinery, 2021.

[10] P. Liu, G. Bravo-Rocca, J. Guitart, A. Dholakia, D. Ellison, and M. Hodak, "Scanflow-k8s: Agent-based framework for autonomic management and supervision of ML workflows in kubernetes clusters," in 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 376–385, 2022.

[11] A. P. P. Hartono and C. Fetzer, "Brofy: Towards essential integrity protection for microservices," in 2021 40th International Symposium on Reliable Distributed Systems (SRDS), pp. 154–163, 2021.

[12] A. P. P. Hartono, A. Brito, and C. Fetzer, "Crisp: Confidentiality, rollback, and integrity storage protection for confidential cloud-native computing," in 2024 IEEE 17th International Conference on Cloud Computing (CLOUD), pp. 141–152, 2024.

[13] B. Alliance, "Wasmtime." `https://wasmtime.dev`, mar 2022.

[14] F. P. Junqueira, I. Kelly, and B. Reed, "Durability with bookkeeper," ACM SIGOPS operating systems review, vol. 47, no. 1, pp. 9–15, 2013.

[15] "Apache bookkeeper." `https://bookkeeper.apache.org`, 2023.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems.," in USENIX ATC'10, vol. 8, 2010.

[17] "Apache zookeeper." `https://zookeeper.apache.org`, 2023.