CloudSkin

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

# CloudSkin

(grant agreement No 101092646)

## Adaptive virtualization for AI-enabled Cloud-edge Continuum

## D3.4 Reference implementation of Cloud native Infrastructure

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** | 25 |
| **WP/Task related to this document** | WP3 / T3.1, T3.2, T3.3 |
| **WP/Task responsible** | IBM |
| **Leader** | Robert Haas (IBM) |
| **Technical Manager** | Raúl Gracia-Tinedo (DELL) |
| **Quality Manager** | Raúl Gracia-Tinedo (DELL) |
| **Author(s)** | Raúl Gracia-Tinedo (DELL), Hossam Elghamry (DELL), Alan Cueva (DELL), Pablo Gimeno Sarroca (URV), Marc Sanchez-Artigas (URV), Radu Stoica (IBM) |
| **Partner(s) Contributing** | IBM, DELL, URV |
| **Document ID** | CloudSkin_D3.4_Public.pdf |
| **Abstract** | Final report that details design of mechanisms and evaluation results for the outcomes of each task of WP3, focused on the CloudSkin dataplane infrastructure and platforms. This report will also include the release of produced frameworks and technologies through open source publication platforms. |
| **Keywords** | Key-value store, ephemeral data, persistency, elasticity, object store, storage tiering, long-term storage, WebAssembly, streaming, data pipelines, cloud, edge. |

# History of changes

| Version | Date | Author | Summary of changes |
|---|---|---|---|
| 0.1 | 19-12-2025 | Raúl Gracia-Tinedo (DELL), Hossam Elghamry (DELL), Alan Cueva (DELL), Pablo Gimeno Sarroca (URV), Marc Sanchez-Artigas (URV), Radu Stoica (IBM) | First draft of all sections |
| 0.2 | 21-12-2025 | Pablo Gimeno Sarroca (URV) | Updated GEDS Wasm section |
| 0.3 | 23-12-2025 | Marc Sanchez-Artigas (URV) | Full review of the deliverable |
| 1.0 | 26-12-2025 | Radu Stoica (IBM) | Review and final version. |
| 1.1 | 27-02-2026 | Marc Sanchez-Artigas (URV) | Minor fix post-review. |

## Table of Contents

## List of Abbreviations and Acronyms

| | |
|---|---|
| **API** | Application Programming Interface |
| **AWS** | Amazon Web Services |
| **CC** | Creative Commons |
| **CLI** | Command Line Interface |
| **CSV** | Comma-separated values |
| **DOI** | Digital Object Identifier |
| **EKS** | Elastic Kubernetes Service |
| **FFI** | Foreign Function Interface |
| **FPS** | Frames Per Second |
| **FUSE** | File-system in User Space |
| **GEDS** | General Ephemeral Data Store |
| **GPU** | Graphics Processing Unit |
| **HDFS** | Hadoop Distributed File System |
| **HTTP** | Hypertext Transfer Protocol |
| **I/O** | Input/Output |
| **JPEG** | Joint Photographic Experts Group |
| **MPI** | Message Passing Interface |
| **NCT** | Network-Centric Therapy |
| **OS** | Operating System |
| **POSIX** | Portable Operating System Interface |
| **RDMA** | Remote Direct Memory Access |
| **S3** | Simple Storage Service |
| **SHMEM** | Shared Memory |
| **SRA** | Sequence Read Archive |
| **TEE** | Trusted Execution Environment |
| **VM** | Virtual Machine |
| **WAL** | Write-Ahead Log |
| **WASI** | WebAssembly System Interface |
| **Wasm** | WebAssembly |

## 1 Executive summary

This deliverable presents an updated and consolidated overview of the CloudSkin dataplane platform activities, focusing on the storage systems developed, including their implementation and architecture, functionality, APIs and use-case integrations. The work performed is within the context of **WP3: Infrastructure Support for the Virtual Cloud-Edge Continuum**.

The document first summarizes the existing GEDS progress as described previously in deliverables D3.1, D3.2 and D3.3. It then presents two further complementary contributions to advance programmable, data-intensive processing across the cloud–edge continuum and Learning Plane: 1) GEDS-based WebAssembly Units, and 2) Nexus, a programmable data management mesh evolving from the GEDS experience, to further enable transparent, near-data computation shipping and execution. Together, the two GEDS and Nexus systems address fundamental limitations in today's cloud–edge data processing stacks by enabling computation closer to data, transparent storage tiering, and policy-driven data transformation.

GEDS-based WebAssembly Units extend WebAssembly beyond its current WASI limitations by integrating GEDS directly at the runtime level. By interposing WASI hostcalls inside the Wasm runtime, legacy Wasm modules gain transparent access to hierarchical storage tiering, persistent and shared storage, and efficient local and remote communication mechanisms (including shared memory and RDMA), without rewriting or recompiling applications. This approach preserves Wasm's core advantages, namely portability, isolation, and security, while overcoming key constraints such as limited I/O expressiveness, ephemeral storage, and memory limitations. The resulting dual-runtime architecture complements the existing C-Cells runtime, enabling the same Wasm binaries to serve both distributed compute workflows and fine-grained, I/O-centric data transformations executed close to storage.

Building on these foundations, Nexus generalizes the earlier GEDS–Pravega integration into a fully programmable data management mesh for tiered stream data. Nexus decouples the data ingestion path from the persistency backend and exploits the chunk-level tiering boundary to execute user-defined data management functions, or streamlets, transparently across the Cloud–Edge continuum. Through the abstractions of streamlets, swarmlets, and policies, Nexus enables buffering under storage outages, compression, semantic annotation using AI accelerators, and privacy-aware routing, all without modifying event streaming systems or impacting real-time ingestion latency. Mesh-like, partition-aware routing ensures consistency and scalable state management across heterogeneous infrastructures.

Taken together, GEDS and Nexus represent the consortium's deliverables for WP3, in line with the description of the tasks T3.1, T3.2, and T3.3.

# 2 GEDS overview

In this section, we provide a summary of the motivation behind GEDS and an overview of the main features developed during the whole duration of the whole project, including deliverables D3.1, D3.2, and D3.3.

## 2.1 Background

The rapid evolution of cloud-native architectures, serverless computing, and cloud-edge compute continuum has introduced a significant demand for efficient management of ephemeral data. Ephemeral data, defined as short-lived, intermediate results, constitutes the bulk of I/O traffic in modern data-intensive workloads, as shown in D3.1. Unlike persistent data, which requires long-term durability and is well-served by persistent object stores like Amazon S3, ephemeral data demands extremely high throughput and low latency, often with relaxed durability requirements as it can be recomputed if lost.

Traditionally, ephemeral data is managed either through local file systems, which lack distributed accessibility, or via persistent storage services, which introduce unnecessary overhead and increase latency. High-performance systems like Apache Crail [1] pioneered the use of RDMA-based ephemeral storage; unfortunately, they often require specialized hardware and lack the flexibility needed for heterogeneous Cloud-Edge environments. GEDS (Generic Ephemeral Data Store) addresses these gaps as a distributed, multi-tiered storage system designed to provide a unified, high-performance namespace for ephemeral objects across the cloud-edge continuum.

## 2.2 Motivation and Problem Statement

The design of GEDS is motivated by three primary challenges in modern distributed computing:

1. **I/O Bottlenecks in multi-stage workloads:** In complex data-intensive pipelines, the time spent exchanging (writing and reading) intermediate results to persistent storage often exceeds the actual computation time.

2. **Resource heterogeneity:** Cloud-edge environments feature a mix of higher-end hardware and resource-constrained edge nodes. A storage solution must scale out across these tiers without requiring uniform specialized hardware (like RDMA-enabled networking) at every endpoint.

3. **Data spillage and elasticity:** While DRAM is the ideal medium for ephemeral data, it is also a scarce and expensive resource. Systems must gracefully handle "spilling" data to slower tiers (NVMe or S3) when memory is exhausted without interrupting application logic.

## 2.3 GEDS Architecture and Design

GEDS is designed as a lightweight, application-integrated storage library rather than a heavy-weight persistent service. This "daemon-less" approach (at the data path level) minimizes context switching and allows for **zero-copy data access**.

### 2.3.1 Multi-Tiered Storage Hierarchy

GEDS organizes storage into three distinct tiers to balance performance and cost:

- **Tier 0 (Performance Tier):** Implements local node DRAM or memory-mapped storage. Data access in this tier is direct, allowing applications to read and write at memory speeds. Remote reads are handled via a high-efficiency TCP-based protocol.

- **Tier 1 (Capacity/Disaggregated Tier):** Leverages local or disaggregated block storage (e.g., enterprise NVMes). This tier serves as an intermediate cache for data that exceeds the DRAM capacity but is still frequently accessed.

- **Tier 2 (Persistency/LTS Tier):** Integrates with S3-compatible object stores for long-term storage (LTS). GEDS can asynchronously "spill" objects to this tier based on Least Recently Used (LRU) policies, ensuring that the performance tier remains available for hot data.

#### 2.3.2 Metadata Service (MDS)

GEDS employs a centralized Metadata Service (MDS, also known as the NameNode) to manage the global namespace. The MDS maintains information about file/object locations and replicas and their metadata, while remaining entirely out of the data path. This separation ensures that data transfers occur directly between the requesting application and the storage node holding the data (or the underlying S3 store), maximizing throughput and minimizing data movement.

### 2.4 Key Features and Interoperability

A core strength of GEDS is its focus on "genericity," achieved through extensive language support and standard interface integration.

#### 2.4.1 Zero-Copy Integration

By integrating directly into application memory, GEDS allows for zero-copy operations. For instance, in Python environments, GEDS read calls can directly allocate writable Python buffers, avoiding the overhead of intermediate data copying between the storage library and the application space.

#### 2.4.2 Language Bindings and APIs

- **Native C++:** The core GEDS engine is implemented in C++ for maximum performance and predictability. GEDS C++ can be used and linked to any existing C/C++ application as a typical shared library.
- **Python Integration:** GEDS provides a native Python API (based on PyBind11) and integrates with the `smart_open` Python library, allowing data scientists and developers to use GEDS as a drop-in replacement for standard file I/O.
- **Java and Hadoop Ecosystem:** GEDS provides an HDFS-compatible `FileSystem` interface via Java Native Interface (JNI). This allows existing big data frameworks like Apache Spark to use GEDS for shuffle operations and temporary data storage without code modifications.

#### 2.4.3 Elasticity and Resilience

GEDS supports dynamic scaling of both metadata and storage resources. Its ability to transparently relocate data to an object store allows applications to handle datasets much larger than the available aggregate cluster memory. Also, recent integrations with tools such as Pravega demonstrate GEDS ability to act as a high-speed buffer that tolerates transient unavailability of long-term storage (LTS) services.

### 2.5 Use Cases and Performance Evaluation

Performance microbenchmarks indicate that GEDS can saturate 200Gbps network links for remote data access, while maintaining latencies comparable to local memory access for Tier 0 operations, as reported in D3.3. To further show real-world benefits, we have validated GEDS for several critical cloud usage scenarios and applications:

1. **Apache Spark shuffle:** By replacing the standard local-disk shuffle with GEDS, Spark jobs benefit from memory-speed data exchange and reduced disk I/O contention.
2. **AI/ML checkpointing:** In deep learning training, GEDS enables asynchronous checkpointing at the speed of local DRAM. An application writes a checkpoint to local Tier 0 storage (at the speed of the combined PCIe interconnect speed of local GPUs), and GEDS handles the background transfer to S3. This fast checkpointing allows the next training epoch to begin immediately.
3. **Edge-to-cloud streaming:** GEDS serves as a performance-tier buffer, absorbing high-velocity data bursts and trickling them into persistent cloud storage as bandwidth permits.

### 2.6 Summary

GEDS represents a significant step toward a unified storage abstraction for the Cloud-Edge continuum. By focusing on the unique requirements of ephemeral data—performance, elasticity, and multi-language support—it provides a robust foundation for next-generation AI and analytics workloads. As an open-source project supported by the CloudSkin framework, we have continued to evolve GEDS, with future work focusing on advanced pub/sub synchronization and further optimization of heterogeneous data paths. The GEDS codebase was open-sourced at `https://github.com/cloudskin-eu/GEDS`.

# 3 GEDS-based WebAssembly Units

To address task T3.3, we designed and implemented a novel WebAssembly (Wasm) runtime tailored to the requirements of data-intensive execution within GEDS. Specifically, this runtime overcomes essential limitations of the current WebAssembly System Interface (WASI), such as restricted access to high-performance storage interfaces, limited support for asynchronous and streaming I/O, and inadequate integration with hierarchical data-tiering mechanisms. These limitations are addressed by introducing native, GEDS-aware I/O primitives. We call these native WebAssembly execution units as **GEDS-based WebAssembly Units**.

Rather than replacing the existing C-Cells distributed runtime, this new solution complements it. The C-Cells runtime is highly specialized for the cooperative execution of distributed, multi-process and multi-threaded applications implemented using MPI and OpenMP standards, enabling efficient parallel computation across cloud–edge resources. However, it is not designed to efficiently support fine-grained I/O-centric workloads and data transformation pipelines that require deep interaction with the storage tier.

To this end, the new runtime focuses on executing I/O-bound Wasm modules responsible for data transformations close to storage, leveraging GEDS tiering support for efficient data placement and movement. Integration between the C-Cells runtime and the new I/O-oriented runtime is achieved by sharing Wasm modules across both runtimes, allowing the same portable binaries to be executed either as part of distributed compute workflows or as standalone data transformation tasks near the data source.

This novel dual-runtime approach enables transparent cloud–edge integration, while preserving the advantages of Wasm, namely portability, isolation, and lightweight execution, combined with the performance advantages of close-to-storage computation and the flexibility of the C-Cells abstraction.

In the following sections, we provide a detailed description of the **GEDS WebAssembly-based Units**, including their architecture, runtime features, integration with the underlying GEDS storage tiers, and an overview of performance results.

## 3.1 Background and motivation

**WebAssembly and WASI**. WebAssembly (Wasm) [2] is a portable binary format that offers lightweight isolation and high performance. While originally designed as an execution environment for web browsers, Wasm's promise of lightweight isolation, portability, and high performance proved useful far beyond the web. The introduction of Wasm standalone runtimes has sparked many new use cases, such as serverless computing [3, 4, 5], cryptography [6, 7], and high-performance computing [8, 9], among others. To support new use cases, and to allow the interaction of Wasm modules running in standalone runtimes with system resources (i.e., file I/O, system clocks, sockets, etc.), the **WebAssembly System Interface** (WASI) [10] was introduced. The idea behind WASI is to enable this interaction **safely** and maintaining the **portability** that characterizes Wasm [11]. For this, WASI introduces a set of **hostcalls**, which roughly translate 1:1 to system calls (syscalls). On **safety**, these hostcalls require special permissions (i.e., WASI capabilities) that can be given to or revoked from Wasm modules as required. On **portability**, modules that are compiled to Wasm targeting WASI do not include any hostcall implementation, but contain symbols that remain unresolved. When running a Wasm module, the implementation is provided by the Wasm runtime. This does not tie Wasm modules to a specific runtime, architecture or operating system. An example of this last point is presented in Figure 1, where a simple "Hello world!" application generates an import for a WASI hostcall, which is later resolved by the Wasm runtime.

Despite the recent advances in Wasm and WASI, their current state raises two significant challenges regarding the integration of GEDS transparently:

- **Poor I/O functionality**. The minimalist, POSIX-like I/O interface of WASI, which is made up of `read`, `write`, `pread` and `pwrite` calls, greatly limits the capabilities of I/O-intensive applications. Additionally, WASI hostcalls entail a greater level of indirection than their syscall counterparts, as the Wasm VM must parse and translate the arguments of all hostcalls to syscalls before

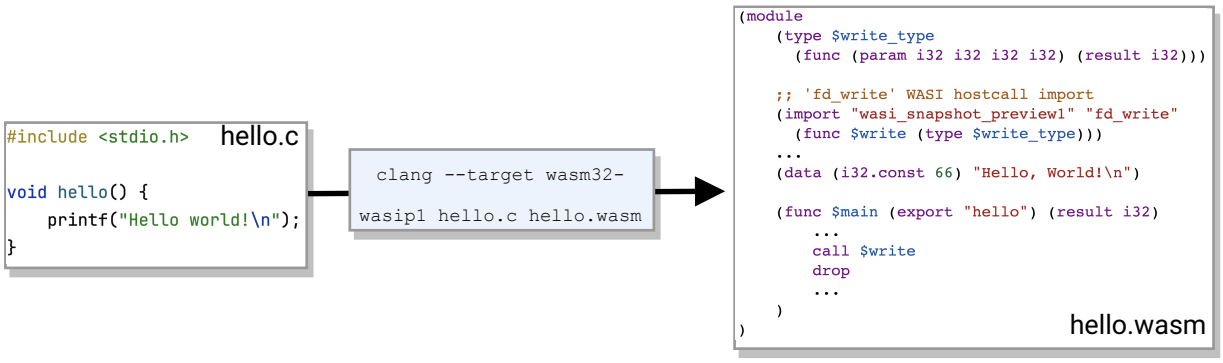


Figure 1: Simple "Hello world!" application compiled to Wasm targeting WASI. hello.c utilizes `printf`, which performs a `write` syscall. This syscall is translated as a `fd_write` hostcall in WASI. The compiled code on the rightmost corner of the figure shows the unsatisfied `fd_write` import symbol.

execution. Additionally, GEDS leverages RDMA for efficient data communication between remote GEDS instances, which is unavailable in Wasm.

- **Limited Wasm extensibility**. Extending legacy Wasm applications is far from straightforward, and often implies rewriting large portions of the source code, which may not always be available. This is because large parts of non-Wasm legacy applications cannot be compiled to Wasm and WASI (e.g., system calls unavailable in WASI, limited memory management, etc.). As an example of this, GEDS relies on memory as a means for temporary Tier Zero storage, utilizing `shmem` and `mmap` calls [12, 13] to efficiently share objects among GEDS instances collocated in the same compute node. This approach cannot be compiled to Wasm directly and must be rewritten to use Wasm's basic linear memory.

- **Storage ephemerality**. Wasm applications often run in containerized environments [14, 15], where local storage is ephemeral and is isolated. GEDS relies on local storage as one of the caching layers for its Tier Zero, and is used to share data among GEDS instances in the same node.

Given these limitations, it is natural to decouple GEDS from Wasm. For this, we explore an integration *outside* Wasm. To understand the possibilities of an integration at this level, we review two common approaches that handle this integration transparently: syscall interposition and FUSE.

**Syscall interposition**. System call (syscall) interposition [16] allows replacing the implementation of system calls by interposing a custom implementation. In Linux systems, this can be achieved by leveraging the dynamic linking and symbol resolution order to inject custom interception functions, This is kwown as the `LD_PRELOAD` trick [17]. A dynamic linked binary relies on loading shared libraries to locate missing symbols. The dynamic linker searches for the shared libraries in the system library paths, loads them into the process address space and tries to satisfy the missing symbols. `LD_PRELOAD` allows specifying paths where alternate implementations of these libraries are stored, and ensures the linker loads them before the libraries located in the system paths. This allows interposing specific functions of the desired libraries, that is, replacing the original implementation of the library's functions with custom implementations.

System call interposition can be used to integrate GEDS transparently: a custom shared library can be loaded to replace the implementation of file I/O syscalls to interact with GEDS instead of the local storage. The integration is transparent to the Wasm modules and no recompilation is required. However, a number of challenges also arise:

- **Low portability**. Syscall interposition is platform-dependent, as each platform has its own distinct mechanism for performing syscall interposition. Thus, this method would tie applications

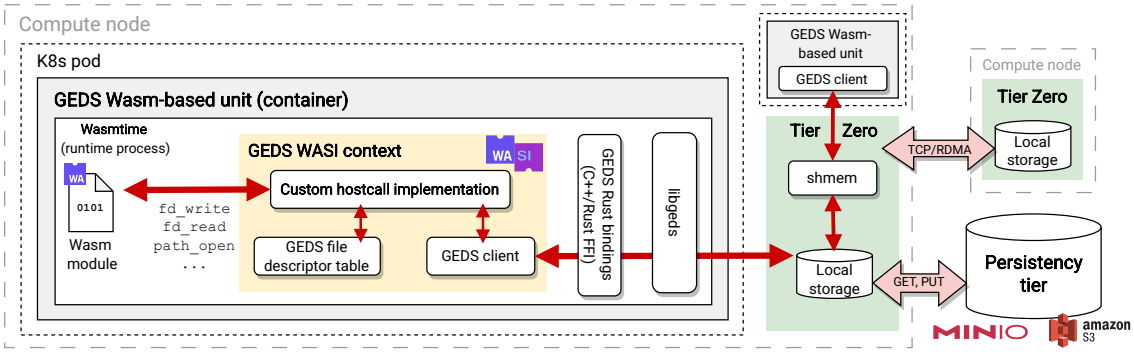


Figure 2: Architecture of GEDS-based WebAssembly Units.

that require GEDS to a specific platform, reducing portability.

- **Lack of permissions**. Isolated environments (e.g., cloud instances) often disallow unprivileged users from performing syscall interposition, since buggy or malicious syscall implementations pose serious security risks.

**FUSE**. Another integration alternative that we considered was based on the **Filesystem in User Space (FUSE)** Linux facility. FUSE is a functionality of the Linux kernel that allows unprivileged users to create their own file systems by executing all file system code safely in user space. Since WASI allows mapping the host's file system to directories visible by the Wasm module, an external durable storage system can be mounted through FUSE and then exposed to the Wasm module. Wasm modules would interact with the local file system through WASI calls, but all file accesses are performed against an external storage system transparently (namely, GEDS). This approach, however, also has a significant flaw: **portability**. As for the integration approach based on `LD_PRELOAD`, FUSE is a Linux-only facility that lacks corresponding implementations on other platforms.

Therefore, after reviewing the existing approaches for the extension of Wasm modules *inside* and *outside* of Wasm, we propose a middle-ground solution that addresses the limitations of both: a deep integration of GEDS and Wasm at the runtime level (§3.2).

### 3.2 GEDS-Wasm integration

To materialize the integration of GEDS and WebAssembly, and to address the limitations of the alternatives that we have presented, we propose **GEDS-based WebAssembly Units**, a deep integration between GEDS and Wasmtime that utilizes **WASI hostcall interposition** (see §3.2.2) to provide legacy Wasm modules with transparent access to GEDS without rewriting nor recompiling existing applications. This integration takes place at the Wasm runtime. This approach avoids both the I/O and extensibility limitations of Wasm with WASI, but also the portability issues introduced by potential integrations outside of Wasm. By adding lightweight WebAssembly sandboxing near the storage layers, GEDS-based WebAssembly Units allow for low-latency computation close to the data.

#### 3.2.1 Integration architecture

The architecture of GEDS-based WebAssembly Units is depicted in Figure 2. From this system diagram, we highlight two main components:

**Runtime.** The runtime is responsible for executing Wasm modules and managing the special linking process of GEDS Wasm Units (i.e., performing WASI hostcall interposition, as in §3.2.2). It is based on Wasmtime [18], an industry-standard WebAssembly runtime that supports WASI. To be able to run GEDS Wasm Units in containerized environments (e.g., Kubernetes over containers), we provide a containerd shim based on runwasi [19], which launches GEDS-based WebAssembly Units in place of regular containers. We design GEDS Wasm Units to be **extensible**, and allow loading cus-

tom WASI contexts to cover other use cases that may benefit from the extension point that we have introduced.

**GEDS WASI context.** The heart of the integration lies in the custom GEDS WASI context. This special WASI context contains all the custom logic needed to interact with GEDS from the Wasm runtime. To do so, it utilizes the `libgeds` shared library to create and destroy GEDS clients and interact with GEDS' metadata service. For maximum efficiency and proximity to the Wasm module, this shared library is loaded together with the Wasm runtime process. To allow the interaction of the runtime, written in Rust, with GEDS, written in C++, we provide an FFI layer written for Rust utilizing CXX[1].

With GEDS Wasm Units integrating GEDS within the runtime, we obtain the following benefits:

- **Transparent storage tiering**. Legacy Wasm modules gain storage tiering capabilities without 1) rewriting large portions of the source code, 2) recompiling them, and 3) transparently, as Wasm modules will still interact with storage as if it was the original local storage. This allows Wasm applications to maintain hot data close in Tier Zero, and automatically offload infrequently-utilized data to the Persistency tier, optimizing the compute node's local storage and taking advantage of GEDS's in-memory storage for fast computations.
- **Efficient data communication**. With GEDS we allow Wasm modules to interact with each other, even between distributed nodes, efficiently. For local communications, Wasm modules now benefit from shared memory calls that were otherwise incompatible with Wasm, while remote communications now leverage RDMA. This benefit could be evaluated in the future, but is currently out of scope for this deliverable.
- **Increased memory**. The linear memory of Wasm legacy applications is limited to 4 GB due to Wasm's 32-bit memory addressing [2]. With GEDS, Wasm modules leverage its Tier Zero to fully utilize the compute node's memory, far beyond Wasm's 4 GB.

### 3.2.2 WASI hostcall interposition

When compiling a Wasm module that targets WASI, all the symbols that reference WASI hostcalls are left unresolved. This is meant for portability: including the hostcall implementation in the Wasm binary would tie that binary to a specific platform. The implementation of these hostcalls is supplied by the Wasm runtime in the form of a **WASI context**, a structure that contains hostcall state and implementation. When running a Wasm module, a second compilation and linking process takes place in the Wasm runtime, where the symbols that were purposely left unresolved are supplied by a WASI context. By altering this linking process, we **interpose custom implementations** of specific WASI hostcalls in the form of plugins.

Regarding GEDS, we leverage this hook to modify file I/O WASI hostcalls to replace the interaction with local storage with GEDS. A summary of the modified WASI hostcalls is available in Table 1. Briefly, we implement a POSIX-like file system at the Wasm runtime level with a custom table of GEDS file descriptors, which is stored in a custom GEDS WASI context. Inside this context, we also provide the functions that we utilize to interpose in all file and file-descriptor-related WASI hostcalls. These functions create, operate with, and destroy GEDS file descriptors. An example of these hostcall functions is `fd_write`, which roughly translates to the POSIX `write` syscall. Listing 1 contains the custom implementation of this hostcall to interact with either GEDS or the local filesystem, as required.

With hostcall interposition, we provide GEDS-based WebAssembly Units the capabilities to extend legacy Wasm modules with storage tiering, persistency (storage in Wasm modules is often ephemeral), and even communication through GEDS' publish-subscribe mechanism **transparently**. WASI hostcall interposition is transparent because it no longer requires recompiling Wasm applications to extend their I/O capabilities. Instead, we integrate custom logic at the Wasm runtime.

[1] https://cxx.rs

Table 1: List of WASI hostcalls interposed to implement GEDS-based WebAssembly Units

| WASI hostcall | POSIX equivalent | Description |
|---|---|---|
| `path_filestat_get` | `stat` | Get attributes of file/dir at path. |
| `path_open` | `open` | Open/create file. Creates file descriptor entry in WASI's file descriptor table. |
| `fd_filestat_get` | `fstat` | Get attributes of open file identified by a file descriptor. |
| `fd_pread` | `pread` | Read data with an offset from an open file identified by a file descriptor. |
| `fd_write` | `write` | Write data to an open file identified by a file descriptor. |
| `path_create_directory` | `mkdir` | Create directories. |
| `path_remove_directory` | `rmdir` | Remove empty directories. |
| `fd_datasync` | `fdatasync` | Synchronize file state with storage device. |
| `fd_close` | `close` | Close a file. Removes file descriptor entry from WASI's file descriptor table. |
| `path_unlink_file` | `unlink` | Delete a name and/or the file it points to at the specified path. |

Additionally, this integration is also **portable**, as it does not rely on an OS or architecture-specific trick. Finally, this approach respects WASI's **security** model and, even though it is still subject to the system's configured permissions, it allows for a certain degree of customization of the WASI hostcalls, without requiring the extra permissions needed by syscall interposition.

Listing 1: A code snippet of the custom GEDS context that 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();
        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
}
```

### 3.3 Evaluation

The evaluation of GEDS Wasm Units consists of two sets of experiments. Firstly, we carry out a series of performance micro-benchmarks to characterize the performance of GEDS Wasm Units (§3.3.1). Secondly, in §3.3.2, we compare the data persistency performance of GEDS-based WebAssembly Units with the alternatives presented in §3.1. Since an integration of GEDS *within* Wasm is not feasible, we abstain from evaluating this particular setup.

**Experimental setup**. The lightweight nature of GEDS Wasm Units allows for executing a high num-

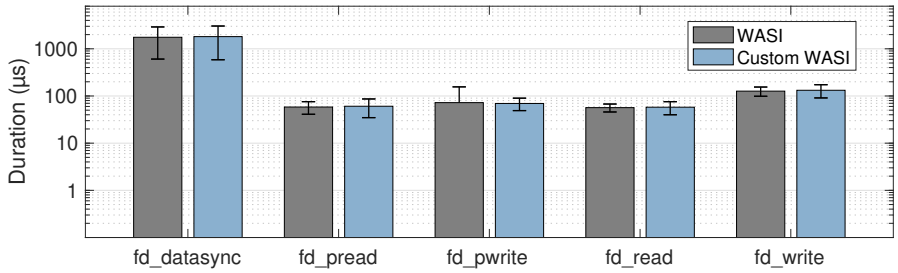


Figure 3: WASI hostcall duration with unmodified WASI against hostcall interposition with a custom NOOP WASI context.

ber of Wasm modules on a single node. For this reason, all experiments have been executed on a single compute node with 16 GB of RAM and 8 CPU cores, running Ubuntu Server 24.10. For GEDS' Tier Zero storage, compute nodes have been provisioned with solid-state storage drives, while the Persistency tier is supported by a MinIO instance located at KIO Networks' facilities. Wasm modules are packaged as OCI-compliant lightweight images and executed in Kubernetes as Wasm containers utilizing a runwasi containerd shim [19].

**Legacy Wasm applications**. As a legacy Wasm application for our experiments, we utilize an existing, pre-compiled Wasm module that performs the image pre-processing tasks of Metaspace's metabolomics pipeline. This application originally retrieves images from local storage (i.e., disk), rescales and normalizes them, and writes them back to local storage. With GEDS Wasm Units, we replace the write logic to integrate with GEDS and data encryption transparently.

**Competing systems**. For experiments where comparisons apply, we compare GEDS Wasm Units with two competing systems: `s3fs` and a custom GEDS integration through syscall interposition. The former is a FUSE implementation that mounts an S3 or S3-compatible (e.g., MinIO) bucket as a file system that can be exposed to Wasm modules transparently. The latter is a custom system call interposition implementation that leverages the `LD_PRELOAD` trick to replace the required system calls to interact with GEDS instead of local storage.

### 3.3.1 Performance microbenchmarks

**Hostcall interposition.** To implement GEDS Wasm Units, we interpose in the I/O path. For this reason, we must ensure that this interposition imposes no significant overhead on host calls, as I/O-intensive applications rely heavily on these functions. To do so, we run a microbenchmark in which we execute a Wasm module that performs all basic file I/O operations. We compare the latency of each I/O operation between an unmodified Wasm runtime and our custom runtime with a WASI context that bypasses the hostcall to the original implementation (NOOP). The results in Figure 3 show that hostcall interposition does not introduce a significant penalty, as the times for all hostcalls remain practically the same.

**Disk and network performance.** Using the Metaspace legacy Wasm application, we run a microbenchmark to characterize the I/O usage (disk and network) of GEDS Wasm Units under load. For this, we run multiple collocated units (modules) in the same node (ranging from 10 to 100) and measure their CPU usage, network upload throughput, and disk read/write throughput. Each module processes a fixed number of images, and thus, increasing the number of modules increases the total number of processed images. The number of GB processed for 10, 25, 50 and 100 units is 2.9, 7.2, 14.3, and 29.7, respectively. Figure 4 portrays the results of this experiment. The first insight is that modules first load the images, as indicated by the small disk read spike at the beginning of the job, and, as modules finish execution, the results are written to GEDS's Tier Zero (disk write spikes), and finally offloaded to S3 in the background (shown by the increase in network throughput). With the provisioned disk and network interface, Wasm Units achieve a peak network throughput of 220 MB/s and a disk write throughput of 205 MB/s for 100 modules. As we increase the number

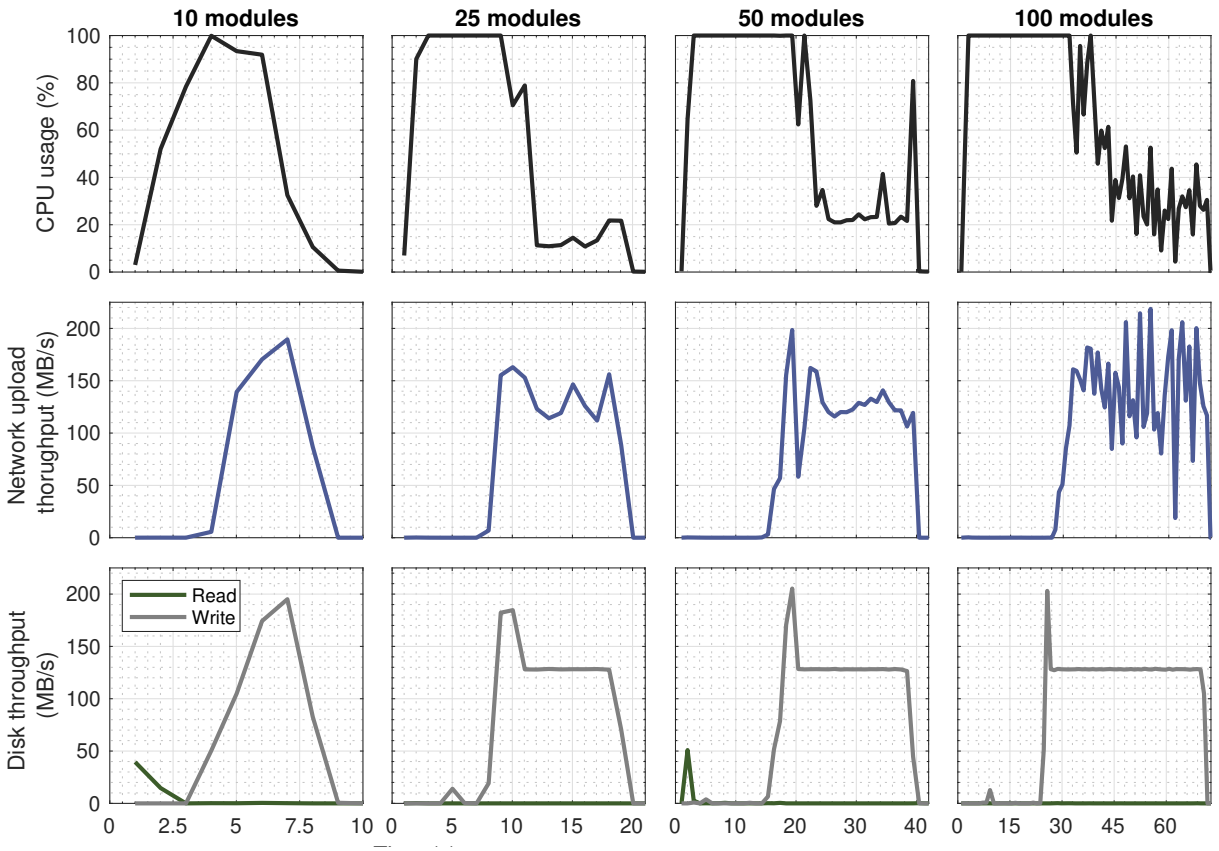


Figure 4: I/O performance microbenchmark of GEDS-based Wasm Units. Each column presents the result for a number of collocated Wasm modules (GEDS Wasm Units) in the same node. X axis shows time in seconds.

of Units, the time to process the images also increases, due to the Wasm modules competing for CPU and I/O resources.

### 3.3.2 Data migration in GEDS-based WebAssembly Units and external alternatives

To provide a comparison of GEDS Wasm Units with the competing alternatives, we carry out an experiment that compares GEDS' persistency tier performance with an S3-based FUSE (`s3fs`), and syscall interposition that replaces file I/O syscalls with S3 PUT and GET operations. To do so, we leverage the Metaspace legacy Wasm application with data encryption disabled, and run it on a single GEDS Wasm Unit for each of the three competing systems. The results are presented in Table 2, showing the number of images processed per second.

The S3 FUSE is the slowest of the three alternatives, with an average throughput of 7.60 images/s. This is due to the extra level of indirection that FUSE introduces (interaction with libfuse). By performing syscall interposition, we obtain a moderate speedup of 1.12× over FUSE, which falls short to GEDS Wasm Units, that achieve a throughput of 10.98 images/s and a speedup of 1.44× over FUSE. This is because even though syscall interposition eliminates the extra indirection of FUSE, it is still located further away from the module in comparison with GEDS Wasm Units. GEDS Wasm Units, on the other hand, are located in the same Wasm runtime process, before hostcalls are translated to syscalls, which gives a certain advantage over the presented alternatives.

The takeaway is that carrying out a deep integration of GEDS at the Wasm runtime process not only provides the benefits presented in §3.2.1, namely improved portability and security, over the explored alternatives, but also improves performance when migrating data to a persistent storage system. This result provides further evidence that the best-suited place for the GEDS-Wasm integra-

Table 2: Average persistent storage data offloading throughput (in images/s) of syscall interposition, `s3fs` and GEDS Wasm Units.

| **Setup** | $\overline{X}$ **(images/s)** | $\sigma$ | **Speedup** |
|---|---|---|---|
| `s3fs` (FUSE) | 7.60 | 0.68 | 1.00× |
| Syscall interposition (`LD_PRELOAD`) | 8.52 | 0.81 | 1.12× |
| GEDS-based WebAssembly Units | 10.98 | 0.42 | 1.44× |

tion is at the Wasm runtime.

### 3.4 Summary

We have introduced GEDS-based WebAssembly Units, a new compute abstraction for CloudSkin that combines Wasm's lightweight isolation, portability and high performance, with GEDS' data tiering and automated data migration to persistent storage. This deep integration, which takes place at the Wasm runtime process, aims to provide existing I/O intensive Wasm applications with transparent access to these features. GEDS Wasm Units are designed to be used alongside other compute abstractions of the project, such as C-Cells, and is especially focused at fine-grained, I/O-focused tasks such as image pre-processing. We have presented its architecture and provided a comparison with two viable alternatives *outside* of Wasm. Throughout the experimental evaluation, we have proved that an integration at the Wasm runtime improves the data migration performance up to 44% over its alternatives, while providing a series of operational benefits, such as improved portability and security. We have also characterized GEDS Wasm Units' disk and network utilization through a legacy Wasm application, and have determined that WASI hostcall interposition introduces no significant overhead, which is critical when interposing in the I/O path of I/O-intensive applications.

# 4 Nexus: Programmable Data Management across the Cloud-Edge Continuum

In the first half of the CloudSkin project (see D3.3), we have integrated Pravega with the Generic Ephemeral Data Store (GEDS) to address a pressing operational risk in the compute continuum: under network unreliability, the streaming storage tiering pipeline in Pravega could stall long-term offloading and eventually throttle ingestion. This was not acceptable in critical use cases like NCT, where Pravega is used as the streaming storage substrate to ingest and manage endoscopic video as data streams. By integrating an intermediate buffering layer at the Edge, GEDS masked Pravega disconnections from long-term storage and allowed the video ingestion path to remain functional until connectivity was restored. This practical enhancement demonstrated that a small, well-placed data service can materially improve end-to-end reliability in Edge/Cloud deployments (see Fig. 5).

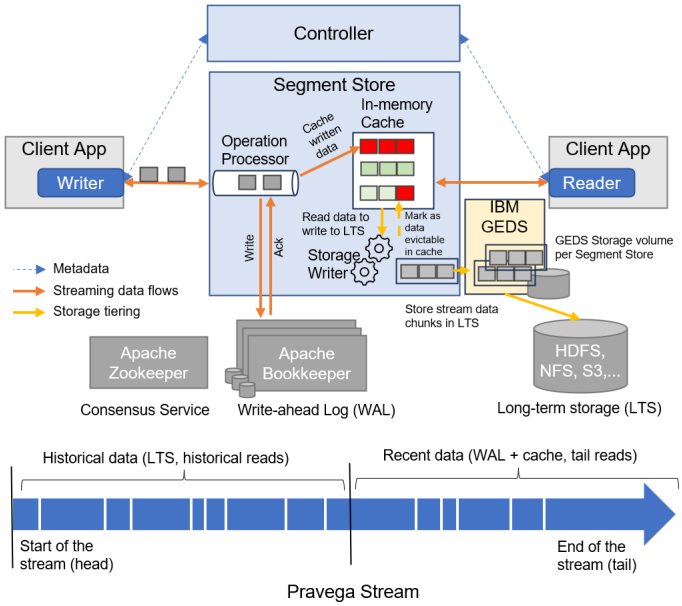


Figure 5: The Pravega and GEDS integration introduced in D3.3.

## 4.1 Motivation

However, as part of this integration work, we realized that the *streaming data tiering path from Edge to Core/Cloud remains under-exploited*. Today's tiering mechanisms in event streaming systems—including Pravega—move *cold* data chunks from the write-ahead log (WAL) to long-term storage via simple bindings executing write and read operations. In use cases like NCT, this is a missed opportunity: as endoscopic video streams transition from real-time ingestion to long-term retention, we could deploy custom in-transit functions for adding value to the stream tiering process across the Cloud-Edge. For example, such functions could (i) *buffer* data during outages to preserve continuous endoscopic video ingestion, (ii) *annotate* video chunks with semantic tags to accelerate clinical retrieval and research, (iii) *compress* or *encrypt* the data to reduce cost or enforce privacy, and (iv) *route* content to specialized Edge/Cloud stores, all these defined via policies. Because stream data tiering operates *offline at the chunk granularity* (1MB to hundreds of MBs), these operations can be performed without impacting video ingestion latency at the *event* granularity where surgeons require strict real-time guarantees (*e.g.*, $>$ 30FPS, sub-10ms write latency). In short, the stream tiering boundary is an ideal enforcement point for advanced, policy-driven data management across the Cloud–Edge continuum. This insight led us to an improved design, **Nexus**, a data management mesh that transparently intercepts tiering operations and executes programmable *streamlets* (*i.e.*, in-line data management functions) across the Cloud-Edge. This new system is envisioned to enable the computational storage functionality, as

[1]The content of this section maps to tasks T3.1 and T3.2 and is related to the paper "*Nexus: A Data Management Mesh for Tiered Data Streams*", submitted for publication.

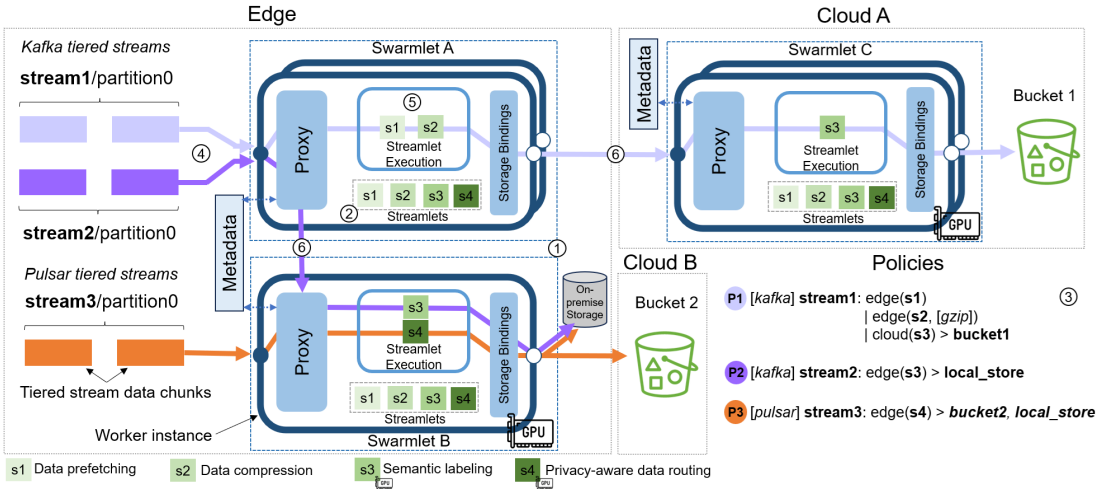


Figure 6: Architecture overview and operation of Nexus.

envisioned in T3.2.

### 4.2 Nexus in a Nutshell

Nexus bridges the gap between event streaming systems and external storage by enabling *transparent, extensible, and location-aware* data management on *tiered stream data chunks*, without impacting the hot streaming ingestion path. For NCT, the goal is to keep real-time AI inference stable at the edge, while enriching and operating on data as it moves to the Cloud Core.

**Design principles** Nexus is built on three main principles: (i) **decoupling** stream processing from data management—stream events at millisecond timescales vs. tiered chunks at second/minute scales; (ii) **transparency** via object-storage API interception (*e.g.*, S3-compatible), so any streaming system (such as Kafka, Pulsar, Pravega) can be supported without changes; and (iii) **extensibility and heterogeneity**, allowing user-defined functions (*streamlets*) to run at specific *locations* (Edge, Cloud) and on required *hardware* (GPU, TEE), orchestrated by *policies*. Returning to the NCT use case, this means placing buffering functions at the edge close to operating rooms, and scheduling AI-powered annotation streamlets on GPU nodes in the hospital IT center.

**Abstractions** Nexus builds on three abstractions that enable programmable data management across the Cloud–Edge continuum: **streamlets**, **swarmlets**, and **policies**. Streamlets are lightweight, user-defined functions executed inline on tiered stream data chunks during storage operations, allowing transformations such as compression or encryption, performance optimizations like caching, semantic enrichment through AI-based annotation, or routing decisions for privacy and compliance. Unlike event-level processing, streamlets operate at chunk granularity (megabytes), making them ideal for tasks that do not belong in the hot ingestion path. Swarmlets group Nexus worker instances into clusters associated with a specific infrastructure domain (Edge or Cloud) and hardware profile (e.g., GPU for AI inference, TEE for confidential execution), exposing a standard object-storage API endpoint so streaming systems can tier data transparently. Finally, policies orchestrate streamlet pipelines by defining which functions to apply, in what order, and where they should run, including hardware constraints and output destinations.

**Architecture** Nexus operates as a middleware layer that transparently intercepts tiered storage operations from event streaming systems and applies policy-driven streamlet pipelines across Edge and Cloud (see Fig. 6). Its architecture consists of containerized worker instances grouped into swarmlets, each associated with a specific location (*e.g.*, operating room edge cluster or hospital IT center) and hardware profile (GPU for AI inference, TEE for confidential execution). Swarmlets expose standard object-storage APIs (such as S3-compatible endpoints), allowing streaming systems like Pravega to

offload chunks transparently. When a storage tiering request arrives, Nexus intercepts it and executes the configured streamlet pipeline inline – transforming, annotating, or routing the data as required – before acknowledging completion to preserve durability guarantees. Policies define the composition and placement of streamlets, enabling multi-stage workflows such as buffering at the edge, semantic annotation on GPU nodes in the cloud, and final storage in a compliance-approved bucket. To enforce these policies efficiently, Nexus implements mesh-like data routing: requests can hop across swarmlets based on location and hardware constraints, while partition-aware routing ensures that stateful streamlets (e.g., annotation maintaining per-partition indexes) process data consistently without global locks. Metadata for swarmlets, streamlets, and policies is maintained locally with lightweight synchronization for cross-location updates, and integrity is preserved through checksum recalculation for transformer streamlets.

**Streamlet state.** While many streamlets are stateless, advanced data management often requires maintaining persistent state across executions – for example, tracking which chunks have been annotated or routed to specific storage tiers. Nexus supports this through lightweight state primitives and partition-aware routing. Each streamlet can store key–value pairs as *object tags* or mark data structures as `@persistent`, which Nexus automatically saves in the metadata backend. To avoid costly global synchronization, Nexus exploits the sequential and partitioned nature of streams: all chunks from the same partition are routed to the same streamlet instance, ensuring that updates occur in order and eliminating contention on shared metadata. This design reduces metadata read amplification, while guaranteeing that stateful operations such as semantic annotation or privacy-aware routing execute consistently.

**Mesh-like data routing.** To enforce pipelines across heterogeneous infrastructures, Nexus implements mesh-like data routing that transparently forwards tiering requests between swarmlets while preserving consistency and performance. When a streaming system offloads a chunk to Nexus, the request first reaches the swarmlet exposed as the tiering endpoint; from there, Nexus may route it to other swarmlets based on policy requirements such as location (Edge vs. Cloud) and hardware constraints (GPU for AI annotation, TEE for confidential execution). Routing decisions also consider partition affinity: all chunks from the same stream partition are deterministically mapped to the same worker instance using consistent hashing, ensuring that stateful streamlets maintain a coherent view of their local state without global locks. This approach minimizes metadata contention and reduces read amplification. Despite involving multiple hops, Nexus preserves transparency for the streaming system and guarantees end-to-end durability by acknowledging requests only after the entire pipeline completes successfully.

### 4.3 Nexus in Action

This section presents Nexus from a developer's perspective and demonstrates how its programmability is applied to the NCT surgery use case. Developers extend Nexus by implementing streamlets that process stream data chunks. Nexus supports two streamlet types: `ByteStreamlet` and `EventStreamlet` (see Table 3).

A `ByteStreamlet` allows developers to process raw byte streams during `PUT` and `GET` operations by implementing the methods `processPutBytes` and `processGetBytes`. These streamlets are suitable for functionality such as compression or encryption. Nexus invokes these methods in-line during storage operations, ensuring that transformations are applied before data is durably stored.

For streamlets like caching or buffering, developers can also implement the `DataSourceStreamlet` interface. This interface extends the byte-based model by allowing streamlets to intercept the data source before a `GET` operation via the `handlePreGet` method. This enables streamlets to pre-load content from alternative sources or serve cached data directly.

The `EventStreamlet<T>` API allows developers to process deserialized records from stream data chunks. These streamlets are ideal for semantic processing tasks, such as AI-based inference or content-based annotation. Developers must provide a `Deserializer<T>` implementation to convert the input stream into typed records. The streamlet then defines `processPutRecord` and `processGetRecord`

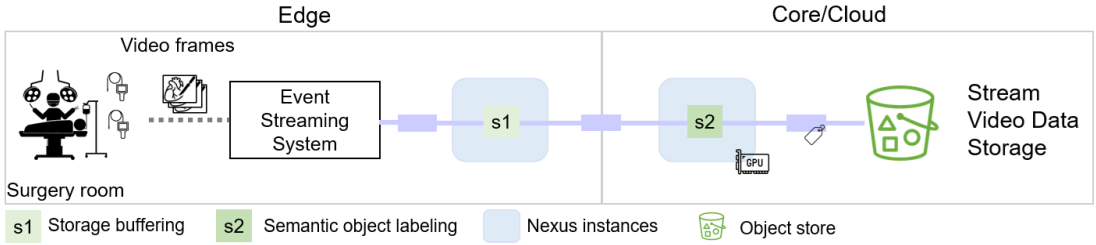


Figure 7: Surgiomics challenges addressed with Nexus.

to handle each record individually. Importantly, event streamlets cannot modify the request content to preserve data integrity.

Both ByteStreamlet and EventStreamlet can maintain persistent state across executions by annotating data structures with a @Persistent tag. Nexus automatically stores and retrieves these structures using the system metadata backend at hand.

Table 3: 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) |

We created a Nexus pipeline for NCT as visible in Fig. 7. The goal is twofold: i) create a data buffering streamlet to provide similar fault-tolerance capabilities to Pravega as the integration with GEDS, ii) demonstrate AI-powered streamlets by annotating data objects based on content using NCT models. Next, we describe the development of these two streamlets:

**Buffering under storage unavailability**. Computer-assisted surgery is an increasingly important domain for improving the success of surgical procedures. To store multimedia in a streaming fashion and perform real-time analytics, NCT is using Pravega streams to ingest video and image data. Unfortunately, Pravega exhibits limitations upon unavailability of long-term storage. It can only continue ingesting events until its in-memory cache is full of data waiting to be offloaded [20]. During a surgery, this may imply that video ingestion and data analytics are impacted in case of minor transient network outages, which is not acceptable. To address external storage unavailability, we implemented a byte streamlet that durably buffers incoming data chunks. This allows Pravega to continue ingesting data uninterrupted. The streamlet implements the DataSourceStreamlet API, serving data from local storage or redirecting GET requests to the object store as needed.

**Semantic stream data annotations**. To help NCT scientists locate relevant fragments of video/image streams, some type of semantic indexing is needed. Hence, we developed an EventStreamlet that integrates NCT AI models and leverages a custom Deserializer to extract JPEG images from Pravega stream chunks. Upon processing all or a subset of images within a chunk, the streamlet annotates the corresponding data object with semantic tags (*e.g.*, surgical phase, instruments). This metadata facilitates efficient retrieval of stream fragments containing relevant visual content.

The use-case evaluation of these streamlets can be seen in D5.4. In what follows, we provide a general evaluation of Nexus via experiments and benchmarks.

### 4.4 Validation

The evaluation of `Nexus` focuses on the following questions:

- Can `Nexus` provide high performance storage request interception and routing (§4.4.2)?
- What benefits may `Nexus` transparently bring to existing event streaming systems (§4.4.3)?

#### 4.4.1 Setup

**Implementation** We have implemented `Nexus` as a middleware for the S3Proxy project [21], an extensible S3-compatible proxy. The middleware intercepts S3 requests and processes them based on system metadata (*e.g.*, swarmlets, streamlets, and policies). From the client's perspective, `Nexus` behaves like a standard S3 interface: the client receives a response only if the request is successfully stored or fails. Internally, however, `Nexus` uses asynchronous programming to concurrently process requests in a streaming fashion. The implementation spans over 7K lines of Java code, including a CLI for managing metadata and scripts for deployment and benchmarking. Redis [22] serves as the metadata backend, with local caching for read-only access. Upon updates (*e.g.*, new policies), Redis notifies the `Nexus` metadata service to apply changes. `Nexus` also supports dynamic compilation and runtime loading of streamlets and deserializers. The code is available at [23].

**Cluster settings**. We deploy `Nexus` in two Kubernetes-based environments. On AWS, we create an EKS cluster on top of 3 `i3en.2xlarge` instances, each with 8vCPUs, 64GB of memory, and 2 local NVMe drives to deploy all instances (`Nexus` workers, Redis, and benchmarks). We use AWS S3 for long-term storage. Our lab cluster is composed of 7 VMs running Kubernetes. Each VM has 8 CPUs, 20GB of memory, and 80GB of storage (1 master, 6 workers). One VM has a GPU available for running AI-related streamlets (Nvidia A16 with 16GB VRAM). In our experiments, Edge and Cloud are logical regions for swarmlets within the same infrastructure.

**Baselines**. We exercise `Nexus` via 3 streaming systems:

*Apache Kafka*: Apache Kafka [24, 25] is a distributed event streaming platform optimized for high-throughput, low-latency data processing via a publish-subscribe model. It structures data into topics, partitions, and offsets, ensuring scalability and fault tolerance through replication. For tiered storage, introduced in KPI-405 [26], we use Aiven's Kafka storage tiering plugin [27].

*Apache Pulsar*: Apache Pulsar [28] is a scalable, low-latency event streaming system. Its tiered storage offloads sealed log segments from high-performance storage (*e.g.*, BookKeeper [29, 30]) to cost-efficient long-term storage, reducing costs while preserving seamless access to historical data.

*CNCF Pravega*: Pravega [31, 20] is a distributed stream storage system for unbounded, high-throughput data. Pravega streams support dynamic scaling, exactly-once semantics, and long-term retention. Its tiered storage integrates low-latency short-term storage via Apache BookKeeper with cost-efficient long-term storage like cloud object stores.

**Benchmarks**. To generate workloads in our experiments, we resort to the following benchmarks:

*FIO*: FIO [32] is a powerful benchmarking tool used to evaluate the performance of storage systems, including object storage services. It supports various workload types such as sequential and random read/write operations, mixed I/O patterns, and custom workloads. FIO allows users to configure parameters like block size, I/O depth, and number of jobs, and it supports multiple I/O engines (*e.g.*, libaio).

*OpenMessaging Benchmark*: OpenMessaging Benchmark [33] is a comprehensive toolset designed for benchmarking messaging systems in the cloud. It supports systems like Apache Kafka, Apache Pulsar, Pravega, and more.

**Datasets**. We used the following datasets for building stream data payloads in our experiments[2]: i) *Data compression*: For data compression experiments, we used traces from HDFS from a collection of system logs available at [34, 35]. ii) *Semantic data routing*: For image payloads, we used images from ImageNet [36] and Cholec80 [37]. iii) *Genomic data*: For the genomics data management experiment, we use FASTQ files available from SRA Toolkit [38].

[2]We extended OpenMessaging Benchmark to load custom payloads.

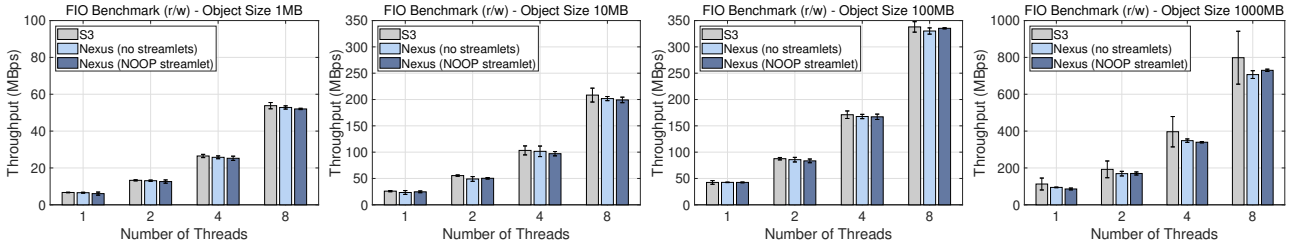


Figure 8: Interception performance of 1 `Nexus` instance under FIO read/write workloads on AWS.

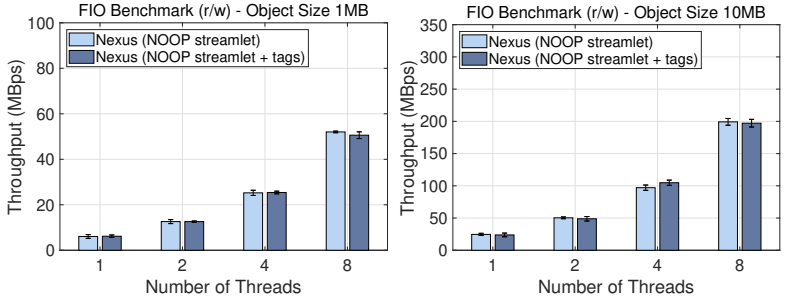


Figure 9: Performance impact of leveraging object tags using 1 `Nexus` instance under FIO read/write workloads on AWS.

### 4.4.2 Interception Performance

First, we evaluate `Nexus` in isolation via FIO to understand its interception and routing performance on AWS.

**IO interception performance**. We execute a read/write workload of objects against S3 via FIO as a performance baseline. Then, we reproduce the same workload when a single `Nexus` worker intercepts storage request without processing and when it executes a no-op streamlet. Fig. 8 shows the results for various levels of benchmark parallelism and object sizes.

First, the main observation in Fig. 8 is that `Nexus` does not induce a significant performance penalty when intercepting object storage requests. In most cases, the throughput reduction that FIO reports when `Nexus` intercepts requests falls below 5%. `Nexus` worst performance cases compared to the baseline seem to be for one benchmark thread and extreme object sizes (1MB and 1000MB). A possible explanation may be that `Nexus` performs metadata checks during the hot path that incur additional latency, whose impact is more pronounced for low parallelism rates. We also observe that throughput variability in `Nexus` is generally lower compared to FIO performing IO directly to S3. This may be due to the server buffering configurations for optimizing HTTP connections in `Nexus`. Overall, a single `Nexus` instance can execute a no-op streamlet from multiple client requests at $\approx$ 730MBps, demonstrating the feasibility of the current implementation.

**Object metadata**. `Nexus` exploits object metadata to allow streamlets annotating objects, as well as to store specific system information (*e.g.*, if the object was processed by a transformer streamlet). To inspect the impact of using object tags, Fig. 9 shows the throughput comparison of a no-op streamlet versus a no-op streamlet that stores and retrieves an object tag for PUTs and GETs, respectively.

Fig. 9 shows that the throughput reduction related to managing object tags is generally minor. Specifically, the worst-case throughput reduction observed is 2.7% and 3.2% for 1MB and 10MB objects, respectively. A key reason for minimizing the performance impact of managing object tags is that these are asynchronous operations. For instance, upon an object PUT, object tags are stored asynchronously once the actual object has been correctly stored, thus not blocking the original request. This experiment demonstrates that `Nexus` can effectively manage object metadata, which can be leveraged for both system and user purposes.

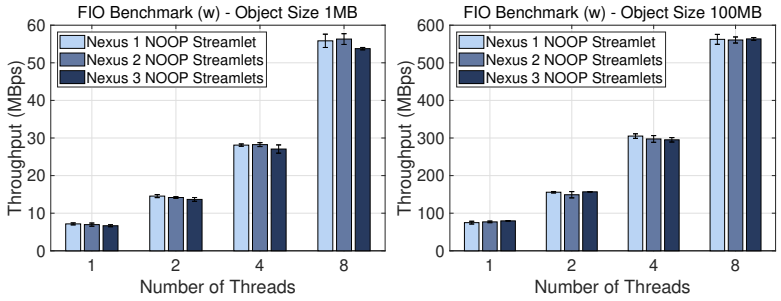


Figure 10: Processing performance depending on pipeline size of 1 `Nexus` instance under FIO write workloads on AWS.

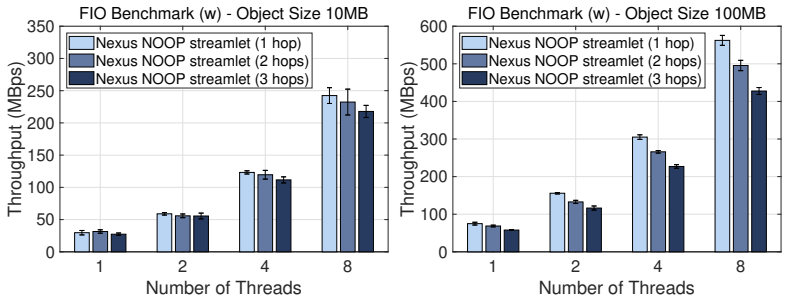


Figure 11: Performance impact of data routing hops under FIO write workloads on AWS.

**Streamlet pipelining overhead**. Next, we evaluate the performance of `Nexus` intercepting object storage requests with a different number of pipelined functions in the same worker instance. To this end, we create policies pipelining up to 3 no-op streamlets for `PUT` requests (see Fig. 10).

Fig. 10 shows that `Nexus` can successfully pipeline multiple functions as a chain. Visibly, each additional no-op streamlet can induce an additional 1% to 6% of throughput reduction, depending on the case. In line with prior observations, the 3-streamlet case with the lowest levels of parallelism exhibits the most pronounced performance drop compared to the baseline (up to 6.8%). Overall, this experiment demonstrates the practicality of building complex streamlet pipelines in `Nexus`.

**Data routing overhead**. `Nexus` can route object requests transparently to event streaming systems based on the policies and streamlets configured. We evaluate the performance impact of executing no-op streamlets across 1 to 3 `Nexus` worker instances (see Fig. 11).

In the more typical 2-hop pipeline scenario (*e.g.*, Cloud-Edge deployments), Fig. 11 shows that the throughput reductions due to routing across swarmlets ranges from 6.2% to 14.8%, depending on object size and benchmark parallelism. We also assess a more demanding 3-hop configuration in `Nexus`. In this case, we observe a higher performance penalty, with throughput reductions reaching up to 25.6%, particularly for larger objects. Despite the added routing complexity, these results confirm that `Nexus` maintains acceptable performance levels and preserves transparency from the event streaming system's perspective, which is a key design goal.

### 4.4.3 Enhancing Event Streaming Systems

Next, we focus on illustrating the advantages that `Nexus` provides to event streaming systems in our on-premises cluster.

**Infrastructure abstraction**. We perform two experiments in this section: *data compression* and *semantic data routing* (see Fig. 12). For the first experiment, we deploy two `Nexus` swarmlets logically associated to Edge and Cloud regions, where data compression takes place at the Edge. For the second experiment, simulating a more realistic use-case, the Edge cluster is expanded with GPU-enabled nodes. Then, we create a new swarmlet in the Edge region and implement a semantic data rout-

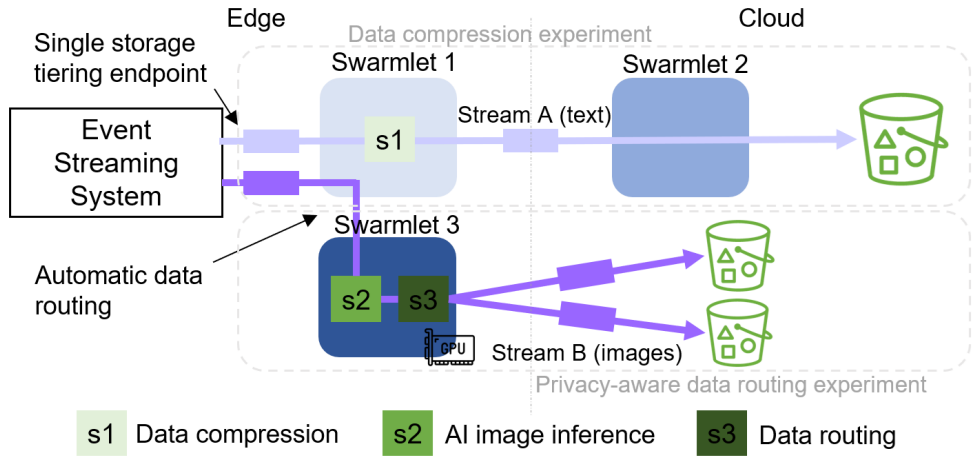


Figure 12: Experiment setup showing `Nexus` capabilities to abstract infrastructure from event streaming systems.

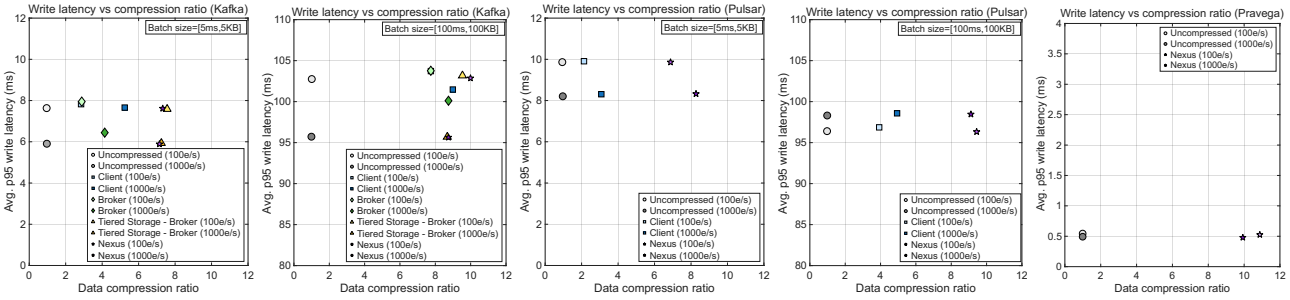


Figure 13: Data compression vs write latency of event streaming systems data compression and `Nexus`.

ing streamlet pipeline that requires GPU acceleration. The crucial insight is that such infrastructure expansion is transparent to the event streaming system, as `Nexus` re-routes storage requests across swarmlets based on hardware requirements. Therefore, `Nexus` allows administrators to manage the infrastructure and policies without requiring changing the (limited) tiered storage configuration of event streaming systems.

**Transparent data compression**. First, we focus on data compression. We implemented a transformer streamlet: GZip data compression (`s1`). We set event streaming systems to offload data to `Nexus swarmlet 1` in our on-premises cluster and configured `Nexus` against a MinIO bucket. We compare the impact of data compression on the event streaming systems built-in mechanisms (if any) in terms of latency and compression ratio for Kafka, Pulsar, and Pravega.

Fig. 13 shows how different compression models —client-side, broker-side, tiered storage (broker), and `Nexus`— affect compression ratio and write latency across event streaming systems. `Nexus` achieves the highest compression ratios (*e.g.*, up to 10.4× in Pravega, 9.9× in Kafka, and 8.3× in Pulsar), while maintaining low write latencies (similar to compressing data at the tiered storage module in the Kafka broker). `Nexus` achieves compression ratios of up to 3.9× higher than the best client- or broker-side compression in Kafka, and over 3.8× higher in Pulsar (Pravega has no compression implemented).

Interestingly, in Kafka and Pulsar, which use user-defined batching strategies (*e.g.*, fixed batch time and size), we observe that larger batch sizes and higher producer rates generally improve compression ratios by leading to larger event batches. For instance, in Kafka, increasing the batch size from 5ms/5kb to 100ms/100kb at 100e/s improves the compression ratio from 2.84× to 7.75× with client-side compression. However, this improvement comes at the cost of increasing the 95th per-

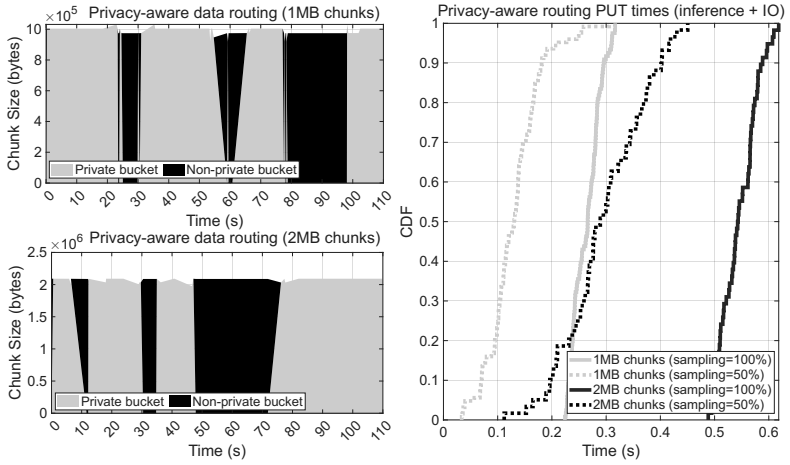


Figure 14: Storage bucket for stream data chunks decided by the semantic streamlet (left) and inference times based on chunk size and event sampling rate (right).

centile write latency by $\approx 15\times$. These results show that `Nexus` enables transparent data compression across diverse event streaming systems, without impacting the hot path write latency.

**Semantic data routing**. We instantiated a new swarmlet (`swarmlet 3`) in our on-premises infrastructure deployed on nodes equipped with GPUs (see Fig. 12). We generate a workload using the OpenMessaging Benchmark in which the benchmark writes images to Kafka containing humans or not, switching randomly every 2MBs. Then, we configured a two-streamlet pipeline at the Edge: i) a semantic human detection streamlet via the Yolov5 [39] model (`s2`), and ii) a data routing streamlet (`s3`) that stores a data chunk on a private bucket if any of the processed images include a human. Note that `s2` is a stateful streamlet, meaning that it keeps a `@Persistent` map of the data chunks stored in the private bucket.

Fig. 14 (left) shows a time-series with the semantic decisions made by the streamlet pipeline. Visibly, `Nexus` stores stream data chunks to the right bucket based on the result of the AI inference process (when Kafka manages 1MB and 2MB chunks). Furthermore, while `s2` is an `EventStreamlet` that processes the internal content of a chunk event by event, its implementation allows the user to perform AI inference on a subset of a chunk's events. Fig. 14 (right) illustrates `s2`'s processing time depending on the image sampling rate and the chunk size. Finally, we tested `s2` by using the proposed partitioned approach vs a shared approach in which any streamlet instance can manage the streamlet state. In this sense, while the number of metadata writes is similar as any update needs to be persisted in metadata, we found that the partition-aware data routing reduced in $\approx 98\%$ read requests to the metadata in our experiment. The reason is that the same streamlet instance was in charge of managing metadata for a given stream partition thanks to `Nexus` partition-aware data routing, thus avoiding having to load the most recent metadata on every request.

**Summary** The evolution from the initial GEDS–Pravega integration to Nexus reflects a natural progression from tactical reliability improvements to strategic programmability across the Cloud–Edge continuum. While GEDS addressed a critical operational gap by masking storage outages in NCT's real-time ingestion path, Nexus generalizes this principle into a policy-driven, extensible framework that transforms tiered storage into an enforcement point for advanced data management. By decoupling hot-path streaming from chunk-level operations, Nexus enables buffering, compression, semantic annotation, and privacy-aware routing without compromising real-time guarantees. Such capabilities are essential for NCT's vision of AI-assisted surgery and beyond. Ultimately, Nexus positions CloudSkin to deliver not only resilience but also intelligence and adaptability at scale, bridging event streaming systems with heterogeneous infrastructures in a transparent and future-proof manner.

## 5 Conclusions

This report presents an updated and consolidated overview of the CloudSkin dataplane platform activities, focusing on the storage systems developed. The document first summarizes the existing GEDS progress, as described in the previous deliverables. It then introduces two new complementary contributions, GEDS-based WebAssembly Units and Nexus, that together address key limitations in current cloud–edge data processing systems.

GEDS-based WebAssembly Units extend WebAssembly beyond the constraints of today's WASI by integrating GEDS directly at the runtime level, enabling transparent access to tiered, persistent, and high-performance storage without rewriting or recompiling existing Wasm applications. This dual-runtime approach complements the C-Cells runtime, allowing the same portable Wasm binaries to support both distributed computation and fine-grained, I/O-centric data transformations executed close to storage.

Building on this foundation, Nexus elevates storage tiering into a programmable control point for data management across the Cloud–Edge continuum. By transparently intercepting tiered storage operations and executing policy-driven streamlets on chunk-level data, Nexus enables buffering, compression, semantic annotation, and privacy-aware routing without impacting the real-time ingestion path of event streaming systems. Its abstractions – streamlets, swarmlets, and policies – allow data management logic to be flexibly placed across heterogeneous infrastructures and hardware, while remaining transparent to cloud streaming systems such as Kafka, Pulsar, and Pravega.

## References

[1] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler, "Unification of temporary storage in the NodeKernel architecture.," in 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019.

[2] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," June 2017.

[3] 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.

[4] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 140–149, IEEE, 2022.

[5] Cloudfare, "Cloudfare workers," mar 2018.

[6] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: type-driven secure cryptography for the web ecosystem," Proc. ACM Program. Lang., vol. 3, Jan. 2019.

[7] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in webassembly," in 2019 IEEE Symposium on Security and Privacy (SP), pp. 1256–1274, 2019.

[8] C. Segarra, S. Shillaker, G. Li, E. Mappoura, R. Bruno, L. Vilanova, and P. Pietzuch, "GRANNY: Granular management of Compute-Intensive applications in the cloud," in 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25), (Philadelphia, PA), pp. 205–218, USENIX Association, Apr. 2025.

[9] M. Chadha, N. Krueger, J. John, A. Jindal, M. Gerndt, and S. Benedict, "Exploring the use of webassembly in hpc," in Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP '23, (New York, NY, USA), p. 92–106, Association for Computing Machinery, 2023.

[10] "Webassembly system interface (wasi)." `https://wasi.dev`, 2019.

[11] N. Fitzgerald, "Making webassembly and wasmtime more portable," sep 2024.

[12] S. W. Poole, O. Hernandez, J. A. Kuehn, G. M. Shipman, A. Curtis, and K. Feind, OpenSHMEM - Toward a Unified RMA Model, pp. 1379–1391. Boston, MA: Springer US, 2011.

[13] G. Linux, "mmap linux manual page."

[14] J. Napieralla, "Considering webassembly containers for edge computing on hardware-constrained iot devices," 2020.

[15] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski, "Webassembly modules as lightweight containers for liquid iot applications," in International Conference on Web Engineering, pp. 328–336, Springer, 2021.

[16] A. Jacobs, M. Gülmez, A. Andries, S. Volckaert, and A. Voulimeneas, "System call interposition without compromise," in 2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 183–194, 2024.

[17] "Preload trick." `https://www.admin-magazine.com/HPC/Articles/Preload-Trick`, 2025.

[18] B. Alliance, "Wasmtime," mar 2022.

[19] containerd, "runwasi - facilitates running wasm / wasi workloads managed by containerd," nov 2022.

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

[21] "S3proxy." https://github.com/gaul/s3proxy, 2025.

[22] "Redis." https://redis.io, 2025.

[23] "Nexus repository." https://github.com/cloudskin-eu/nexus-nct-streamlets, 2025.

[24] J. Kreps, N. Narkhede, J. Rao, et al., "Kafka: A distributed messaging system for log processing," in NetDB '11, vol. 11, pp. 1–7, 2011.

[25] "Apache kafka." https://kafka.apache.org, 2024.

[26] "Kip-405: Kafka tiered storage." https://cwiki.apache.org/confluence/display/KAFKA/KIP-405%3A+Kafka+Tiered+Storage, 2025.

[27] "Remotestoragemanager for apache kafka tiered storage." https://github.com/Aiven-Open/tiered-storage-for-apache-kafka, 2025.

[28] "Apache pulsar." https://pulsar.apache.org, 2024.

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

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

[31] "Pravega." https://cncf.pravega.io, 2023.

[32] "fio - flexible i/o tester." https://fio.readthedocs.io, 2025.

[33] "Openmessaging benchmark." https://github.com/openmessaging/benchmark, 2025.

[34] J. Zhu, S. He, P. He, J. Liu, and M. R. Lyu, "Loghub: A large collection of system log datasets for ai-driven log analytics," in IEEE ISSRE'23, pp. 355–366, 2023.

[35] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in ACM SOSP'09, p. 117–132, 2009.

[36] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in IEEE CVPR'09, pp. 248–255, 2009.

[37] D. M. J. M. M. D. M. N. P. Andru Twinanda, Sherif Shehata, "Endonet: A deep architecture for recognition tasks on laparoscopic videos," IEEE Transactions on Medical Imaging, vol. 36, 02 2016.

[38] "Sra toolkit." https://hpc.nih.gov/apps/sratoolkit.html, 2025.

[39] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, Y. Kwon, K. Michael, J. Fang, Z. Yifu, C. Wong, D. Montes, et al., "ultralytics/yolov5: v7.0-yolov5 sota realtime instance segmentation," Zenodo, 2022.