CloudSkin

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

CloudSkin

(grant agreement No 101092646)

Adaptive virtualization for AI-enabled Cloud-edge Continuum

D4.3 Reference implementation of Cloud-edge platform

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	30
WP/Task related to this document	WP4
WP/Task responsible	IMP
Leader	Peter Pietzuch (IMP)
Technical Manager	Carlos Segarra (IMP)
Quality Manager	Marc Sanchez-Artigas (URV)
Author(s)	Carlos Segarra (IMP), Huanzhou Zhu (IMP), Guo Li (IMP), Peter Pietzuch (IMP), Alan Cueva (DELL), Ardhi Putra Pratama Hartono (TUD)
Partner(s) Contributing	IMP, TUD, DELL, KIO
Document ID	CloudSkin_D4.3_Public.pdf
Abstract	Final report that details design of mechanisms and evaluation results of WP4, CloudSkin's execution layer. This report also includes the release of produced frameworks and technologies through open source publication platforms, their integration with the different use-cases, and their deployment on KIO network's cloud-edge testbed.
Keywords	Execution layer, WebAssembly, C-Cells, GRANNY, Scone, Pravega

History of changes

Version	Date	Author	Summary of changes
0.1	14-11-2025	Carlos Segarra	Skeleton of the document and placeholders.
0.2	30-11-2025	Carlos Segarra, Huanzhou Zhu, Guo Li, Peter Pietzuch	Fill in text for main body sections.
0.3	03-12-2025	Ardhi Putra Pratama Hartono	Confidential computing contents.
0.4	15-12-2025	Carlos Segarra, Guo Li, Peter Pietzuch	GRANNY reference implementation, architecture, and open-source release.
0.5	22-12-2025	Carlos Segarra, Huanzhou Zhu, Guo Li, Peter Pietzuch	Address reviewers' comments.
1.0	29-12-2025	Carlos Segarra, Huanzhou Zhu, Guo Li, Peter Pietzuch	Final version.

Table of Contents

List of Abbreviations and Acronyms

AI	Artificial Intelligence
AOT	Ahead-of-Time
API	Application Programmable Interface
C-Cell	Cloud-edge Cell
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
EPC	Enclave Page Cache
GB	Giga Byte
GPU	Graphic Processing Unit
HPC	High Performance Computing
IoT	Internet of Things
JIT	Just-in-Time
KPI	Key Performance Indicator
MEE	Memory Encryption Engine
MPI	Message Passing Interface
OMP	Open MP
OS	Operating System
PRM	Processor Reserved memory
SDK	Software Development Kit
SGX	Software Guard eXtensions
SSD	Solid State Drive
TCB	Trusted Computing Base
TDX	Trust Domain eXtensions
TEE	Trusted Execution Environment
TPU	Tensor Processing Unit
VM	Virtual Machine
WASI	WebAssembly System Interface
WASM	WebAssembly
WP	Work Package

1 Executive summary

Deliverable D4.3, *Reference Implementation of the Cloud-Edge Platform*, presents the final outcome of CLOUDSKIN's WP4, *Adaptive Virtualization for the Cloud-Edge Continuum*. The goal of WP4 is to provide the **execution layer to enable virtualization in the cloud-edge continuum**. This deliverable builds on D4.1, where we introduced C-Cells , CLOUDSKIN's execution abstraction, and D4.2, where we introduced GRANNY, the first system implementation of a runtime using C-Cells. In this deliverable, we present a unification of the previous prototypes into a single execution layer, and full open-source reference implementations for CLOUDSKIN, and Sconeified Pravega, a combination of secure C-Cells with the streaming storage platform from WP3. We also present integrations of WP4 with different use-cases, and an extensive evaluation on the KIO testbed that confirms the fulfillment of all KPIs related to C-Cells.

The key contributions of this deliverable are:

- **Final C-Cell abstraction.** The CLOUDSKIN execution abstraction integrates low-overhead WebAssembly isolation, with fine-grained snapshotting and live migration as presented in GRANNY.
- **Confidential C-Cells.** Secure execution of C-Cells inside trusted execution environments (TEEs) with the integration between SCONE and Pravega for secure storage and processing of data.
- **Two reference implementations.** The GRANNY runtime, for scientific applications, and Sconified Pravega, for privacy-preserving data streaming.
- **Two use-case integrations** based on the metabolomics and computer-assisted surgery use-cases.
- **Deployment and validation** We validate the reference implementations as well as the use-case integration in the KIO testbed.
- **Open-source releases** We describe the open-source releases and maintenance of the reference implementations.

Together, these contributions present a unified view of CLOUDSKIN' execution platform and its integration with the storage layer (WP3), learning plane (WP5), and its use-cases.

2 Introduction

2.1 CLOUDSKIN WP4 Objectives and Scope

WP4 is responsible for the execution layer of the cloud-edge continuum. The goals and objectives of WP4 can be summarized in its four main tasks:

T4.1. C-Cell Abstraction: In D4.1, we presented C-Cells, a unified execution abstraction for threads and processes based on the lightweight WebAssembly sandbox. The use of WebAssembly as an intermediate representation and sandbox solution enables the transparent support of legacy applications written in a variety of programming languages, including complex stacks like MPI or OpenMP, without sacrificing performance. WebAssembly also makes C-Cells portable, enabling their transparent execution in heterogeneous hardware.

T4.2. Execution and Virtualization for C-Cells: In D4.2, we instantiated the first execution runtime for C-Cells, GRANNY. GRANNY is a distributed runtime that supports the execution of C-Cells running unmodified OpenMP and MPI applications. GRANNY manages the distributed message passing and shared memory accesses through the use of control points a lightweight virtualization technique to trap application execution with very limited runtime overhead. In D4.3, we provide further details on GRANNY's design, implementation, and open-source release and maintenance.

T4.3. Orchestration and Migration for C-Cells: In D4.2, we also described the design and implementation of different resource orchestration policies for CLOUDSKIN based on performance, locality, or cost-effectiveness. Using the control-points from T4.2, we also implement lightweight C-Cell migration, a key requirement to materialize the resource orchestration policies.

T4.4. Confidential C-Cells: In D4.3, we present the mature integration of C-Cells with trusted execution environments (TEEs), in order to enable the integration of CLOUDSKIN' WP4 with use cases that require confidential data processing.

2.2 Role of WP4 in the CLOUDSKIN Context

As the unified execution layer of CLOUDSKIN, WP4 needs to interface with the ephemeral data storage offered by WP3 and the learning-plane offered by WP5.

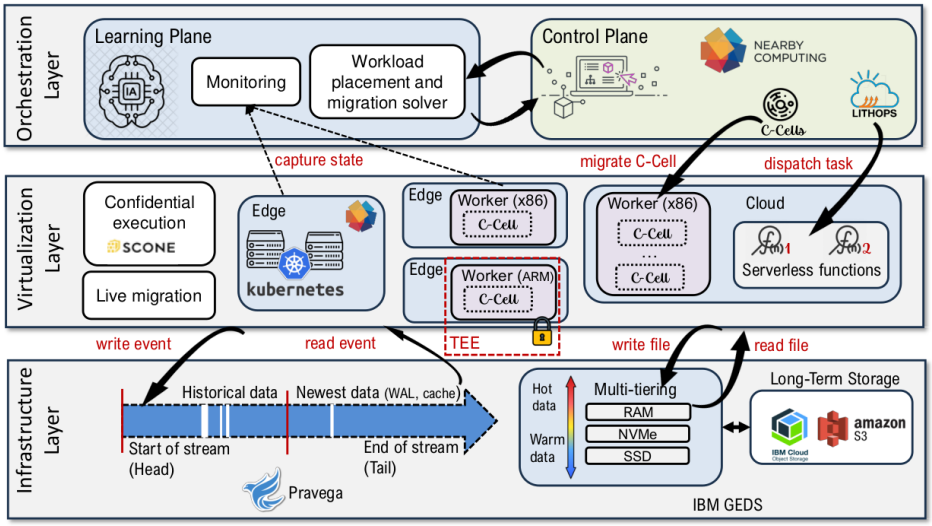


Figure 1: CLOUDSKIN Architecture Overview

Figure 1 shows the high-level architecture of CLOUDSKIN, and the role of WP4 in relation to other work packages, and the overall system. As previously introduced, WP4 provides the virtualization layer for the cloud-edge continuum. It offers lightweight isolation of applications via the C-Cell abstraction, as well as live migration and confidential execution via GRANNY and Sconified Pravega.

The latter integrates with the ephemeral and steaming infrastructure layer (i.e. Pravega), whereas the former integrates with the learning plane by exposing configurable orchestration policies that are workload- and infrastructure-aware.

A Sconified Pravega is a Pravega deployment in which server components are executed inside Trusted Execution Environments (TEEs) using the SCONE framework. SCONE "sconifies" applications by placing them inside Intel SGX enclaves, thus transparently encrypting memory and network traffic and enforcing attestation before starting up, so applications and data are protected from the host OS, hypervisor, or cloud operator. Using this Pravega version in the infrastructure layer brings benefits when handling sensitive real-time streams (e.g. medical video analytics) with minimal changes to code, auditable trust through attestation, and a balanced security performance profile suitable for production.

WP4 is, by design, a horizontal work-pacakge, where we provide a unified execution layer that suits all targeted use-cases of the cloud-edge continuum. To show the vertical integration of the different layers, as part of our use-case integrations, we build systems that utilize components from each layer. For example, we developed an image analysis pipeline that consumes medical images from a Pravega instance into a secure C-Cell to extract embeddings, without any personal identifiable information. It then transfers those results to an OpenMP job that performs ML inference in parallel, where the degree of parallelism (i.e. the number of C-Cells involved in the computation) is decided elastically, by an algorithm in the orchestration layer. This algorithm uses lightweight monitoring and decides when to add or remove C-Cells from the computation. This whole prototype is deployed on KIO networks' cloud-edge testbed.

2.3 Scope and Non-Goals of D4.3

D4.3 consolidates the different prototypes of WP4, namely GRANNY and Sconified Pravega, into a single platform, and delivers their reference implementation and open-source distribution and maintenance. D4.3 also demonstrates the prototype integration with a third-party cloud-edge provider (KIO) and use-cases from WP5 and WP6. More importantly, D4.3 does not re-explain the detailed background from D4.1 and D4.2, even though their core contributions are summarized in Section 3.

Table 1 summarizes the focus areas of the different deliverables in WP4, their contributions, and their relationship to D4.3.

Deliverable	Focus Area	Contribution	How D4.3 Extends It
D4.1 Initial C-Cell prototype	T4.1—C-Cell abstraction and early prototype.	Introduced first C-Cell design including WebAssembly based isolation, and early SCONE integration.	D4.3 unifies the abstractions into a single execution layer, and showcases it via vertical use-case integrations.
D4.2 C-Cell release candidate and GRANNY	T4.2 + T4.3—Virtualisation and orchestration for C-Cells.	Introduced GRANNY: full distributed execution of C-Cells, with live migration of MPI and OpenMP applications and resource orchestration policies.	D4.3 integrates AI-enabled policies from the learning plane (in the use-case) via Grafana monitoring.
D4.3 Final reference implementation	Full WP4 scope (T4.1–T4.4).	Reference system implementation and KPI validation.	N/A

Table 1: Evolution of WP4 deliverables from initial prototype to final reference implementation.

2.4 WP4 Achievement Summary

As D4.3 is the last deliverable for WP4, we also summarize the key achievements of this work package:

Core Technical Achievements

- Design and implementation of CLOUDSKIN's universal execution layer for the cloud-edge continuum.
- Design and implementation of transparent, efficient live migration of distributed applications via control-points.
- Adaptive virtualization using WebAssembly and trusted execution environments, supporting a wide-variety of use-cases.
- Transparent confidential execution through the integration with SCONE with modest execution overhead.

Reference Implementation & Open-Source

- GRANNY is a distributed system using C-Cells for the execution, management, and orchestration of unmodified scientific applications using MPI and OpenMP. GRANNY was presented at USENIX NSDI 2025 [1], and is open-sourced as part of the Faasm serverless runtime with almost one thousand stars on Github. For dissemination, we also forked the repository under the `cloudskin-eu` GitHub organization.[1]
- Sconified Pravega demonstrates a confidential data streaming stack by integrating the open-source Pravega [2] data streaming platform and the SCONE [3] confidential computing platform.[2]

Deployment & Validation

- We validate our reference implementations through a full-deployment on KIO's cloud-edge testbed, including SGX-enabled nodes.
- We showcase the fulfilment of all the targeted KPIs.

[1] `https://github.com/cloudskin-eu/granny`
[2] `https://github.com/cloudskin-eu/scone-pravega/`

3 Background From Previous Deliverables

This deliverable acts as a colophon to WP4, and builds, as shown in Table 1, on the previous deliverables D4.1 and D4.2. In this sectionm, we briefly give relevant background.

3.1 Deliverable D4.1

D4.1 introduces C-Cells and provides extensive background on its key enabling technologies, WebAssembly and SCONE. It also describes the first implementation draft, based on Faasm's Faaslets [4]. D4.1 also provides an early integration of Faaslets with SCONE. For efficiency and integration purposes, the confidential C-Cell support presented in this deliverable deviates slightly from that prototype. In the evaluation section, we present early results on multi-thread and multi-process support in WebAssembly. We show that WebAssembly's inherent overheads are low, but distributed shared memory, message passing, and confidential execution, if not done properly, can introduce substantial runtime overheads.

3.2 Deliverable D4.2

D4.2 focuses primarily on GRANNY, the first full-system implementation using C-Cells. GRANNY's key contribution is a distributed runtime for C-Cells that manages the comunication and (shared) state of each C-Cell. To demonstrate how GRANNY' design can enable a variety of workloads with limited overhead, we introduce support for unmodified scientific applications using OpenMP and MPI. We execute various molecule dynamic simulation applications as well as linear algebra programs and show limited overhead. We also implement live migration and elastic scaling of MPI processes and OpenMP threads, respectively, and implement basic resource orchestration policies that utilize this mechanisms to improve some target, cluster-wide, metrics.

4 Final CLOUDSKIN Execution Runtime

This section presents the final overview of the execution layer for CLOUDSKIN and summarizes its key design choices, features, and deployment.

4.1 Overview

CLOUDSKIN uses C-Cells as a universal execution unit across the cloud-edge continuum. As a universal execution layer, C-Cells must support a variety of programming languages and hardware deployment modes across cloud and edge nodes. The choice of WebAssembly as intermediate representation and sandboxing solution automatically ticks all of these boxes. In order to provide multi-thread, multi-process, snapshotting, migration and TEE support we extend the WebAssembly abstraction from the simple Faaslet interface, to what we now refer to as C-Cells, as we describe next.

4.2 Evolution of the C-Cell Abstraction

C-Cells were initially single-node, single-thread, and only supported simple applications. We then implemented control-points as lightweight trap mechanism to transfer execution control from the C-Cell to the host runtime using WebAssembly's native symbol interface. Additionally, using WebAssembly's linear memory layout, we implement efficient C-Cell snapshotting. Collectively, control-points and snapshots are the building blocks to implement efficient inter-VM snapshot-restore which, in turn, enabled two key integrations:

1. **Rich application support** C-Cells support unmodified execution of a variety of applications written in a variety of programming languages. From multi-process applications written in MPI, multi-threaded applications written in OpenMP, and machine-learning inference pipelines written in Rust. As part of the evaluation results presented in D4.2, for example, we execute dynamic molecule simulations using LAMMPS [5], an open-source C++ framework with tens of thousands of lines of code and stars on Github.

2. **Resource orchestration policies** In order to enable the efficient management of C-Cell-powered applications, C-Cells must support low-overhead, management to effectuate the policies indicated by the learning plane. Via control-points, we implement snapshotting and live migration, achieving migration times of less than 100 ms.

KPI	Description	WP4 Contribution
KPI1	Performance parity between instrumented and non-instrumented workloads.	C-Cells introduce **less than 5% runtime overhead**
KPI2	Transparent migration	C-Cell migration with **sub-100 ms distruption**
KPI3	Heterogeneous execution	C-Cells execute **across the cloud-edge continuum**.
KPI4	Trusted execution with minimal overhead.	Confidential C-Cells via SCONE execute in **SGX enclaves**.
KPI5	Support for complex applications.	Distributed C-Cells can execute **unmodified MPI/OpenMP applications**.
KPI6	Dynamic elastic scaling.	Vertical scaling **reduces idle vCPUs by 20-40%** for OpenMP applications.
KPI7	Support for bursty, short-lived workloads.	C-Cell start-up time is **less than 10 ms**.
KPI8	Real-world testbed deployment.	Deployment of prototypes on KIO network's testbed.

Table 2: Relationship between CLOUDSKIN KPIs and WP4's contributions

This evolution of the C-Cell abstraction has enabled the achievement of all the relevant KPIs, as we showcase in Table 2. KPIs 1 and 2 require performance parity between C-Cell-enabled applications and their native counterparts, as well as minimal impact from live migration. Performance

parity is achieved through the use of WebAssembly, which also enables seamless execution across the cloud-edge continuum in KPI 3. Similarly, confidential C-Cell execution is achieved with the integration with SCONE, as exemplified by the Sconified Pravega prototype, achieving KPI 4. C-Cells also implement low-overhead migration, dynamic elastic scaling, and support for complex applications, thus fulfilling KPIs 2, 5, and 6. Finally, in this deliverable, we present the deployment on heterogeneous clusters within KIO networks, fulfilling KPI 8.

4.3 Confidential C-Cells with TEE Integration

C-Cells are designed to run arbitrary applications in a sandboxed environment. In addition, to protect the sandboxed application from its host environment, thus protecting the confidentiality and integrity of both the C-Cell and the application, C-Cells support being executed inside Intel SGX enclaves. We implement a prototype of confidential C-Cells using SCONE's lift-and-shift approach. Since SCONE has additional features to enhance confidentiality and trust, such as centralized attestation, network shield, and file system shield, a more complete approach can be considered.

In general, the confidential C-Cell is built as follows. First, we identified the feature incompatibilities. For example, early C-Cell adoption assumes a glibc-based interface, while SCONE is based on musl-libc. Due to this migration from glibc to musl-libc, some system calls used by C-Cells are not supported, so we replaced them with comparable alternatives. We apply this modifications directly on the C-Cell runtime's source code, enabling greater compatibility. As a result of this process, C-Cells with TEE support are finally achieved.

4.4 Heterogeneous deployment on the cloud-edge continuum

To support real-world deployments across the cloud-edge continuum, C-Cell-enabled applications must support a variety of deployment modes on different hardware architectures. Using KIO's infrastructure, we implement support for C-Cells to be deployed and orchestrated using `docker compose` and Kubernetes by packaging the WebAssembly runtime and core SGX dependencies inside regular OCI container images. On KIO's testbed we use x86 servers with SGX-enabled VMs, and illustrate how confidential and non-confidential C-Cells can co-exist.

5 Reference Implementations

This section describes the design and implementation of two fully-fledged systems that use C-Cells for different goals: (1) GRANNY for the dynamic management of scientific applications, and (2) Sconified Pravega for the secure execution of event-processing applications.

5.1 GRANNY

GRANNY executes multi-threaded and multi-process applications as a set of C-Cells (shown as circles in Figure 2). A C-Cell represents a single thread of execution, and has its own mappings for code and data: a multi-threaded application (OpenMP) is therefore a set of C-Cells with shared-memory code and data mappings; a multi-process application (MPI) is a set of C-Cells with non-shared data mappings.

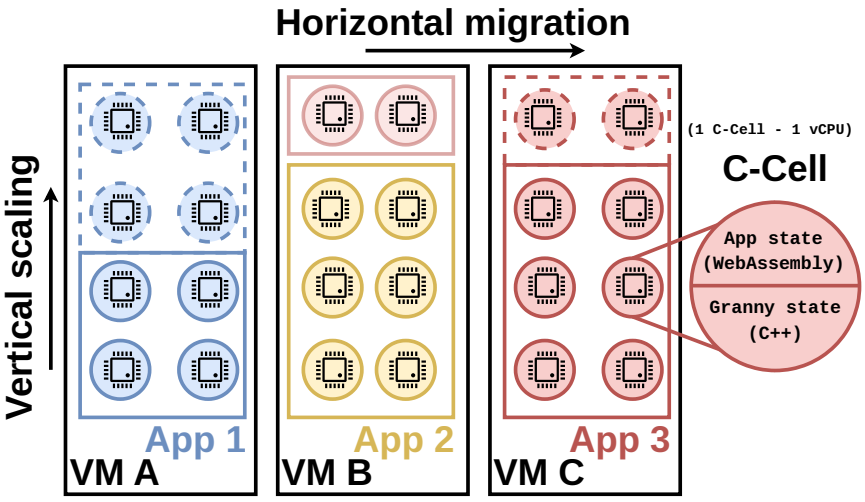


Figure 2: GRANNY executes applications as sets of C-Cells

Each C-Cell runs in a single WebAssembly [6] module with its application code, and executes on a vCPU in a VM. The GRANNY runtime decouples C-Cell state from OS kernel state, which makes the snapshotting of C-Cells simpler and faster than traditional process checkpoints [7]. It also allows GRANNY to execute multiple C-Cells within the same process (shown as different colors in Figure 2), because WebAssembly modules are isolated from each other and the runtime. Each VM runs a single instance of the GRANNY runtime, and there is also one cluster-wide GRANNY scheduler. Currently, the scheduler applies very simple policies based on heuristics, but in the future it could be connected to CLOUDSKIN' learning plane for AI-enabled orchestration.

GRANNY uses the C-Cell abstraction and snapshots to implement management: it can spawn C-Cells with thread semantics to vertically scale multi-threaded applications, and it can horizontally migrate the snapshot of a C-Cell with process semantics. The GRANNY scheduler uses these management actions to implement dynamic scheduling policies: as shown in Figure 2, it can speed up app 1 with the newly released vCPUs of VM A, improving utilization, or consolidate app 3 to VM C, reducing fragmentation and improving locality.

5.1.1 C-Cell abstraction

Figure 3 shows the GRANNY architecture. A C-Cell executes as an OS thread inside the VM where it is spawned, with application code deployed in an isolated WebAssembly module that only has external access to pre-defined OpenMP, MPI and POSIX operations implemented by the GRANNY runtime (the various backends in Figure 3). The GRANNY runtime backends are built on top of a shared GRANNY runtime core that keeps the per-C-Cell state, and decouples backends from the OS.

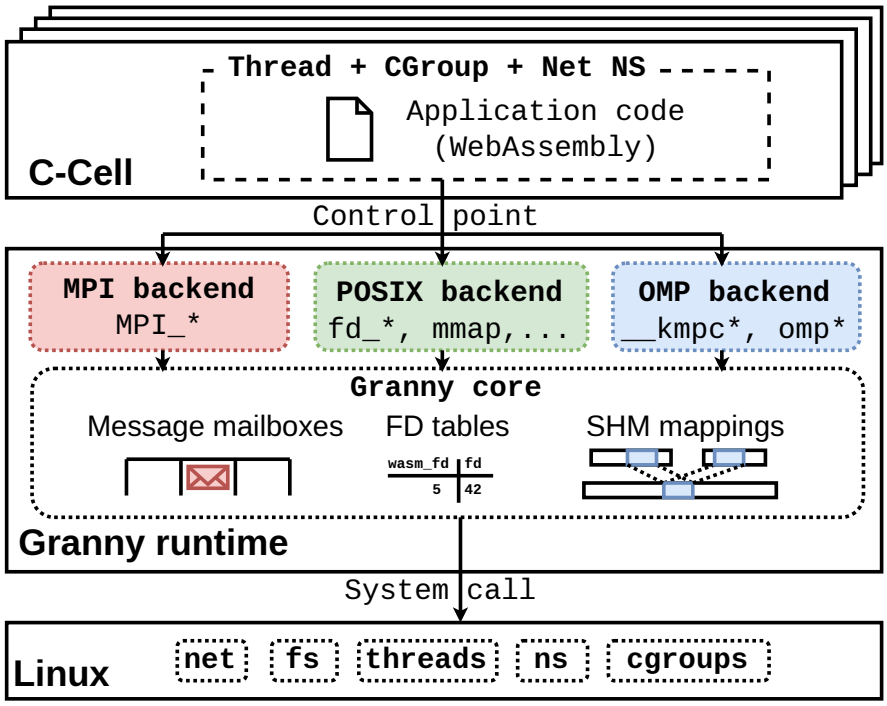


Figure 3: GRANNY architecture

Applications must be cross-compiled to WebAssembly [6], a memory-safe and platform-independent binary instruction format that supports a wide range of programming languages. The use of WebAssembly allows C-Cells to execute side-by-side within one virtual address space, together with the GRANNY runtime, while enforcing memory safety [8] and reducing interaction with the privileged OS kernel. In addition to the file-system and network operations necessary for multi-process applications, GRANNY only relies on the OS kernel to schedule threads and guarantee resource (CPU and network) fairness.

WebAssembly code cannot, in general, interact with its host environment. In GRANNY, application code can use control-points, C-Cells' native trapping mechanism, to interact with the runtime. A control-point is triggered by a call to one of the supported APIs, POSIX, MPI, and OpenMP, implemented in the backends. Control-points come at no cost for developers and at little cost for the runtime: they are injected by leaving the corresponding API symbol (e.g. `MPI_Barrier`) as undefined during cross-compilation and marked as a function import [9]. The symbol is resolved at runtime and triggers a WebAssembly context switch executed in tenths of cycles, similar to a function call [10].

The GRANNY runtime makes, whenever necessary, system calls to the underlying OS, e.g. to send cross-VM messages or write to a file descriptor. It records these interactions to ensure that C-Cell state can be encapsulated in a snapshot.

5.1.2 C-Cell implementation

Control-points. Our CLOUDSKIN prototype currently supports three backends:

- The **MPI** backend implements the standard `MPI_*` APIs, e.g. `MPI_Reduce` [11] and provides reliable C-Cell-to-C-Cell messaging. It uses in-memory message mailboxes for message reception, and the kernel's network stack for cross-VM messaging. The GRANNY runtime maintains consistent C-Cell addressing tables across C-Cell migrations.
- The **OpenMP** (OMP) backend implements the interface exposed by LLVM's OpenMP runtime

(`libomp`) after OpenMP pragmas have been expanded, e.g. `__kmpc_fork_call` [12]. For correct and safe OpenMP execution, the GRANNY runtime must carefully manage the shared and private memory regions of different C-Cells.

- The **POSIX** backend implements WebAssembly's standard system interface (WASI) [13]. The adoption of WASI simplifies the effort of cross-compiling large codebases as we can statically link applications with WASI-aware libraries such as `wasi-libc` [14]. The backend only implements the symbols needed by compute-intensive applications, which are mostly file-system APIs. To maintain a consistent C-Cell state and facilitate snapshots and migration, the GRANNY runtime maps the file-descriptors used internally in WebAssembly code, to the OS file-descriptors used in practice.

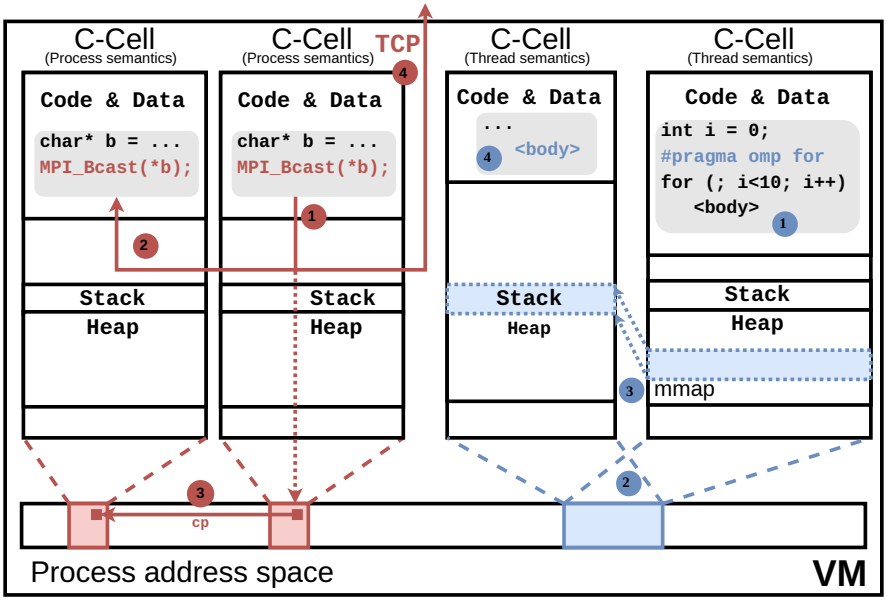


Figure 4: C-Cells memory layout in a single GRANNY instance

Memory layout. Due to WebAssembly's linear memory model [6], C-Cells have a simple memory layout, shown in Figure 4, which facilitates spawning and snapshotting. A C-Cell occupies a contiguous region of virtual memory with the code, data, a stack and a heap. Spatial isolation comes from a combination of WebAssembly's memory safety and guard pages, and is enforced by the WebAssembly runtime [15].

The main difference between a C-Cell executing with process semantics and with thread semantics is their memory layout (red and blue C-Cells in Figure 4, respectively): C-Cells with processes semantics have separate linear memories, with the same application mapped into them. To send messages, the sender C-Cell indicates the buffer to be sent (Figure 4, ❶-red), and the MPI backend in the CLOUDSKIN runtime decides if it is a local or a remote message. For a local message, the runtime can directly enqueue the message metadata (❷) and copy the buffer contents into the reception buffer (❸). For a remote message, the runtime will send the whole payload over TCP to the appropriate CLOUDSKIN runtime instance (❹).

C-Cells with thread semantics share a linear memory, and have separate stacks. When a C-Cell creates new threads, such as with OpenMP `#pragma omp for` (Figure 4, ❶-blue), the GRANNY runtime spawns a new C-Cell mapped to the same linear memory (❷). To give each thread a separate execution context, but maintain WebAssembly's memory sandboxing, GRANNY allocates an area in the parent's heap for the child's stack (❸), and sets the child's code entrypoint to the corresponding

OpenMP task (❹). Additional care goes into maintaining shared variable visibility, as well as their consistency.

Snapshots. C-Cells consequently have a simple memory layout, which means that their execution state can be captured combining the span of the linear memory, together with the C-Cell state in the GRANNY runtime (Figure 3). The combined linear memory and runtime state is what we call a C-Cell snapshot. Having a concise snapshot representation for C-Cells is an essential requirement for migration. Since all state is contained in a snapshot, GRANNY does not require OS kernel modifications to obtain a C-Cell's full execution state. This is in contrast to general process checkpointing [7], which must also extract process state from the OS kernel.

5.1.3 C-Cell Management

GRANNY performs management actions, vertical scaling, and horizontal migration by interrupting application execution at well-defined control-points. This ensures GRANNY always operates with a consistent application state. We next describe these operations in more detail.

(a) Interrupting C-Cells at control-points

GRANNY takes control over C-Cell execution at control-points, i.e. at every call to one of the supported runtime backend APIs (see Figure 3). GRANNY defines two types of control-points, regular and barrier control-points, that enable the runtime core to perform different operations. At regular control-points, such as calls to the POSIX API, the application state is not guaranteed to be consistent, but this still allows GRANNY to send point-to-point messages or operate on shared memory. In contrast, barrier control-points guarantee that the application state is consistent, i.e. no messages are in-flight and no modifications to shared variables are pending to be synchronized. Therefore, GRANNY can only perform management operations at barrier control-points.

Differentiating between regular and barrier control-points requires semantic knowledge of the shared memory/message passing API, as well as control over its implementation. For example, GRANNY's MPI backend implementation of `MPI_Barrier` has a reduce phase in which all C-Cells send messages to the C-Cell with the lowest MPI rank, and a broadcast phase in which all C-Cells are notified that the barrier has completed. After the reduce phase, the C-Cell with the lowest MPI rank can rely on the fact that there are no outstanding messages and has thus reached a consistent state. A similar explanation, with shared memory synchronization instead of messages, applies to GRANNY's OpenMP backend implementation of `#pragma omp barrier`.

As a consequence, vertical scaling and horizontal migration in GRANNY is a co-operative process. When a C-Cell triggers a barrier control-point, the runtime interacts with the scheduler and adds, removes, or migrates C-Cells as indicated. In practice, barrier control-points are frequent enough, so that their co-operative nature does not hinder the benefits in resource management, as we saw extensively in D4.2.

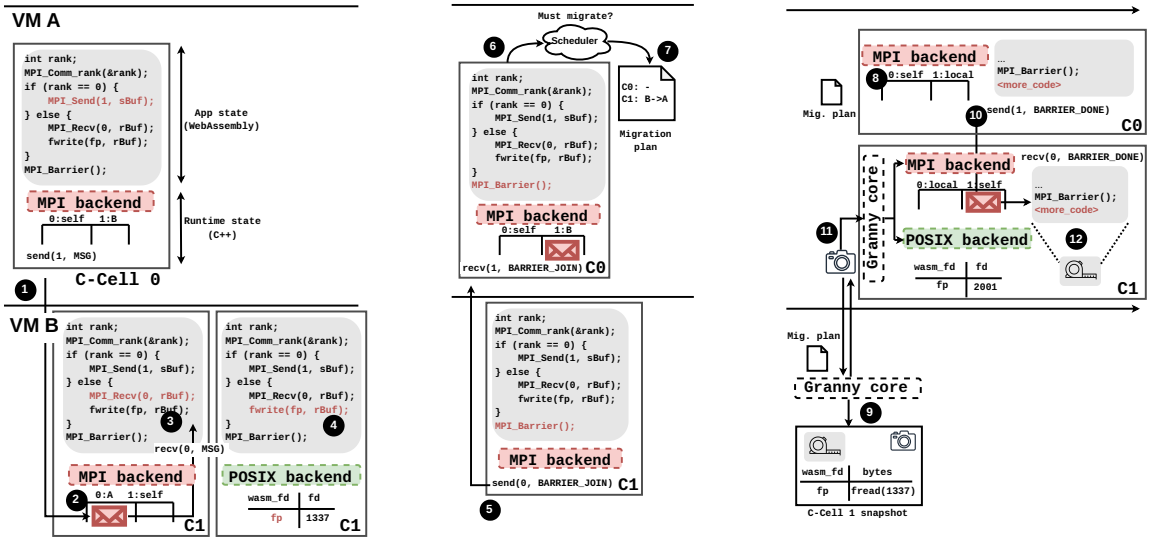
(b) Vertical scaling

Implementing vertical scaling in GRANNY is straightforward if we take into account the memory layout presented in Figure 4. To create a child C-Cell from a parent C-Cell, the runtime allocates the child's stack in the parent's heap, and instantiates a new C-Cell on top of the same WebAssembly linear memory, changing only the stack pointer. The child's entrypoint is indicated as an index in the WebAssembly module's function table. We also add guard pages around each child's stack to mitigate potential stack overflows (typical of various threading implementations in WebAssembly [16]).

Since vertical scaling happens during barrier control-points in OpenMP, the runtime can easily distribute the work across any C-Cells by simply setting the appropriate per-thread and global variables in the OpenMP specification. We design a scheduling policy using vertical scaling and showcase it in one of the use-case integrations.

(c) Horizontal migration

Figure 5 illustrates horizontal migration. The MPI backend gives each C-Cell a unique integer identifier, akin to the MPI rank, and implements messaging as a three-step process: it captures the send



(a) Normal execution (b) Barrier control-point (c) Migration

Figure 5: Horizontal migration overview

operation as a call to a backend operation such as `MPI_Send` (❶); it copies the message into the receiving C-Cell's mailbox within the target GRANNY runtime core (❷); and it delivers the message to the application by capturing (❸), and potentially blocking, calls to a receive operation (which the MPI backend matches to the corresponding mailbox, according to the MPI specification). Note that C-Cells may also call other API backends, such as file writes via the POSIX backend (❹). As previously introduced, the POSIX backend serves the request and keeps track of the mapping between WebAssembly file descriptors and OS ones.

When C-Cells reach a barrier control-point (such as `MPI_Barrier` in Figure 5b), the runtime can perform horizontal migrations safely. For example, in the case of `MPI_Barrier`, all C-Cells send a `BARRIER_JOIN` message (❺) to the zero-th MPI rank (C-Cell 0 in VM A in the example), and wait for a `BARRIER_DONE` message before continuing. When the runtime for C-Cell 0 has received all messages, it queries the scheduler for any horizontal migrations (❻). The resource scheduler then applies the scheduling policy and returns a migration plan (❼). After all migrations have been performed (if any), the runtime broadcasts a `BARRIER_DONE` message so all blocked C-Cells resume execution. In our example, C-Cell 1 must be migrated from VM B to VM A.

To perform the migration, shown in Figure 5c, the runtime in VM A distributes the migration plan to the other runtimes involved in the execution (in this case the runtime in VM B), and all proceed to individually prepare for the migration. For VM A, this means the runtime must create a new, empty C-Cell 1, setup its mailbox mappings, and update the entry for C-Cell 1 on the mailbox of C-Cell 0 (❽). For the runtime in VM B, this means creating a snapshot of C-Cell 1, including any runtime state, and terminating the old C-Cell 1 (❾).

Once the migration has been prepared, the runtime in VM A can resume execution of C-Cell 0, which broadcasts the `BARRIER_DONE` message to the updated mailboxes (❿). The runtime in VM A can also restore C-Cell 1 from the received snapshot, re-construct the file descriptor tables, and resume execution where it left off, i.e. blocked waiting for a `BARRIER_DONE` (⓫). Since the restored C-Cell 1 uses the updated mailboxes, it has the message and resumes execution (⓬). In D4.2 we showcased two scheduling policies using horizontal migration and their impact on fragmentation and cloud costs.

5.2 Sconified Pravega

Pravega, as one of the solutions for data processing architecture, is not equipped with the confidential computing approach. We leverage the SCONE framework with its *lift-and-shift* approach to enable Pravega to be able to run on top of the Trusted Execution Environment (TEE), such as Intel SGX. Essentially, SCONE's toolchain converts Pravega to run on top of SCONE's runtime rather than the native one. This runtime is designed to leverage TEE's hardware features, such as memory isolation, integrity, and encryption, on by default, out of the box. A powerful adversary would not be able to inspect Pravega's memory footprint since it now runs with a TEE.

Given how integral Pravega plays a role in the data distribution, a powerful adversary (such as cloud service providers with root and physical access to the server) could steal confidential data on where Pravega is deployed. Encrypting the data at rest could prevent those; however, it is still susceptible to rollback attacks. Skilled adversaries may be able to inspect the memory region of running Pravega and infer something from it. With the introduction of TEE, the latter would be prevented, hence one can trust the execution. SCONE introduces some features that can extend the trust to also include the storage and network connection with its filesystem- and network- shield, respectively. Those protection shields work transparently in the C-library at the system call level. A shielding layer encrypts the data before writing it out to the interfaces, as well as decrypts the data transparently after reading. Therefore, both data in transit and data at rest could be encrypted, thereby extending trust to them. In the future, a comprehensive solution could also be included in this aspect.

Pravega consists of many components, as shown in Table 3. For the current deliverable, we focus solely on the parts that are developed by DELL: The Controller and Segment Store. The Controller is responsible for providing the abstraction of the Stream and handling its lifecycle. Segment Store manages the Stream segments internally with respect to modifying the contents. Both are an integral part of the Pravega.

The rest of the components are external projects. We assume that the deployment and execution of external project components are done in a trusted way. Zookeeper and Bookkeeper are used by Pravega in order to handle coordination and journaling. HDFS is a file system that stores data in a distributed way. Pravega may be used in tandem with other open source tools that are outside of the scope, such as Hadoop, Flink, etc.

Table 3: Status of Pravega Components in SCONE Environment

Pravega Component	Previous Status	Current Status
Controller	Vanilla	Sconified
Segment Store	Vanilla	Sconified
Zookeeper	Vanilla	Assumed to be trusted (external project)
Bookkeeper	Vanilla	Assumed to be trusted (external project)
HDFS	Vanilla	Assumed to be trusted (external project)

5.2.1 Building Sconified Pravega

The goal of using SCONE is to make minimal modifications to the original application to make it compatible with the confidential computing capabilities using TEE. Both the Controller and Segment Store are based on Java; therefore, we first convert the Java runtime. By default, the Java runtime we used is dynamically linked with the glibc. We modify the runtime's header (in the binary) so that it uses SCONE libc and is therefore compatible with the TEE.

The second step is to ensure that both the Controller and Segment Store do not violate restrictions in the TEE and SCONE runtime. For example, using asynchronous input/output with the kernel is not supported. Getting OS-wide information from the `/proc` file system is restricted. Both restrictions

Table 4: Status of Pravega Components in SCONE Environment

Configuration	Description	Sample
SCONE_SLOTS	System calls queue length	384
SCONE_VERSION	Flag to enable environment launch information	1
SCONE_TCS	Number of Thread Control Structures	48
SCONE_MODE	Execution mode in SCONE (Hardware/Simulation)	hw
SCONE_HEAP	Heap memory allocated to the enclave	32G
SCONE_ESPINS	Idle iterations the in-enclave scheduler does before it exists the enclave and goes to sleep	1000000
SCONE_SSPINS	Backoff behaviour on the enclave scheduler when waiting for a system call invocation before it goes to sleep	500
SCONE_SSLEEP	Amount of sleeping when there is no system call invocation	1

are there to protect the application from possible attack , such as from kernel. Fortunately, after thorough analysis, there is no part of the Pravega code to date that violates this restriction. This fact allows us to convert the Controller and Segment Store with minimal changes.

However, in the default configuration, certain fields require adjustment. For example, one that sets up the maximum cache size. In a vanilla implementation, the application will allocate the maximum cache size in the memory and immediately release it. At this point, the application cooperates with the kernel to manage such memory allocation. In a limited memory environment, such as in the TEE, we have to be conservative in terms of declaring memory allocation. Therefore, we set up the maximum cache size to be equal to the heap size that the enclave can handle.

5.2.2 Deploying Sconified Pravega

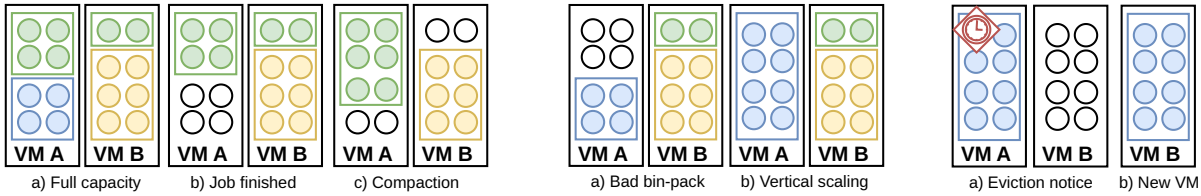
A Pravega cluster requires a set of components such as Zookeeper, Bookkeeper, tier storage, controller, and Segment Store. As mentioned in Table 3, we focus on deploying the Controller and Segment store in the TEE. Those two are pipelined into SCONE's toolchain, and we produced Docker images for them, including manually configuring them. Mandatory configuration includes adding the SGX drivers, so the SGX Enclave most be added to the devices and the SGX environment variables most be added to the container. Table 4 shows the specific SCONE configuration that should be in the deployment docker compose yaml file.

The following is an example of the SCONE configuration in the Docker Compose deployment file. The complete file and other deployment files are available on Github repository.[3]

Listing 1: Example of SCONE configuration file

```
    ......
    devices:
      - /dev/sgx_enclave:/dev/sgx_enclave
    environment:
      SCONE_SLOTS: 384
      SCONE_VERSION: 1
      SCONE_TCS: 48
      SCONE_MODE: hw
      SCONE_HEAP: 32G
      SCONE_ESPINS: 1000000
      SCONE_SSPINS: 500
      SCONE_SSLEEP: 1
    volumes:
      - ./sgx-musl-controller.conf:/etc/sgx-musl.conf
      ......
```

[3] https://github.com/cloudskin-eu/scone-pravega/



(a) Compaction policy (b) Elastic policy (c) Spot policy

Figure 6: Dynamic scheduling policies implemented for their use with GRANNY (Different colors indicate different jobs.)

6 Evaluation Summary and KPIs

We now describe the evaluation result and the validation activities to demonstrate the fulfilment of KPIs for both, GRANNY and Sconified Pravega.

6.1 GRANNY

As part of GRANNY's evaluation we implement three scheduling policies, described in the following sections, to improve the utilization and performance of cloud-based multi-process and multi-thread scientific applications. The policies use GRANNY's support for vertical scaling and horizontal migration, and treat the scheduled applications as black boxes, i.e. without knowledge of future applications, their distribution of sizes, or their expected duration. These policies are deliberately simple, and could be extended with more complex policies provided by the learning plane from WP5.

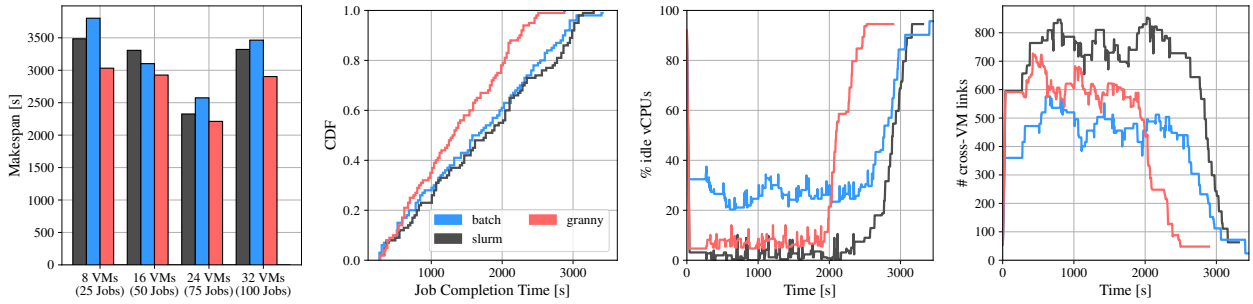
As baselines, we compare against two cloud resource schedulers, `batch` and `slurm`, that mirror the behaviour of Azure Batch [17] and Slurm [18], respectively. Unless otherwise stated, `batch` and `slurm` execute MPI applications with OpenMPI v4.1 [19] and OpenMP ones with `libomp` v4.5 [20]. We deploy GRANNY and any baselines on a Kubernetes cluster on Azure. The cluster consists of Standard_D8_v5 VMs [21] with 8 vCPU cores and 32 GB of memory. We deploy the resource scheduler in a separate VM in the same cluster.

(a) Improving locality with compaction policy

When scheduling MPI applications, existing cloud schedulers face a utilization/locality trade-off: they can either assign available vCPUs to an MPI application to achieve high utilization, or assign an entire VM to an MPI application to achieve high locality at the expense of having unused vCPUs in those VMs. The former reduces queue wait times for applications, but it may increase execution time, because applications may have to communicate across different VMs; the latter optimizes execution time at the cost of increasing queuing time due to the wait for an available VM of the required size.

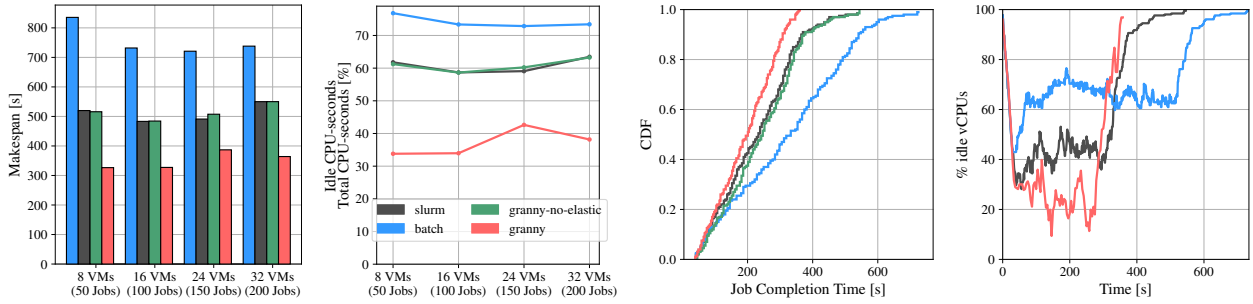
Instead, we adopt a compaction policy (see Figure 6a) that uses GRANNY to combine the benefits of both approaches. The scheduler first eagerly deploys MPI applications on any available vCPUs, which results in the highly fragmented green application in our example. When vCPUs are later released (e.g. the blue application terminates), the scheduler performs horizontal migration of C-Cells to increase the locality of executing applications. This results in less cross-VM communication, which speeds up the green application. We define our fragmentation metric as the total number of C-Cell-to-C-Cell connections that cross a VM boundary in an application. When the application reaches a barrier control-point, the scheduler checks if it can reduce its cross-VM links by performing any set of horizontal migrations.

Figure 7 shows various performance metrics, as we increase the number of VMs and the number of MPI applications in the trace. Figure 7a shows that the compaction policy improves end-to-end execution time (makespan) by up to 20%. Using compaction, GRANNY always improves makespan across all baselines and cluster sizes. To show the compaction policy in action, Figure 7c and Figure 7d plot the time-series of idle vCPUs (as a proxy for cluster utilization) and cross-VM network



(a) Makespan (b) CDF JCT (32 VMs) (c) Idle vCPUs (32 VMs) (d) xVM Links (32 VMs)

Figure 7: Improving locality with compaction policy



(a) Makespan (b) Aggregate idle vCPUs (c) JCT (32 VMs) (d) Idle vCPUs (32 VMs)

Figure 8: Improving utilization with elastic policy

links (as a proxy for locality) for the (32 VMs, 100 jobs) execution. We see that, differently to Slurm, GRANNY deliberately leaves a percentage of vCPUs idle corresponding to a target utilization (5% in this experiment). GRANNY can use these spare vCPUs to defragment applications at runtime by performing horizontal migrations, achieving consistently 25% less fragmentation than Slurm. In fact, GRANNY, with only 5% idle vCPUs, is closer in terms of fragmentation to Azure Batch, which behaves optimally with respect to this metric but leaves 30% of vCPUs unused.

Figure 7b shows that this reduction in fragmentation at high cluster utilization has a direct impact on job completion time (JCT). For the (32 VMs, 100 jobs) execution, GRANNY improves median and tail JCT by up to 20%. We conclude that the compaction policy enables GRANNY to exploit the utilization/locality trade-off more effectively compared to Azure Batch and Slurm.

(b) Improving utilization with elastic policy

Even with a compaction policy, some cluster resources remain idle due to the nature of bin-packing scheduling. VMs may have some spare capacity that is insufficient to deploy the next application in the queue. This is particularly true with OpenMP applications that cannot be distributed across VMs.

In response, we introduce an elastic policy (see Figure 6b). When there is spare capacity in a VM (Figure 6b-a) and an OpenMP application reaches a barrier control point, the scheduler triggers a vertical scale-up to utilize these idle resources (Figure 6b-b). This improves cluster utilization and application performance by exploiting extra parallelism. Our policy is careful to not mistake fork-join patterns of co-located OpenMP application with truly available resources by keeping track of the `OMP_NUM_THREADS` environment variable.

We execute a trace of OpenMP applications as jobs, and each application executes a large-scale

version of the p2p ParRes OpenMP kernel [22], which performs a compute-intensive pipelined parallel algorithm on a large matrix, and requires a different number of OpenMP threads. In this experiment, the baselines (batch and slurm) execute applications using `libomp` [20]. We include an additional baseline, granny-no-elastic, which corresponds to slurm using GRANNY's OpenMP backend.

Figure 8a shows that GRANNY improves makespan by up to 60% compared to the native baselines and GRANNY without the elastic policy (granny-no-elastic). This confirms that the performance improvements come from the policy. Indeed, we can assert that GRANNY reduces the idle CPU cores in the cluster by up to 30% (Figure 8b), and it uses these extra cores to improve median and tail JCT by up to 50% (Figure 8c).

This large performance gap can be understood when considering how many compute resources are left idle by the native baselines. Figure 8d shows a time series of the percentage of idle vCPUs when running 200 jobs on a 32-VM cluster. Azure Batch and Slurm, even when the job queue is not empty, consistently leave 60% and 40% of vCPUs idle. This is due to multiple reasons: (i) OpenMP applications have fixed parallelism; (ii) it is not possible to distribute them across VMs; and (iii) in the case of Azure Batch, it is not possible to run different applications on the same VM concurrently. By vertically scaling-up, GRANNY maintains the fraction of idle vCPUs at around 20% while there are still pending jobs.

CLOUDSKIN's improvements in terms of JCT are partly due to the fact that our OpenMP workload always benefits from increasing its parallelism. In a real deployment, an elastic policy should be accompanied by runtime profiling to determine when a workload exhausts its parallelism.

(c) Ephemeral VMs with spot policy

Compute-intensive applications may be deployed on a pool of spot VMs to benefit from lower costs. This introduces the challenge of handling partial failures, because spot VMs are withdrawn by the cloud provider after a short grace period.

Figure 6c shows our spot policy. We follow a two-part horizontal migration approach: (1) when the cloud provider notifies the scheduler of an upcoming spot VM eviction (Figure 6c-a), the scheduler stops scheduling C-Cells to that VM; (2) when a C-Cell running on the to-be-evicted VM reaches a barrier control-point (e.g. `MPI_Barrier` or `#pragma omp barrier`), the resource manger tries to reschedule it on the remaining VMs (Figure 6c-b). If there are insufficient resources, snapshots are taken of all C-Cells in the application, and they are terminated. Interrupted applications are then added to the beginning of the scheduler's queue.

6.2 Sconified Pravega

Modern computer-assisted surgery (CAS) systems increasingly rely on real-time video analytics to assist surgeons during complex procedures. In CLOUDSKIN, the National Center for Tumor Diseases (NCT) requires ingesting high-definition endoscopic video streams, performing AI-driven inference for tasks such as instrument detection and surgical phase recognition, and storing video data durably for later analysis and model training. These operations are highly latency-sensitive: any delay in processing frames can compromise surgical decision-making. To address these challenges, we developed a Proof of Concept (PoC) within the CLOUDSKIN project that leverages Pravega as the backbone for elastic and durable video stream storage, combined with GStreamer for ingestion and containerized AI models orchestrated across the Cloud–Edge continuum (see D2.3). This architecture demonstrated strong performance and simplified the development and deployment of NCT AI models for real-time inference, validating the feasibility of streaming-based approaches for surgical workflows.

However, healthcare applications impose stringent requirements on *data security and confidentiality*. Surgical video streams often contain sensitive patient information, and compliance with regulations such as GDPR and HIPAA demands robust protection of data in use, at rest, and in transit. While the initial PoC focused on performance and elasticity, it did not address confidentiality guarantees against privileged insiders or untrusted infrastructure. To overcome this limitation, we integrated SCONE, a confidential container technology that leverages Trusted Execution Envi-

ronments (TEEs) to provide hardware-enforced isolation and attestation. SCONE enables existing containerized applications to run inside secure enclaves without code modifications, ensuring that sensitive data and AI models remain protected even in multi-tenant or cloud environments.

This section evaluates the performance impact of "sconifying" Pravega, i.e. running its server-side components inside SCONE-enabled confidential containers—compared to non-confidential deployments.[4] Specifically, we measure the I/O performance overhead introduced by TEEs under different streaming workloads, using OpenMessaging Benchmark (OMB) to quantify latency and throughput across varying event sizes and production rates. Our goal is to determine whether the confidentiality guarantees provided by SCONE can be achieved without compromising the real-time requirements of surgical video analytics. The results presented here offer insights into the trade-offs between security and performance in latency-critical healthcare scenarios, guiding future designs for secure, scalable CAS platforms.

Methodology

This experiment relies on benchmarking different streaming workloads on two Pravega server instances with different levels of security and compares the output metrics to have a better understanding of the impact of secure computations on distributed streaming platforms. The following are the necessary elements:

- **Secured Pravega:** also known as Sconified Pravega. It is a Pravega server instance running in an Intel SGX TEE by using Scone runtime and network shield features.
- **Standard Pravega:** it is a Pravega server instance without any special security configuration.
- **Benchmark Client:** Standard OpenMessaging Benchmark (OMB) [24] client, which is a benchmarking framework designed to evaluate the performance of distributed messaging systems. OMB provides a standardized way to compare event streaming systems by measuring key metrics such as throughput, latency, and resource utilization under various workload scenarios.

Figure 9 shows how these elements interact with each other. All of them are running in separate Docker containers. Both Pravega instances are running on the same server, but not at the same time, when running the test case, the correct instance was turning on and at the end turning off. Sconified Pravega is running in a TEE (Intel SGX) and using a Network Shield, which provides encryption and an application-level firewall that denies untrusted connections.

As mentioned above, the Benchmark client uses different configurations to produce different streaming workloads. The following are the configuration options, permuting these configurations produced the 12 test cases considered in this experiment:

- **Production rate:** 100, 1000 and 10000 events per second.
- **Event size:** 100B, 1KB, 10KB and 100KB. Payload file sizes are generated from the default payload files in the OpenMessaging Benchmark tool.

The OpenMessaging Benchmark number of messages or events sent varies depending on the production rate, but is not always the same even using the same configuration. The following are the default configuration for all test cases:

- 5 minutes benchmarking and 1 minute warmup.
- No compression or encryption.
- 1 topic and 1 partition per topic.
- Retention policy of 10 MB.

[4] The experiments in this section are a continuation of our previous work sconifying the Pravega client [23].

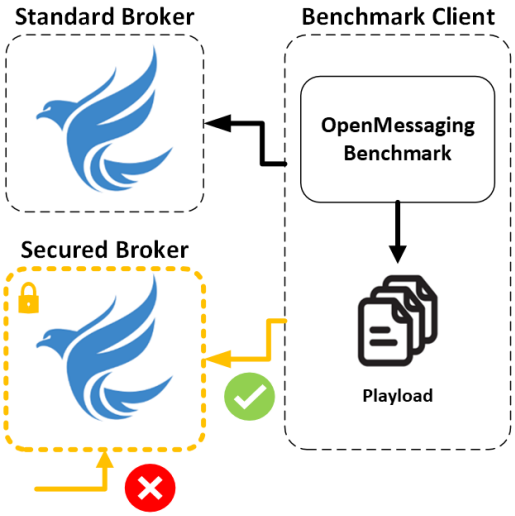


Figure 9: Experiment workflow: A OpenMessaging Benchmark client sending messages to two Pravega instances, one secured and one standard. Pravega instances are running on the same server but not at the same time meanwhile the Benchmark client is running on their own server.

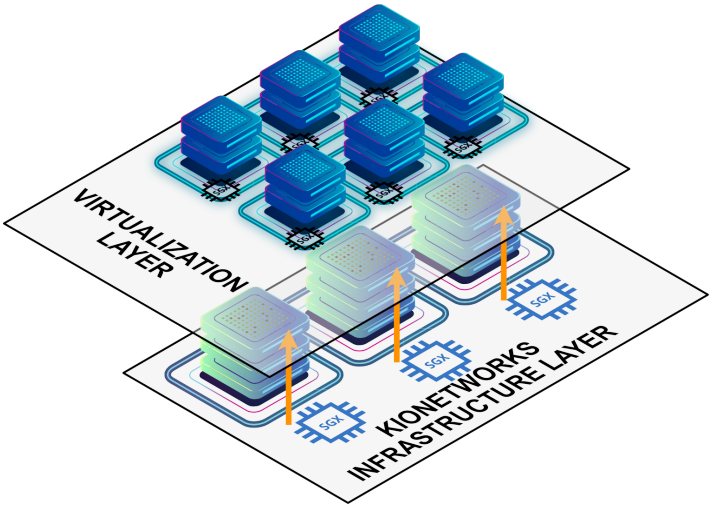


Figure 10: KIO infrastructure for executing the Sconified Pravega experiments.

Metrics

The following is a list and brief description of the metrics considered in this experiment. Most of these metrics are derived from the OpenMessaging Benchmark tool, which generates a JSON-formatted output file containing various measured and computed variables collected during the benchmarking process.

- **50th Percentile Write Latency:** Represents the median latency experienced by the publisher when writing messages.
- **75th Percentile Write Latency:** Indicates the latency below which 75% of the write operations fall. This metric helps identify moderate tail latencies.
- **95th Percentile Write Latency:** Reflects the latency threshold below which 95% of the write operations occur, highlighting higher-end tail latencies.

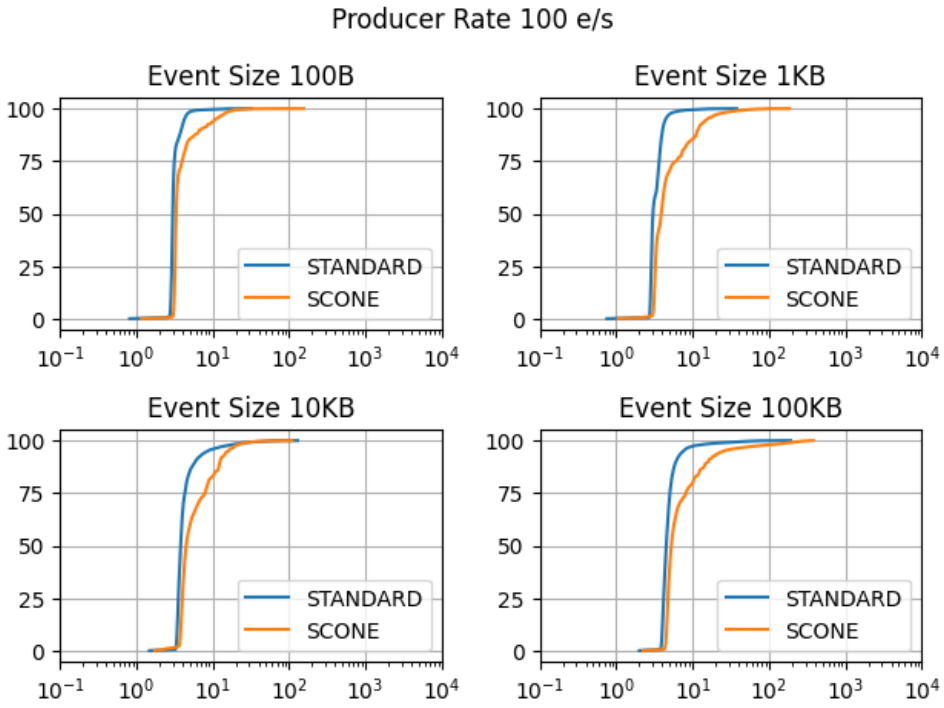


Figure 11: CDF plots show write latencies (logarithmic scale on the X-axis) against percentiles (Y-axis) for test cases with a producer rate of 100 events per second across varying event sizes. The results indicate that workloads on the standard Pravega instance achieve lower latencies compared to those on the secured instance. Notably, the 90th percentile latency for the standard Pravega is approximately 62% lower than that of the secured configuration.

- **99th Percentile Write Latency:** Represents the latency below which 99% of the write operations fall, useful for understanding worst-case performance scenarios.
- **Latency Distribution (CDF):** A cumulative distribution function (CDF) of all recorded write latencies, providing a comprehensive view of latency behavior across the entire benchmark run.
- **Throughput (MB/s):** Defined as the total volume of data successfully published divided by the total execution time, excluding the warm-up period. Throughput is measured in megabytes per second (MB/s) and is computed using the total number of bytes published (from a modified version of the OpenMessaging Benchmark) and the elapsed time, determined by timestamps recorded before and after the benchmark execution.

Test Environment

The current virtualization environment consists of a VMware vSphere 8 cluster comprising three physical hosts. Each host operates under VMware ESXi 8 and participates in a unified resource pool managed through vCenter. The cluster is configured with EVC CPU Mode: Intel “Sapphire Rapids” Generation, ensuring full compatibility and seamless vMotion operations across all hosts. EVC Graphics Mode (vSGA) is also enabled, providing virtualized GPU support for workloads that require graphical acceleration.

- **Host Configuration:** Each host in the cluster is based on the Cisco UCSC-C240-M7SX platform, delivering enterprise-grade performance and reliability. The nodes are equipped with Intel Xeon Platinum 8458P processors, offering 176 logical processors per host. Each server has 8 physical NICs, providing redundancy and high network throughput for virtualized workloads.

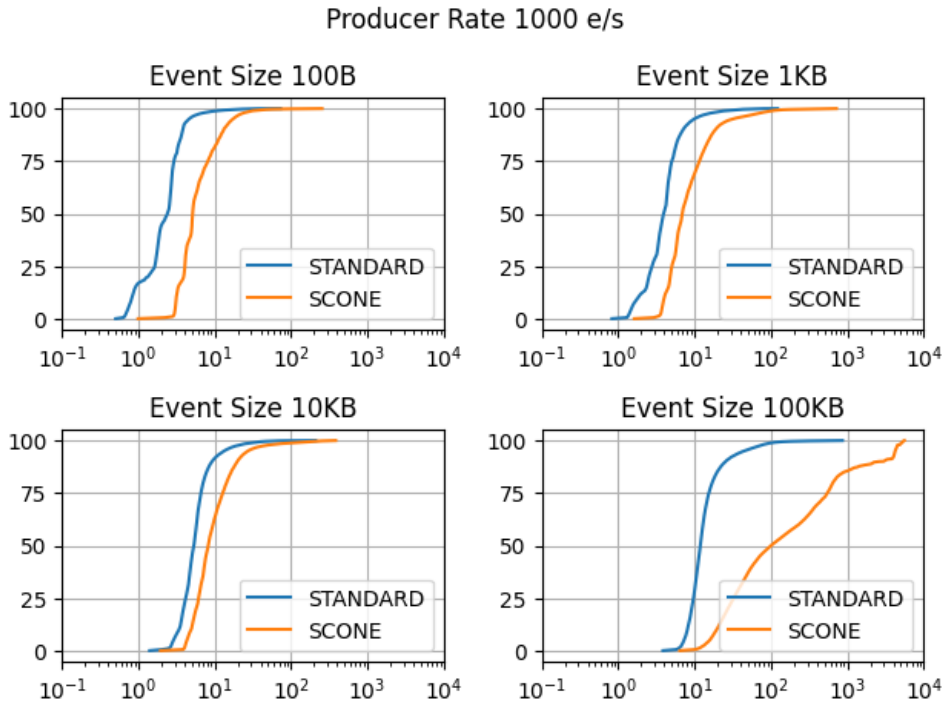


Figure 12: CDF plots show write latencies (logarithmic scale on the X-axis) against percentiles (Y-axis) for test cases with a producer rate of 1,000 events per second across varying event sizes. The results indicate that workloads on the standard Pravega instance achieve lower latencies compared to those on the secured instance. Notably, the 90th percentile latency for the standard Pravega is approximately 72% lower than that of the secured configuration. Additionally, it can be seen on event size of 100KB, the secured Pravega begins to exhibit performance degradation due to the high volume of data transmitted per second.

All nodes are in a Connected state within vCenter, ensuring full participation in cluster-level resource management and HA/DRS operations.

- **Security and SGX Capabilities:** The Intel Xeon Platinum 8458P CPUs feature Intel Software Guard Extensions (SGX), enabling secure enclaves for sensitive workloads. SGX provides hardware-based memory encryption that isolates specific application code and data in memory, protecting it from unauthorized access or modification even if the system software is compromised. This capability is especially beneficial for secure data processing, confidential computing, and regulated workloads where data privacy is critical.

  On top of the vSphere cluster, several virtual machines are configured with SGX passthrough to directly leverage the underlying CPU enclave capabilities. These VMs operate within isolated network segments, ensuring strict separation from general-purpose traffic and minimizing the attack surface. Connectivity to external environments is tightly controlled, allowing only predefined and secured communication channels. This design enables secure data processing workloads —such as cryptographic operations, confidential analytics, or trusted execution environments— to run in compliance with high-security and privacy standards.

- **Resource Summary**
  - *CPU resources*: Total cluster capacity of 712.8 GHz for workloads.
  - *Memory resources*: Total of 3,071 GB for scaling virtual machines or deploying new workloads.

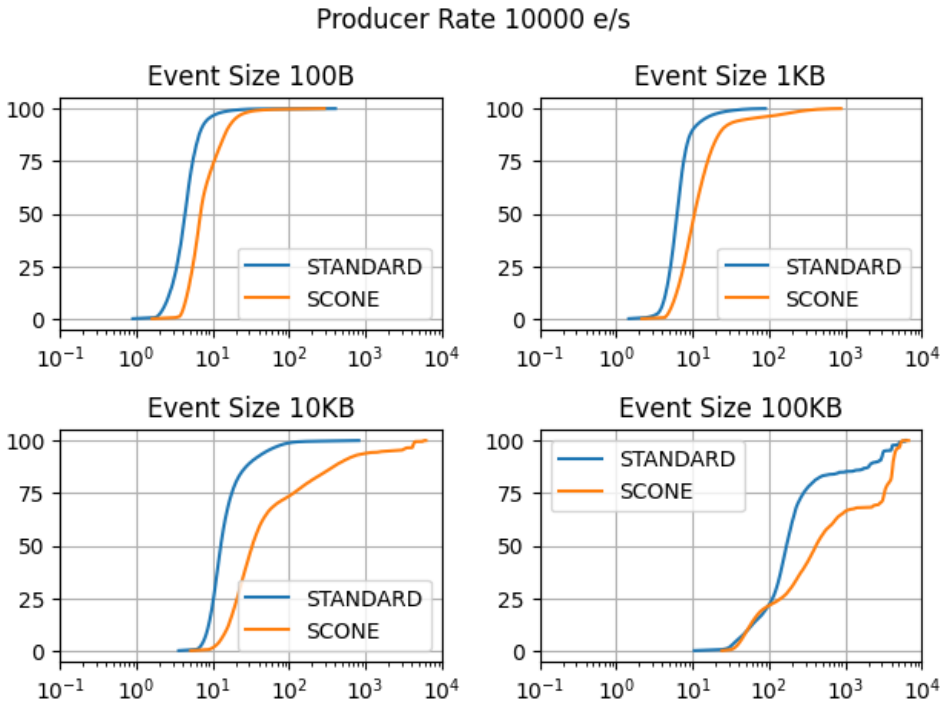


Figure 13: CDF plots show write latencies (logarithmic scale on the X-axis) against percentiles (Y-axis) for test cases with a producer rate of 10,000 events per second across varying event sizes. The results indicate that workloads on the standard Pravega instance consistently achieve lower latencies compared to those on the secured instance. Notably, the 90th percentile latency for the standard Pravega is approximately 63% lower than that of the secured configuration. Additionally, it can be observed that when the event size of 100KB, the system begins to exhibit performance degradation due to the high volume of data transmitted per second.

– *Storage resources*: Shared storage capacity of 21,739 GB providing ample space for VM data, snapshots, and future growth.

Benchmarking Results

A benchmark was set using 12 use cases, along with their respective configurations and metrics, on the two previously mentioned Pravega server instances (standard and secured with SCONE). Overall, the standard Pravega instance demonstrates lower write latencies compared to the secured one. However, it is essential to evaluate this latency degradation in different scenarios and determine under which conditions it remains acceptable.

Figure 11 illustrates the results for messages transmitted at a rate of 100 events per second, representing a relatively small data load. The plots show that workloads on the standard Pravega instance exhibit slightly lower write latency compared to those on the secured instance. Specifically, the 95th percentile latency for the standard Pravega is approximately 62% lower than that of the secured Pravega. This difference is substantial, yet the secured configuration remains viable for use cases requiring low latency, such as NCT.

Figure 12 continues to demonstrate the performance advantage of the standard Pravega instance, with an even greater disparity. At a message rate of 1,000 events per second, the 95th percentile latency for the standard Pravega is roughly 72% lower than that of the secured instance. Furthermore, when the event size reaches 100KB, the secured Pravega instance begins to exhibit noticeable performance degradation—a factor that can be critical during system design.

Finally, Figure 13 shows the results for a producer rate of 10,000 events per second, and it reveals

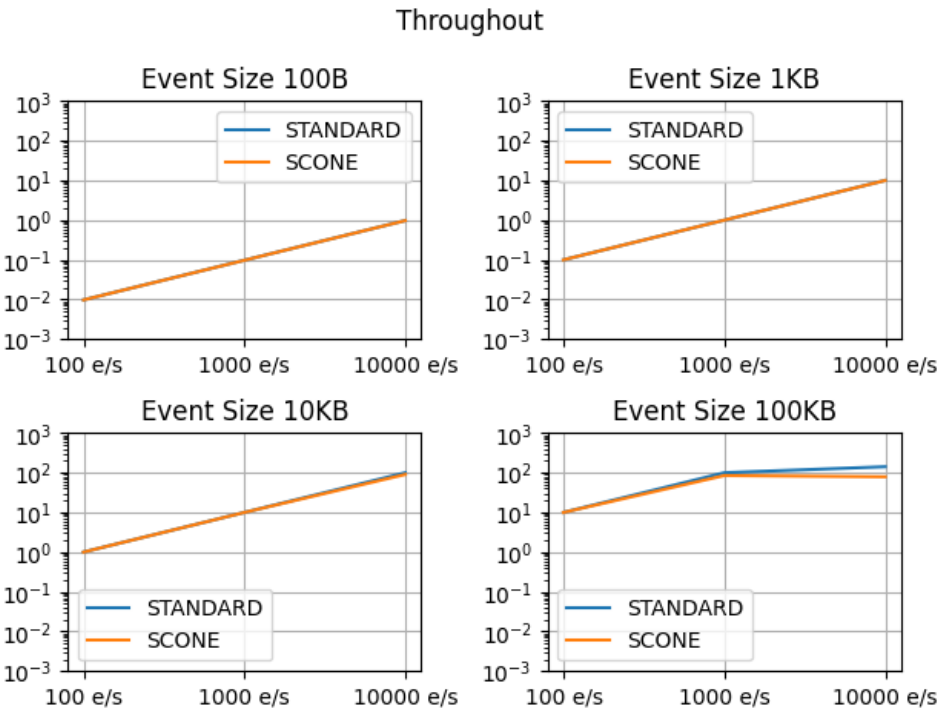


Figure 14: The throughout during benchmarking is almost the same for secured and standard Pravega instances except for the last use case (event size 100KB and 10,000e/s) supporting the performance degradation idea.

a similar trend, though the gap narrows compared to the previous scenario. Here, the standard Pravega instance achieves approximately 63% lower latency at the 95th percentile than the secured instance. Additionally, performance degradation for messages with an event size of 100 KB becomes more pronounced and affects both the secured and standard Pravega configurations.

Despite the observed differences in latency, throughput remains nearly identical in all use cases without performance degradation, as shown in Figure 14. This indicates that the secured Pravega instance can process the same volume of work per unit of time as the standard one under these conditions. However, in the final use case (event size of 100KB and a production rate of 10,000 events per second), throughput decreases for both Pravega instances, supporting the notion that system performance degrades when handling high volumes of data transmission per second.

The benchmark results indicate that while the standard Pravega instances consistently achieve lower latency compared to the secured ones, both configurations maintain equivalent throughput. This suggests that the security mechanisms integrated into the secured Pravega instance introduce additional processing overhead, which impacts response time without reducing overall data-handling capacity. Consequently, the performance trade-off primarily affects latency-sensitive applications, whereas throughput-driven workloads remain largely unaffected. These findings highlight that security can be incorporated without compromising scalability, although its impact on real-time responsiveness must be carefully considered in latency-critical scenarios.

6.3 Summary of achieved KPIs

In this section we have presented empirical evidence, in the form of reproducible experiments, that CLOUDSKIN's execution layer achieves the project's KPIs. In particular,Table 5 revisits the high-level KPI table presented in Table 2 and presents them in-relation to the results shown in this section.

In summary, throughout the different deliverables in WP4 we have demonstrated that C-Cells support execution of complex applications across the cloud-edge continuum at near-native speed and additional confidential C-Cell execution comes at moderate runtime overhead. C-Cells low-overhead

KPI	Description	WP4 Contribution
KPI1	Performance parity between instrumented and non-instrumented workloads.	C-Cells execute MPI and OpenMP applications at native speed (see Figure 7).
KPI2	Transparent migration	C-Cell migration enables high-locality (see Figure 7) and high-elasticity (Figure 8) without overheads.
KPI3	Heterogeneous execution	C-Cells can execute in ephemeral spot VMs (see Figure 6c), regular X86 servers, and SGX enclaves (see Figure 11.
KPI4	Trusted execution with minimal overhead.	Confidential C-Cells can process streaming messages with limited overhead (see Figure 12).
KPI5	Support for complex applications.	Distributed C-Cells can execute the LAMMPS molecule dynamic's simulator (see Figure 7).
KPI6	Dynamic elastic scaling.	Vertical scaling of C-Cells executing OpenMP jobs (see Figure 8).
KPI8	Real-world testbed deployment.	Most experiments in this section are deployed on KIO's testbed.

Table 5: Demonstration of KPIs based on experimental results

snapshot-restore mechanisms enable efficient live migration and vertical scaling, unlocking more efficiency, utilization, and performance through C-Cell-aware scheduling and orchestration policies.

7 Use Case Integration

D5.4 covers the integration between technical work-packages and all industrial use-cases. D5.4 describes each use-case in detail, provides a high-level overview of its interacting components, and includes an in-depth evaluation. In this deliverable we provide a brief overview of the use-cases that interact more closely with WP4, and defer the reader to D5.4 for further details.

7.1 Metabolomics

In the metabolomics use-case, we build an image processing pipeline that performs an image similarity search on private medical data without leaking the database that the image is compared against, which also contains private images. We achieve this through a combination of confidential C-Cells, to guarantee the confidentiality of the processed data, and an OpenMP job using a pool of parallel C-Cells to adapt, elastically, to the available resources in the underlying testbed.

The first part of the pipeline uses a confidential C-Cell to ingest a dataset of private medical images and generate intermediate embeddings. These embeddings, as argued in D5.4, provide enough anonymization to protect the confidentiality of the data owners. Once anonymized, this data can be ingested by a pool of parallel C-Cells operating on shared memory to maximize throughput. This pool is, under-the-hood, an OpenMP job with elastic scaling, where each C-Cell executes with thread semantics and performs a k-nearest-neighbour search on a batch of images.

The whole pipeline is deployed on the KIO cloud-edge testbed in a server with Intel SGX support. The communication between different stages in the pipeline happens through object storage, using MinIO. The elastic scaling of the OpenMP job is done based on utilization metrics of the host system. Important metrics of the pipeline execution are published to a Grafana monitor for visualization.

7.2 Computer-assisted surgery (CAS)

Computer-assisted surgery is one such case in which not only is the data sensitive, but also the integrity of the execution is of the highest importance. By running it inside TEE, both can be guaranteed. At the moment, we successfully leveraged SCONE's lift-and-shift approach to enable confidential computing in the Surgery use case in two frameworks: ROS (Robot Operating System) and GStreamer (multimedia stream framework).

The ROS framework is a middleware suite designed to work with robotic infrastructure. In computer-assisted surgery, the inference output could be piped to a robotic hand. However, since the inference is still performed at the edge node, it leaves it in an exposed position where attacks may be launched. Therefore, we opt to make the inference process trustworthy by deploying and running it in an Intel SGX enclave. The model is offline and embedded in a docker image. We built the ROS node, capable of doing confidential inference. The result will then be forwarded to the subscribers, such as image output or robotic hands.

For completeness, we also protected the inference when using the GStreamer framework. Gstreamer is a pipeline-based multimedia framework that allows us to manipulate streaming data. In the Surgery use case, it takes a video source as an input, performs inference to detect liver segmentation, and produces a video output. Complementary to our previous implementation, which protected only the inference part, here we ensure the entire process runs within a TEE. Using SCONE, both the main application (`gst`) and the inference plugin will be loaded into the same memory region, thus the same level of protection.

8 Open-Source Releases and Maintenance Plan

8.1 GRANNY

GRANNY is released as an extension to the Faasm serverless runtime, and upstreamed to the parent repository with version `v0.27.0`[5]. We include a screenshot of the Faasm release in **??**.

The Faasm serverless runtime has numerous contributors, almost a thousand stars on GitHub, and has been actively maintained for the last seven years. By integrating GRANNY into a larger open-source project, we increase its potential outreach and ease its maintenance. GRANNY's main contribution to Faasm is contained in a library called `faabric` that manages the communication between C-Cells, and their shared memory accesses.

Newer CLOUDSKIN features have also been upstreamed to Faasm. In the interest of dissemination, we make a fork of the Faasm runtime under the `cloudskin-eu` organization. We fork at Faasm's version `v0.33.0`, and include a screenshot of the repository's read-me file in **??**.

8.2 Sconified Pravega

The integration of Sconified Pravega into the broader ecosystem is designed with long-term usability in mind. Our objective is to ensure that the enhancements introduced through this effort remain accessible to the community and continue to align with evolving security and performance requirements. Although a formal roadmap has not been defined at this stage, the project is structured to encourage future contributions, updates, and community-driven maintenance. Comprehensive documentation and integration guidelines are available in its GitHub repository[6], enabling developers and organizations to adopt and extend the solution as needed. By embracing open-source principles, we expect this work to serve as a strong foundation for ongoing innovation and collaboration within the Pravega community.

[5] `https://github.com/faasm/faasm`

[6] `https://github.com/cloudskin-eu/scone-pravega/`

9 Conclusions

In this deliverable, we have presented the final design and reference implementation of CLOUDSKIN' execution layer. At the core of this execution layer lie C-Cells, a universal lightweight execution sandbox based on WebAssembly. C-Cells can execute unmodified code from a variety of programming languages, in heterogeneous hardware like Intel SGX enclaves, with minimal performance overhead. Groups of C-Cells can be orchestrated to send messages to each other, or operate on regions of shared memory. We use these features of C-Cells to support execution of more complex applications, including scientific applications using MPI and OpenMP, and streaming applications with enhanced privacy guarantees.

This deliverable acts as the colophon of WP4, and complements the previous deliverables D4.1 and D4.2 where we introduced the initial C-Cell design, and the initial evaluation of message passing and shared memory, respectively. This deliverable presents the final reference implementation of all components, additional evaluation results, and the open-source maintenance plan.

References

[1] 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.

[2] R. Gracia-Tinedo, F. Junqueira, T. Kaitchuck, and S. Joshi, "Pravega: A tiered storage system for data streams," in Proceedings of the 24th International Middleware Conference, Middleware '23, (New York, NY, USA), p. 165–177, Association for Computing Machinery, 2023.

[3] 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, Nov. 2016.

[4] 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.

[5] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," Computer Physics Communications, vol. 271, p. 108171, 2022.

[6] 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," ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2017.

[7] CRIU, "Checkpoint-Restore in Userspace." `https://www.criu.org/Main_Page`, 2021.

[8] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Veriwasm: Sfi safety for native-compiled wasm," NDSS'21.

[9] FFmpeg Contributors, "Webassembly lld port." `https://lld.llvm.org/WebAssembly.htmlimports`, 2022.

[10] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita, D. Tullsen, and D. Stefan, "Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi," in Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, 2023.

[11] MPI, "MPI Forum." `https://www.mpi-forum.org/`, 2022.

[12] LLVM Project, "LLVM OpenMP Runtime Library Interface." `https://openmp.llvm.org/doxygen/index.html`, 2024.

[13] WASI, "WebASsembly System Interface." `https://wasi.dev/`, 2024.

[14] WebAssembly, "WASI libc implementation for WebAssembly." `https://github.com/WebAssembly/wasi-libc`, 2024.

[15] ByteCode Alliance, "WebAssembly Micro Runtime." `https://bytecodealliance.github.io/wamr.dev/`, 2024.

[16] WebAssembly, "WASI Threads." https://github.com/WebAssembly/wasi-threads, 2024.

[17] Microsoft, "Azure Batch." https://azure.microsoft.com/en-us/services/batch/, 2021.

[18] SchedMD, "Slurm Workload Manager." https://slurm.schedmd.com/overview.html, 2024.

[19] OpenMPI, "OpenMPI: Open Source High Performance Computing." https://www.open-mpi.org/, 2021.

[20] LLVM Project, "LLVM/OpenMP documentation." https://openmp.llvm.org/, 2022.

[21] Microsoft, "Azure Virtual Machines." https://docs.microsoft.com/en-us/azure/virtual-machines/dv2-dsv2-series, 2021.

[22] ParResKernels Team, "Parallel Research Kernels." https://github.com/ParRes/Kernels, 2021.

[23] A. Cueva Mora, K. Jayasena, R. Krahn, E. Chirivella-Perez, and C. Fetzer, "Understanding the latency-security tradeoff: Tee-based confidential computing for streaming workloads," in 2025 IEEE 33rd International Conference on Network Protocols (ICNP), pp. 1–6, 2025.

[24] "The openmessaging benchmark framework." https://openmessaging.cloud/docs/benchmarks/, 2018. Accessed: Aug 11, 2025.