CloudSkin

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

# CloudSkin

(grant agreement No 101092646)

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

## D4.1 Initial prototype for Cloud-edge cells

Due date of deliverable: 30-06-2023
Actual submission date: 30-06-2023

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

# Summary of the document

| | |
|---|---|
| **Document Type** | Other |
| **Dissemination level** | Public |
| **State** | v1.0 |
| **Number of pages** | 21 |
| **WP/Task related to this document** | WP4 / T4.1 |
| **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), André Martin (TUD) |
| **Partner(s) Contributing** | IMP, TUD |
| **Document ID** | CloudSkin_D4.1_Public.pdf |
| **Abstract** | Early software release of Cloud-edge cells virtualisation technology for different environments (C/C++, Python) as well as partial TEE support to ensure confidentiality and integrity of the involved components. |
| **Keywords** | WebAssembly, isolation, wasm, edge, trusted execution environments, TEEs |

# History of changes

| Version | Date | Author | Summary of changes |
|---|---|---|---|
| 0.1 | 12-05-2023 | Carlos Segarra, Peter Pietzuch, André Martin (TUD) | First draft. |
| 0.2 | 12-06-2023 | Carlos Segarra, Peter Pietzuch | Address minor comments prior to F2F meeting in Dresden. |
| 0.3 | 15-06-2023 | Carlos Segarra, Peter Pietzuch | Update future work section after integration discussions during F2F meeting in Dresden. |
| 0.4 | 29-06-2023 | Carlos Segarra, Peter Pietzuch, Huanzhou Zhu, Guo Li | Update evaluation section. |
| 1.0 | 30-06-2023 | Carlos Segarra, Peter Pietzuch, Huanzhou Zhu, Guo Li, André Martin (TUD) | 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 |
| **IoT** | Internet of Things |
| **JIT** | Just-in-Time |
| **JS** | JavaScript |
| **JVM** | Java Virtual Machine |
| **MEE** | Memory Encryption Engine |
| **MPI** | Message Passing Interface |
| **OMP** | Open MP |
| **OS** | Operating System |
| **PRM** | Processor Reserved memory |
| **SDK** | Software Development Kit |
| **SFI** | Software Fault Isolation |
| **SGX** | Software Guard eXtensions |
| **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 |
| **WAST** | WebAssembly Text Representation |

## 1 Executive summary

This deliverable presents the first design and implementation of Cloud-Edge Cells (C-Cells), the execution unit of the CLOUDSKIN cloud-edge continuum. We first introduce the requirements for C-Cells and the background concepts we build on: lightwieght WebAssembly-based isolation as implemented in Faasm, and transparent confidential execution as implemented in SCONE. Then, we describe the first C-Cell prototype building on Faasm's Faaslets, and what components we need to add in the distributed runtime for CLOUDSKIN. In the evaluation we show how C-Cells can be used to execute multi-threading and multi-processing applications with very limited overhead whilst enabling necessary features for the cloud-edge continuum like live migration.

## 2 Introduction

CLOUDSKIN's goal is to build an enviornment for the Cloud-edge continuum. This work package, WP4, is responsible for the design and implementation of a key piece in the Cloud-edge continuum: its execution unit. We have named the execution units of the cloud-edge continuum Cloud-Edge Cells (or C-Cells, for short). This deliverable describes the first prototype implementation of C-Cells.

The characteristics of the Cloud-edge continuum impose a set of requirements on the design of C-Cells. The Cloud-edge continuum requires adaptive virtualization powered by, and enabling, AI workloads. Let us breakdown these requirements one by one, and compare them with existing execution units like linux processes [1], containers [2], virtual machines (VMs) [3], or language specific abstractions like JavaScript isolates (as used in Cloudflare [4]) or WebAssembly [5].

Cloud-edge means that C-Cells's design must be independent of the hardware environment in which they are deployed. In particular, C-Cells must support being executed in a powerful server in the cloud, and a low-end device in the edge. Each with, potentially, different instruction sets and capabilities. In short, **C-Cells must be hardware, and instruction set, agnostic**. Language-independent isolation units like processes, containers, and VMs require non-trivial changes to be executed in different architectures. For example, different docker images need to be built for different architectures [6]. Managed language runtimes achieve hardware independence by firstly deploying hardware-specific versions of the runtime, and then transparently executing the same code (e.g. Java bytecode in the JVM [7] or WebAssembly runtimes).

Continuum means that C-Cells must support being moved back-and-forth between servers in the cloud and devices in the edge. For example, a C-Cell may be moved from the edge to the cloud to improve performance or reduce the load in the edge device. Similarly, a C-Cell may be moved from the cloud to the edge to take computation closer to data in a data-shipping fashion. In short, **C-Cells must support transparent live migration**. Process [8], container [9], and VM [10] migration exists, and can be used to checkpoint the execution state, stop it, and resume it at a later point in time, potentially in a different server. Live migration of managed language runtimes is experimental [11] and not as common-place.

Adaptive virtualization means that **C-Cells must support different software and hardware-based virtualisation techniques for performance and isolation**. For example, when processing confidential data, C-Cells should be executed with stronger isolation guarantees, for example those provided by Trusted Execution Environments (TEEs). Processes, containers, and VMs support transparent lift-and-shift [12] approaches to execute unmodified binaries, container images, and VM images in TEEs like SGX enclaves [13] or confidential VMs [14, 15]. Managed language runtimes support running the runtime inside a TEE, and provide mechanisms to transparently get the bytecode inside the TEE [16, 17].

AI-enabling means that **C-Cells must support executing AI workloads** both in the training and inference phases. Supporting AI-workloads not only requires C-Cells to be highly performant, but also language-independent and accelerator-aware (i.e. GPU [18], or TPU [19]). This requirement rules out the usage of managed language runtimes like Python or Java, as C-Cells require greater access to hardware resources, and language independence to accomodate for the new AI models being developed and yet to come. Even though AI workflows can be *written* in Python, the underlying execution environment are usually C/C++ kernels executing in GPUs.

In Table 1 we list some of the relevant existing execution units in the cloud and in the edge, together with CLOUDSKIN's requirements to enable AI workloads in the cloud-edge continuum. In the rest of this document, we will describe the first C-Cell implementation (C-Cell D4.1). First, in §3 we introduce the previous work that we build on to design C-Cells. Second, in §4 we provide the first prototype of C-Cells. We then evaluate the initial implementation with realistic applications in §5. Lastly, in §6 we outline how we plan on improving upon this initial prototype.

## 3 Background

In this section we cover the background concepts and related foudnational work for the design and implementation of Cloud-Edge Cells.

| Execution Unit | Isolation Mechanism | Executable | Hardware Agnostic | Live Migration | Adaptive Virtualisation | Language Independence |
|---|---|---|---|---|---|---|
| Processes | Page Tables | ELF Binaries | ✗ | ✷ [8] | ✷ [12] | ✓ |
| Containers | Processes + NS | Container Images | ✗ | ✓ | ✷ [20] | ✓ |
| VMs | HW Virtualisation | VM Images | ✗ | ✓ | ✓ | ✓ |
| JS Isolates | SFI [21] | JS Code | ✓ | ✷ [22] | ✗ | ✓ |
| WASM Modules | SFI [21] | WASM Bytecode | ✓ | ✗ | ✷ [17] | ✓ |
| C-Cells D4.1 | Faaslets [23] | WASM Bytecode | ✓ | ✓ | ✗ | ✓ |
| **C-Cells** | Adaptive | WASM Bytecode | ✓ | ✓ | ✓ | ✓ |

Table 1: Comparison between existing execution units against CLOUDSKIN requirements for the cloud-edge continuum. We use a ✷ to indicate features that are not supported by default, but can be enabled using research or experimental prototypes. We also position the initial C-Cells implementation presented in this deliverable (D4.1) with respect to the ideal C-Cells implementation.

### 3.1 WebAssembly

WebAssembly [5] is a binary instruction format designed with security and portability in mind. WebAssembly was originally created to provide software-fault-isolation (SFI) [21] when executing untrusted code in a browser. Since then, its potential as a general-purpose SFI mechanism has been exploited in serverless [23], edge computing [24], smart contracts [25], and even as an alternative back-end for docker containers [26].

WebAssembly offers strong memory safety guarantees by constraining memory access to a single linear byte array, referenced with offsets from zero. This enables efficient bounds checking at both compile time and runtime, with runtime checks backed by traps. The correctness of the bound checks has been formally verified [27]. Other pointers, like function pointers, are expresses as offsets in tables, like function tables. The security guarantees of WebAssembly are well established in existing literature, which covers formal verification [28], taint tracking [29], and dynamic analysis [30].

WebAssembly is exposed as a back-end compilation target in LLVM, offering mature support for languages with an LLVM front-end such as C, C++, C#, Go and Rust [31]. Integration with dynamic programming languages is mostly experimental, and toolchains exist for Typescript [32] and Python [33]. Experimental toolchains also exist for Swift [34] and Java [35].

#### 3.1.1 Format

```c
int myFun(int x) {
  return x + 2;
}
```

```
(module
  (table 0 anyfunc)
  (memory $0 1)
  (export 'memory' (memory $0))
  (export 'myFun' (func $myFun))
  (func $myFun (; 0 ;) \
      (param $0 i32) (result i32)
    (i32.add
      (get_local $0)
      (i32.const 2)
    )
  )
)
```

Figure 1: A C source function and its representation in WebAssembly text format, WAST.

Figure 1 shows two listings, one of a simple C function, and the other of the text representation of its corresponding WebAssembly, using the WebAssembly Text Representation (WAST) [36]. WAST

is a human-readable format that supports two-way translation with WebAssembly binaries via the WebAssembly Binary Toolkit [37].

In the WAST listing we see a top-level `module`, which forms the basis of all WebAssembly binaries. Interaction between WebAssembly modules is not yet officially supported, although multi-module WebAssembly is under development [38], and there is also partial support for dynamic linking [39]. Each WebAssembly module defines a function table, which is used to support indirect function calls via the `call_indirect` instruction. Indirect calls are used to support function pointers in source languages that use them, and when dynamically linking two WebAssembly modules. The `memory` keyword shows the definition of the WebAssembly linear memory for this module, which can also include limits on the size of the memory. If no limit is specified, this limit will be 4 GB, corresponding to the maximum value that can be represented using a 32-bit integer. The `export` command specifies which parts of this module are accessible to the WebAssembly runtime and other WebAssembly modules.

WebAssembly uses a stack machine, with each function defining a set of local variables that can be moved onto and off the stack and manipulated in place. WebAssembly functions can also load and manipulate values held in the global linear memory array using integer offsets. WebAssembly enforces a structured control flow, so does not support arbitrary jumps, hence cannot compile certain language features, such as `goto` statements or virtual method tables in C/C++ [40].

### 3.1.2 Linear memory

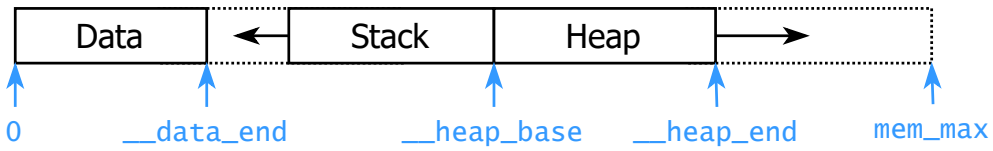



Figure 2: WebAssembly linear memory layout. All addresses are expressed as offsets from zero. Stack memory grows downwards towards `__data_end`, while heap memory grows upwards towards its maximum offset at `mem_max`.

Memory in WebAssembly is handled via zero-indexed linear byte arrays. All memory accesses are expressed as offsets from zero, so bounds can be checked by verifying they are greater than zero, and less than the maximum memory size. WebAssembly code has access to several memory-related operations including `grow-memory` and `shrink-memory`, but cannot allocate disjoint address ranges. The WebAssembly runtime is responsible for maintaining the memory that backs this linear address space and converting the offsets into pointers to it. The runtime must halt execution if an invalid memory access is made. The conformance of a runtime to these rules can be tested using the WebAssembly specification tests [36].

Figure 2 shows the layout of the WebAssembly linear memory, which contains three regions, static data, a stack, and a heap [41]. The static data is placed at the beginning to make sure constants have low addresses, reducing code size. The WebAssembly stack grows downwards towards the end of the data section, indicated by the constant `__data_end`. The maximum stack size is the difference between `__data_end` and `__heap_base`. The heap grows upwards towards the maximum memory size, which can be set by the module, or left to the default maximum which is roughly four gigabytes in 32-bit WebAssembly.

WebAssembly linear memory is most commonly implemented using a linear byte array placed at the bottom of a 8 GB virtual memory region. 4 GB is the largest possible linear memory array for a 32-bit WebAssembly module, and also the largest offset that can be indicated with a 32-bit integer, so a pointer+offset can point as far as 8 GB. By pre-allocating virtual memory, the runtime can rely on the OS's memory manager to cause a page fault if the application makes memory accesses outside of this region. This avoids the runtime needing to add its own bounds checks on each memory access, which would otherwise severely affect performance. A standard x86 machine can support a

virtual memory address space of $2^{48}$ bytes, so it has room for more than two billion such mappings. However, this technique will not be possible with 64-bit WebAssembly, whose linear address space will exceed the available virtual memory address space of the machine.

#### 3.1.3 Toolchains and runtimes

There are two steps in order to execute arbitrary code as a WebAssembly module. First, code needs to be cross-compiled to WebAssembly bytecode using a compilation toolchain. Second, the WebAssembly bytecode must be executed by a specialised WebAssembly runtime. We breakdown these two aspects separately.

Although any compiler toolchain may choose to implement a back-end for WebAssembly, the most developed and widely used are LLVM [31] and CraneLift [42]. LLVM also provides front-ends for a range of languages including C and C++ through Clang [43], Rust [44], and Go [45]. To execute an application compiled to WebAssembly, all library dependencies must also be compiled to WebAssembly. Standard libraries for all supported languages such as libc and libc++ are provided by open-source projects like WASI [46]. This language-independence and flexibility make LLVM an appealing compiler toolchain choice.

There are now many WebAssembly runtimes, each focusing on different execution environments, performance goals, and breadth of features. The most significant difference between WebAssembly runtimes is whether they interpret the WebAssembly at runtime, compile it ahead of time (AOT) into machine code, or compile it at runtime Just-in-Time (JIT). AOT runtimes offer better performance as the code generation step can introduce architecture-specific optimisations, but require running this code generation once for each target architecture. Interpreters are slower, but not platform-specific, so are more suitable for execution environments where the architecture is not known ahead of time, such as web browsers. The most popular AOT WebAssembly runtimes are currently: WAVM [47], a general-purpose WebAssembly runtime written in C++ with support for all the most recent WebAssembly features; WAMR [17], written in C and targeting a smaller resource footprint, suitable for IoT, edge and trusted execution environments; and wasmtime [48], another general-purpose WebAssembly runtime written in Rust with support for all current WebAssembly features. Even though WAVM used to have the best performance, recent benchmarks show that it is falling behind as it is no longer maintained [49]. The most popular interpreter runtimes are those implemented in the four major browsers; V8 in Chrome/Chromium [50], SpiderMonkey in Firefox [51], Chakra in Microsoft Edge [52] and JavascriptCore/SquirrelFish in Webkit [53]. We plan on including a benchmark comparison of different WASM runtimes in a future deliverable.

#### 3.1.4 WASI: the WebAssembly system interface

WebAssembly itself does not define a foreign function interface or set of supported syscalls. Such an API would be platform-specific, and hence at odds with the goals of the project to provide platform-independent isolation. Platform-specific APIs for interaction with the execution environment and underlying host will vary between WebAssembly runtimes, but the Bytecode Alliance [54] has created the WebAssembly System Interface (WASI) [46], which standardises a range of POSIX-like system calls. WASI uses a capability-based security model across a suite of POSIX-like system calls [46]. These calls include file I/O, networking, timing and error handling.

### 3.2 Trusted Execution Environments

In this section we cover the background concepts related to security for the design of Cloud-Edge Cells.

#### 3.2.1 Motivation & Overview

With the use of Trusted Execution Environments (TEEs), we aim to provide confidentiality, integrity and freshness guarantees for applications running within a C-Cell. TEEs exist in various flavors, with each of them providing different guarantees. Some TEEs such as Intel SGX [55] were originally designed to place only security-critical functions in so called enclaves, i.e., secure compartments which are only accessible by the function itself, while other approaches such as Intel TDX [56], AMD

SEV [57], etc., provide different abstractions for TEEs that place Micro-VMs in those compartments in order to achieve isolation. As mentioned previously, each of these technologies provides different guarantees regarding confidentiality, integrity and freshness of the data that is processed.

### 3.2.2 Intel SGX

One of the previously mentioned TEEs we envision to use is Intel SGX which was one of the first technologies that combines a processor mode and a set of processor instructions to provide Trusted Execution Environments (TEEs). SGX ensures that processes are isolated from each other by utilizing a dedicated and cryptographically protected memory region (so called enclave) [55]. Enclaves use a contiguous memory region as a block of protected memory borrowed from the Dynamic Random Access Memory (DRAM) as Processor Reserved Memory (PRM). The PRM comprises the Enclave Page Cache (EPC), a set of 4KB memory pages, and enclave meta data. The PRM is neither accessible from other applications nor privileged code such as the operating system or the hypervisor. The Memory Encryption Engine (MEE), part of the processor, encrypts and authenticates data for non-PRM memory and protects EPC pages.

### 3.2.3 Attestation Mechanisms

In the context of SGX, attestation describes the process of proving an enclave's identity to another enclave. During enclave creation, the program data to run the code is first loaded into the enclave memory from the non-protected memory. All subsequent steps that may modify the previously loaded program code are recorded and included in the enclave measurement hash calculation in order to detect potential manipulations. Hence, the measurement hash can later be used to attest an enclave both locally and remotely.

The previously described attestation mechanism enables users to verify if an application is executed in a legitimate SGX-enabled platform. To perform such a verification also known as remote attestation, SGX transforms a local report into a verifiable quote [55] using the SGX's internal Quoting Enclave. The Quoting Enclave prepares an SGX attestation signature using an SGX attestation key which is unique to the SGX-hardware. The generated quote can then be delivered to remote platforms for verification [55, 58].

### 3.2.4 Envisioned Integration and Challenges

In CLOUDSKIN, we target a hardware-agnostic approach which requires a runtime for WebAssembly that automatically detects the provided hardware features and takes appropriate measures such that the C-Cell code runs in isolation and achieves the highest level of protection.

However, this imposes several challenges: First, the underlying hardware must support isolation mechanisms (e.g., enclaves such as provided in Intel SGX). Second, such trusted execution environments need to provide means for outside communication, i.e., through system calls in order to establish data exchange either through the network and file system with e.g. other C-Cells. Third, migration mechanisms must be established for enabling the migration of the sensitive C-Cells across different hardware. This is challenging as the destination host might offer different hardware capabilities than its origin, lowering the provided guarantees in case a migration is executed. Alternatively, a workload migration can be rejected or must be performed using a different target node in order to satisfy the user's security requirements.

## 3.3 Faasm: High-Performance Serverless Runtime

Faasm [23] is a high-performance stateful serverless runtime. In a serverless environment, users register pieces of code - functions - that they want to execute on-demand. Functions tend to be short-lived, meaning that low-startup time and high-density become of paramount importance. Faasm isolates functions using a lightweight mechanism called a Faaslet. Faaslets are based on threads which operate in a shared address space on each host. This means that, while Faaslets provide isolation and fair access to resources, they also support concurrent, zero-copy access to shared state held in memory. This is in contrast to existing serverless platforms which isolate functions in their own container of VM, and do not support parallel processing on shared data.

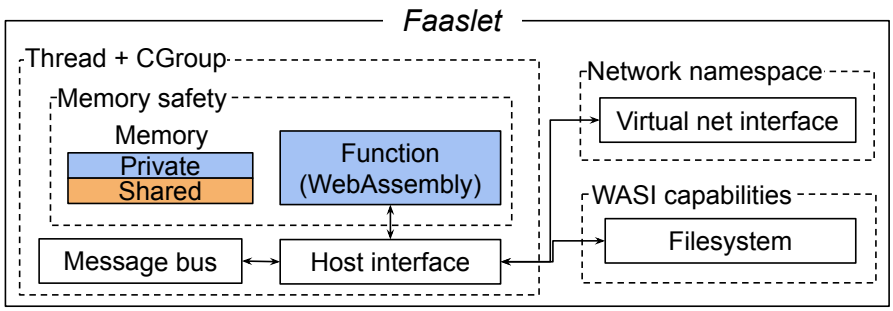



Figure 3: Faaslet isolation in Faasm

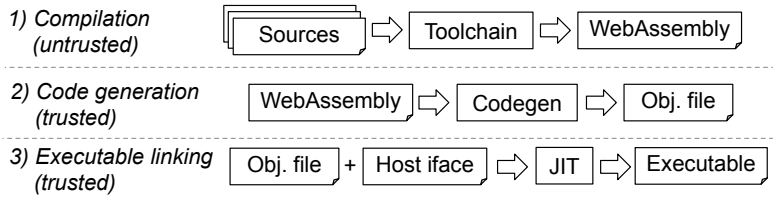



Figure 4: Creation of a Faaslet executable

### 3.3.1 Faaslets: thread-based isolation with WebAssembly

Faaslets provide multi-tenant isolation with orders of magnitude lower overheads than containers or VMs. This is done in part, using software fault isolation (SFI) with WebAssembly (§3.1). Each function associated with a Faaslet, together with its library and language runtime dependencies, is compiled to WebAssembly before being uploaded to the system. The Faasm runtime then executes multiple Faaslets, each with a dedicated thread, within a single address space. For resource isolation, the CPU cycles of each thread are constrained using Linux cgroups [59] and network access is limited using network namespaces [60] and traffic shaping. Many Faaslets can be executed efficiently and safely on a single machine.

The structure of a Faaslet is depicted in Figure 3. Faaslets are built around an instance of a WebAssembly module to provide the necessary isolation guarantees for multi-tenancy in serverless clouds; in contrast, traditional serverless systems typically rely on containers [61, 62]. Faaslets provide a more lightweight execution environment than containers by only virtualising the necessary environment for serverless functions. Therefore, Faaslets have a low memory footprint and can be spawned in the hundreds of microseconds against hundreds of milliseconds for container. This brings Faaslets much closer to the user's idea of the programming model of a function, which FaaS is meant to provide and is the unit of granularity that Faasm manages. We provide below details of the resource control Faasm has in place.

To mitigate the serverless cold starts, users can define initialisation code separately from their main function code, during which the language runtime and packages will be loaded. The resulting WebAssembly memory can be safely serialised at this point and saved to the state which once pulled on each host set to run this Faaslet will speed up start-up times by 490× compared to the equivalent start-up process for a container.

### 3.3.2 Building Faasm functions

Each function in Faasm is first compiled to WebAssembly and uploaded to the system. To convert this into an executable, it needs to be combined with the host interface and other Faasm helper libraries. This process is outlined in Figure 4 and is made up of three steps, with the user only aware of the first. This first step generates a WebAssembly module that can safely handled onwards thanks to the WebAssembly guarantees (§3.1). Faasm relies on a trustworthy open-source WebAssembly

embedder, WAVM [47] and LLVM-JIT libraries [31], to validate and manipulate WebAssembly modules and object code, which make up the second and third steps.

### 3.3.3 Faasm's System Architecture

Faasm is a distributed system. Faasm's architeture comprises a set of workers, in charge of executing Faaslets, an upload service to upload (register) WASM functions, a load-balancer to distribute requests across workers, and a storage backend. This distributed architecture is depicted in Figure 5. Faasm can be deployed both on Kubernetes [63], VMs, or on bare metal.

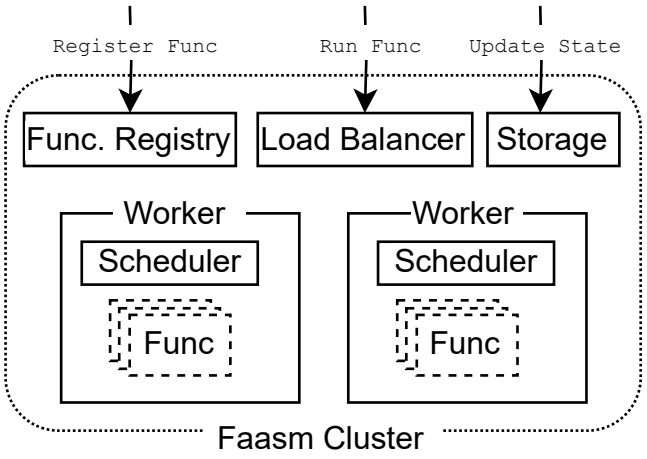



Figure 5: Distributed architecture of the Faasm runtime. It includes three HTTP endpoints: one to register WASM functions, one to upload shared state and data, and one to invoke function execution. Functions are executed in one of the workers, each worker having its own scheduler and state management.

Most notably, the architecture figure does not include a standalone scheduling component. Faasm implements a distributed shared state scheduler, similar to Omega [64]. Requests are randomly routed to a worker instance. If a function need to be scaled up, the worker will spawn Faalets locally, and scale-out to other workers if local resources are exhausted. This simplified scheduling design reduces cold-starts, but prevents system-wide optimisations.

## 3.4 SCONE

TEE such as Intel SGX are typically integrated into existing applications using the respective SDKs provided by the hardware manufacture. Although the Intel SGX SDK provides software developers interfaces to run applications in enclaves in its entirety, the usage of such interfaces is cumbersome as it requires developers to provide glue code for every possible system call an application may use. To overcome this burden, frameworks such as SCONE [65] or Haven [66] evolved to simplify or even completely eliminate manual steps required to run applications in TEEs such as Intel SGX. SCONE achieves this by providing a custom standard C-library which contains glue code as well as all enclave- and system-call handling routines. Additionally, SCONE provides ready to use functionality for attestation and secret provisioning.

The design of SCONE is led by three key principles: First, it focuses on a minimal Trusted Computing Base (TCB) to reduce the number of potential bugs inside an enclave by keeping a small code base. Second, it uses asynchronous system calls and internal threading to reduce application transitions between the so-called enclave mode and normal mode. Third, it protects the file system and network data by transparently shielding the application. 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.

Since SCONE enables users to run legacy applications in Intel SGX enclaves without modifications through its modified dynamic loader and its extended C-library, we envision the use of SCONE as one of the building blocks for the Cloud-Edge Cells.

# 4 Cloud-Edge Cells : Initial Prototype

In this section we describe the first iteration of Cloud-Edge Cells. We will refer to it as C-Cells D4.1, or just C-Cells. This initial prototype satisifies some of the requirements outlined in Table 1, but not all of them. In the coming deliverables we plan on expanding this design (§6), provide a more thorough evaluation, and integrate it with the industrial use-cases in CLOUDSKIN.

The rest of this section is structured as follows: first, we introduce the C-Cell abstraction (§4.1); second, we describe how C-Cells achieve language and API independence (§4.2); third, we describe how C-Cells can be live-migrated (§4.3); fourth and last, we depict the architecture of the system running C-Cells (§4.4). Most notably, the initial prototype of C-Cells presented in this deliverable is not hardware agnostic as it only supports being executed in x86-64 servers, and does not have any adaptive virtualisation features. We leave these features as future work for coming deliverables (§6).

## 4.1 C-Cell abstraction

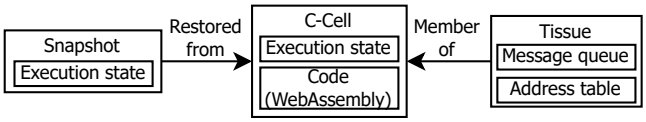



Figure 6: Key abstractions in a C-Cell

The goal of C-Cells is to replace the thread/process abstraction with an execution abstraction that is better suited for the cloud-edge continuum of CLOUDSKIN. To that extent, the goal of this first implementation is to support multi-threading and multi-processing semantics. We succeed in doing so, and present our initial results in §5.

As previously introduced, we replace the thread/process abstraction with C-Cells. Each C-Cell executes application code compiled to WebAssembly [5]. The use of WebAssembly enforces memory safety: its isolation mechanism allows C-Cells to execute side-by-side in a single instance of the runtime. It also allows for a simple snapshotting mechanism because the complete execution state of a C-Cell is captured in the single linear memory array of a WebAssembly module (§3.1). Lastly, the usage of WebAssembly also allows for efficient C-Cell interrupts through undefined symbols and function imports [67], which helps achieving language and API independence (§4.2).

C-Cells are spawned (or restored) from snapshots of other C-Cells, akin to the parent/child semantics of threads and processes. Each snapshot has a copy of a C-Cell's execution state: its linear memory, mutable global variables, function tables and stack pointers. To restore a C-Cell, from a snapshot, the runtime copies the stack pointer, function table and globals from the snapshot into the C-Cell, and creates a copy-on-write mapping of the C-Cell memory onto the snapshot's linear memory. Given the simplicity of WebAssembly's linear memory model, snapshots of child C-Cells can be efficiently represented as diffs of their parent's snapshot.

C-Cells are groupped in Tissues . Equivalently, a Tissue is a group of C-Cells. C-Cells in the same Tissue can share a single distributed address space for multi-threading, or have private memory and exchange messages directly for multi-processing. The runtime can checkpoint a Tissue for live migration (§4.3) or rollback recovery in case of C-Cell or VM failure. Figure 6 summarises the key abstractions in a C-Cell.

Given its isolation properties and the usage of snapshots and Tissues, C-Cells can be transparently deployed in either one or several VMs (only x86-64 architectures are supported, though). Figure 7 shows how C-Cells and snapshots can be used to replicate process and thread semantics across VMs. Each application has a base snapshot, whose linear memory contains the static data of the application.

C-Cells restored from point-in-time snapshots of their parent C-Cells; C-Cells with thread semantics share a single linear memory mapping with other C-Cells on that VM.

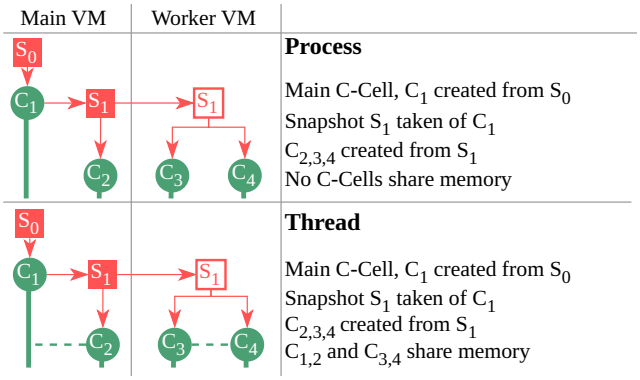



Figure 7: Transparently executing C-Cells across VMs using snapshots and Tissues.

### 4.2 Language Independence

To achieve language and API independence, C-Cells need to support: (i) multiple programming languages, (ii) multiple host environments, and (iii) arbitrary API calls. The first point is implicit in WebAssembly as exposed in §3.1. The second and third points derive from the ability to implement a low-overhead interrupt mechanism for C-Cells. We name this collaborative interrupt mechanism control points . C-Cells execution may be interrupted at control points in order to: (i) provision new C-Cells to change the scale of an application (e.g. to replicate the behaviour of `fork`); (ii) migrate C-Cells to change the distribution of an application (e.g. to off-load computation from an edge device to the cloud); (iii) synchronise shared memory between VMs; (iv) deliver messages between C-Cells; or (v) checkpoint an application for rollback-recovery.

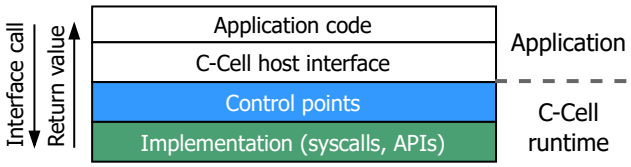



Figure 8: Control points (Control points are triggered when application code calls functions from supported APIs, before the host runtime implementation of the given function is executed.)

A control point is triggered when a C-Cell invokes certain system calls and supported APIs (Figure 8). These symbols are left undefined when cross-compiling application code to WebAssembly, and marked as function imports [67]. The runtime then provides definitions for these symbols, and links them to the WebAssembly module at runtime. We use this approach to transfer control to the runtime on system calls related to thread and process operations, e.g. `pthread_create()` and `fork()`, and will be able to support any custom API similarly.

### 4.3 Live Migration of C-Cells

In the cloud-edge continuum, C-Cells must support continued execution between the cloud and the edge. This means not only being able to run in the cloud and in the edge, but also be able to begin executing somewhere, checkpoint and stop upon request, and resume execution elsewhere. In the literature this process is known as live migration [68]. Even though research in live migration is abundant [8, 69], CLOUDSKIN faces an additional difficulty: live migration needs to happen between heterogeneous hosts, potentially running different instruction sets.

Traditional process migration tools like CRIU [8] do not support heterogeneity between migrated hosts. This is because the checkpointing mechanism relies heavily on the underlying operating system and will not support subtle changes in the environment, for example different shared libraries version. More modern migration tools, like pod checkpoint-restore in Kubernetes [70] will work as long as migration happens within the same instance of the deployed Kubernetes cluster. For CLOUDSKIN we deploy instances of the runtime in each host, with potentially different instruction sets, and use snapshots together with careful tracking of accesses to the host environment (through control points) to implement our WebAssembly-specific checkpoint-restore mechanism. For C-Cells that communicate via message passing or make use of shared memory (i.e. using some sort of multi-processing or multi-threading API), special care must be taken when checkpointing the state of the C-Cells, not to lose any in-flight messages or shared state. To this end, we only allow live migration to happen at barrier control points. These are control points that block all C-Cells of an applcation.

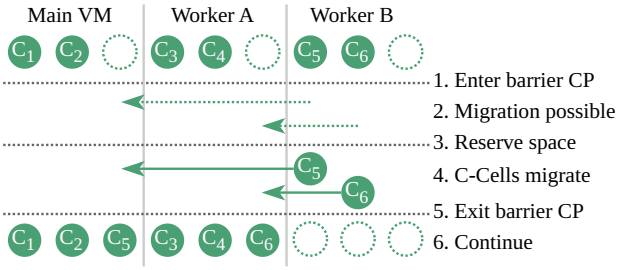



Figure 9: C-Cell migration at barrier control points

Figure 9 illustrates C-Cell migration. When C-Cells reach a barrier control point, they wait for a notification from the application's main VM. When the main C-Cell reaches the barrier control point on the main VM, it queries the Planner to know whether it must trigger a migration or not. The Planner is a new centralised scheduling component that we add on top of Faasm and cover in detail in §4.4. If the Planner indicates to migrate, each C-Cell either snapshots its state and sends it to the recipient VM, or waits on the post-migration barrier point. Once all C-Cells in an application hit the post-migration barrier, execution can successfully resume.

The actual migration of a C-Cell is performed via the same mechanism used to create child processes and threads. The migrating C-Cell takes a snapshot, and sends the snapshot as part of a migration request to the target VM. The target VM creates a new C-Cell with the required semantics, i.e. with a new private mapping of the snapshot's linear memory for a process, or sharing a linear memory mapping with existing C-Cells for a thread.

### 4.4 System Architecture

In this section we describe the runtime architecture for CLOUDSKIN. The system builds on top of Faasm, and replaces the Faaslet isolation unit with C-Cells. In particular, and similarly to Faasm, the runtime architecture for CLOUDSKIN (or CLOUDSKIN for short), exposes three HTTP endpoints: one to register application code cross-compiled to WebAssembly, one to invoke applications via a load balancer, and one to upload shared data and state (see Figure 5).

However, there are a number of ways in which CLOUDSKIN builds upon, and modifies, Faasm. First, the set of distributed workers is now deployed in an heterogeneous VM set, with different architectures and instruction sets. This means that we need to modify the worker runtime to run in each different device, including the WebAssembly runtime. Second, accommodating to different instruction sets also means we need to modify the service to register application code cross-compiled to WebAssembly. This is because as part of the registration, Faasm generates optimized machine code to the target architecture it runs on. For CLOUDSKIN, it needs to generate as many versions of the optimised machine code as target architectures present in the deployment. Lastly, to integrate with the learning plane and scheduling presented in D5.1, we re-implement the scheduler from a

simple distributed-state one, to a centralised implementation that we call the Planner .

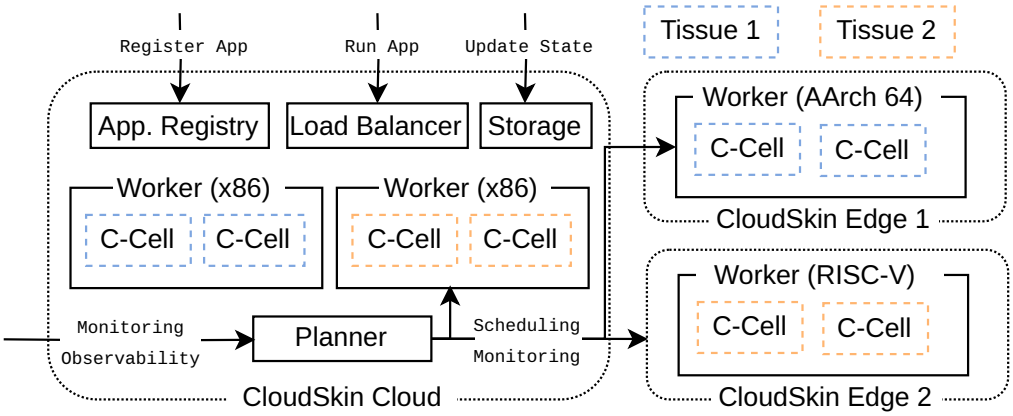



Figure 10: High level diagram of the CLOUDSKIN's system architecture for the cloud-edge continuum. The execution unit in CLOUDSKIN are C-Cells, which are groupped in Tissues.

The CLOUDSKIN architecture is illustrated in Figure 10. The Planner keeps a centralised view of the state of all applications running in CLOUDSKIN. Each application is made up of one or more C-Cells, grouped in Tissues. Depending on their configuration, C-Cells in the same Tissue can either share a common address space, or send messages to each other. The Planner allocates C-Cells to VMs according to a very simple scheduling policy, and exposes HTTP endpoints for monitoring and observability. The implementation of the Planner for the first prototype presented with D4.1 is very simple, but its design should allow for simple integration with more advanced scheduling policies developed in Work Package 5.

## 5 Evaluation

In the evaluation of the initial C-Cell prototype, we want to demonstrate that (i) C-Cells can replace threads/processes as execution abstraction with modest overhead (§5.1), (ii) C-Cells can be transparently live-migrated mid-execution (§5.2), and (iii) C-Cells can transparently execute over a distributed set of VMs (§5.3). Throughout the evaluation section we report the execution time speed-up with respect to native execution. The speed-up is defined as the ratio between the execution time using threads and processes and the execution time using C-Cells: Speed-up $= \frac{t_{th/pr}}{t_{\text{C-Cell}}}$. This way, a speed-up greater than one means that C-Cells execute faster than threads/processes. Symmetrically, a speed-up lower than one means that C-Cells execute slower than threads/processes. Bear in mind that the current C-Cell prototype only runs in x86-64 systems, and does not support adaptive virtualization.

### 5.1 Executing (multi-)threads and (multi-)processes with C-Cells

To illustrate that C-Cells are a suitable replacement for threads and processes, we port a multi-threading (with shared memory) and a multi-processing (with message passing) framework to the CLOUDSKIN execution environment. Given its popularity and widespread use, we pick OpenMP [71] and MPI [72]. To support the OpenMP and MPI APIs, we follow the procedure indicated in §4.2. First, we cross-compile two applications using each API. For OpenMP, we pick a dense matrix multiplication (DGEMM), part of the ParRes kernels [73]. For MPI, we pick LAMMPS [74, 75], a popular molecular dynamics simulator written in C++. During cross-compilation, we leave all MPI and OpenMP symbols undefined; they will trigger control points at runtime. Second, we implement all these undefined symbols in the C-Cell runtime, adapting the behaviour of the symbol to the CLOUDSKIN environment. For example, during `MPI_Init` the MPI runtime would spawn `MPI_WORLD_SIZE - 1` child processes. Instead, the C-Cell runtime spawns child C-Cells with private memory mappings and creates a Tissue so that C-Cells can directly message each other.

**Multi-threading results.** For the OpenMP application, we deploy a cluster in the cloud with just one worker. We use a 1-VM Kubernetes cluster in Azure. The VM is of type Standard_D8_v5 VMs [76] with 8 vCPU cores and 32 GB of memory. As a baseline we execute native OpenMP directly on the VM. We measure the ratio of execution times (slowdown) as we increase the number of parallel C-Cells/OpenMP threads.

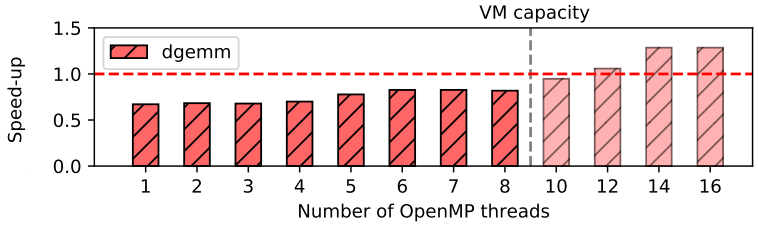



Figure 11: Slowdown executing a dense matrix multiplicatoin (DGEMM) in an 8 vCPU VM using C-Cells compared to native OpenMP threads.

Figure 11 summarises the results. We see that, for lower thread counts, C-Cells introduce a noticeable overhead with respect to native OpenMP threads. This is due to the added overhead of spawning C-Cells compared to threads. However, this cost is amortised as we use more OpenMP threads (up to the 8 available cores in the VM).

**Multi-processing results.** For the MPI application we deploy a 2-VM cluster with the same VM type using Kubernetes on Azure. We compare against native OpenMPI running on the underlying VMs. We measure the execution time as we increase the number of C-Cells/MPI processes and report the speed-up. We use two different LAMMPS benchmarks from LAMMPS' benchmark suite [77]. The unmodified LJ benchmark with 4 million atoms (`compute`-bound), and a modified benchmark to increase the communication rounds (`network`-bound).

Figure 12 summarises the results. For the compute-bound, MPI with C-Cells performs equally or better than native OpenMPI processes. This is because MPI processes communicate scarcely, and mostly with co-located MPI processes. In this scenario, C-Cells benefit from very fast in-memory messaging, compared to native OpenMPI's comparatively costly IPC. For the network-bound benchmark, where inter-VM communication is abundant, C-Cells introduce a moderate 5-10 % worst-case overhead. This is because, to manage Tissues, we must add an extra layer of indirection in network communications.

### 5.2 Live Migration of C-Cells

This experiment measures the benefit of migrating C-Cells at runtime (i.e., live migration). Note that native OpenMPI can not transparently migrate MPI processes while guaranteeing execution integrity (this is, that no messages are lost). As baselines, we run the same MPI application as before, LAMMPS, and a network-bound all-to-all kernel. The all-to-all kernel performs an accumulation and

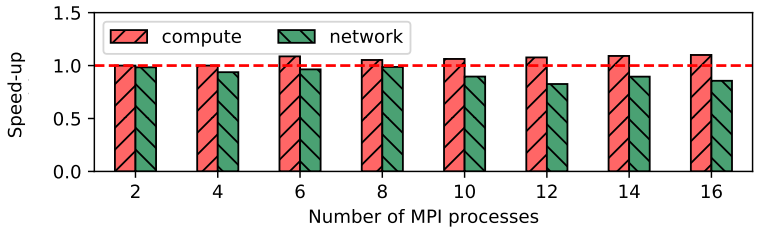
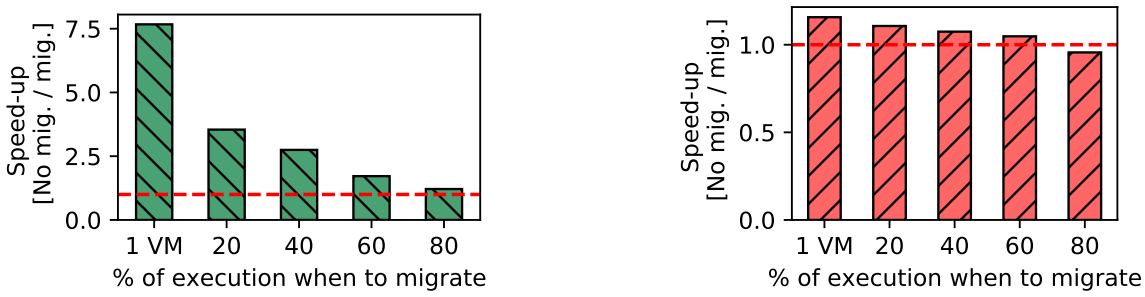



Figure 12: Speed-up executing an MPI application (LAMMPS) using C-Cells compared to native OpenMPI processes.




(a) All-to-all (b) LAMMPS

Figure 13: Speed-up when live-migrating C-Cells compared to not-doing so. A speed-up greater than 1 means that the execution time (including migration time) is lower than the execution time without migrating.

reduction over a vector of shared variables, and synchronises the results across all MPI ranks. Then, it repeats the same procedure in a tight loop.

We execute each application with 8 MPI ranks that we split over two 8-vCPU machines: 4 MPI ranks in one machine, 4 MPI ranks in the other, and each machine has 4 idle vCPUs. We then migrate 4 MPI ranks from one machine to the other at 20%, 40%, 60%, or 80% of execution. For reference, we also measure the execution time if the 8 MPI ranks are co-located in the same machine from the beginning (1 VM).

Figure 13 shows the speed-up achieved when migrating. This time the speed-up is calculated as the ratio between the execution time of an execution where the 8 ranks are split between two machines, and the execution time of an execution where we perform a live migration. A speed-up greater than 1 means that the execution time (including migration) is lower than the execution time when not migrating.

For a network-bound kernel, halving the ranks across VMs has a high cost: the speed-up for 1 VM is 7.5×. By migrating after 20%, 40%, 60%, and 80%, of execution, we achieve speed-ups of 3.5×, 2.7×, 1.7×, and 1.2×, respectively. We conclude that, for a network-bound application, migrating C-Cells at runtime always reduces execution time.

For a compute-bound kernel, halving the ranks across VMs has a lower cost: the speed-up for 1 VM is 1.2×. This is because the fragmentation splits 4 processes in 1 VM, and 4 processes into another, which means that there is substantial intra-VM messaging. By migrating after 20%, 40%, and 60% of execution, we achieve speed-ups of 10%, 8%, and 5%, respectively. When migrating after 80% of execution, the costs of migrating outweighs its benefits, achieving a slow-down of 5%. LAMMPS has large code and data sections, which leads to larger C-Cell snapshot, increasing the cost of migration.

### 5.3 Transparent distribution across VMs

This experiment measures the benefits of transparently distributing C-Cells across distributed VMs. To do so, we execute the same dense matrix multiplication using OpenMP (DGEMM) increasing the number of C-Cells/OpenMP threads (see §5.1), but when exhausting the local vCPUs the C-Cell runtime will start using C-Cells in another VM. Note that this is not possible in native OpenMP as it would require a general-purpose distributed shared memory implementation, which is known to be very hard to achieve without giving up on performance [78, 79]. On the other hand, we use a lightweight synchronisation mechanism between C-Cells based on snapshot diffs.

Figure 11 summarises the results (when the thread count exceeds 8 OpenMP threads). In particular, we compare the execution time using more C-Cells distributed across two VMs with just using 8 native OpenMP threads. Interestingly, we measure a 25 % speed-up when using twice as many C-

Cells. Even though this may not be a resource-efficient result, we believe it showcases the feasibility of C-Cell synchronisation using diffs.

## 6 Future Lines of Research

In Table 1 we presented the different requirements C-Cells must have to power the cloud-edge continuum in CLOUDSKIN. We compare it with other existing execution units, and with the prototype implementation presented in this deliverable (§4). Inspecting the table, it is clear which are the areas we need to work on in the coming deliverables.

First, C-Cells must support concurrent execution in heterogeneous architectures and instruction sets. There are a number of challenges associated in doing so. The WebAssembly runtime must be suitable to run in different architectures, which not all of them are. In addition, not all WebAssembly execution modes (interpreted, ahead-of-time compiled, and just-in-time compiled) are supported in all architectures, which can make support even more challenging. In terms of scheduling, the Planner needs to be aware of the different supported architectures and incorporate this information into its scheduling policies, taking into consideration the different capabilities of different resources. Lastly, access to shared memory or the network may also be different, yet the interface to C-Cells will have to remain the same.

Second, C-Cells must support adaptive virtualisation. This is, depending on the data they are processing, C-Cells must support transparently executing inside TEEs. There are two ways to transparently execute C-Cells inside TEEs depending on the granularity of the TEE itself. First, we can integrate with existing WASM runtimes that support executing WASM modules inside SGX enclaves, like WAMR. This means that, within a single process, we could spawn as many TEEs as necessary. C-Cells could still be co-located inside a TEE or not, but, most importantly, the local C-Cell runtime would have visibility on which C-Cells run inside TEEs. Second, we could integrate with existing *lift-and-shift* programs that transparently run unmodified linux processes inside TEEs, like SCONE. This means that the whole local C-Cell runtime runs inside a TEE, and consequently all C-Cells scheduled therein are also co-located in the TEE. In this case only the Planner, and not the local C-Cell runtime, has visibility into which C-Cells run in a TEE.

The first option is theoretically more resource efficient, as it allows to package C-Cells more tightly and reduces memory duplication. However, it requires considerable engineering effort, as all the interactions with the host operating system need to be re-implemented to fit the chosen TEE's ECall/OCall mechanism. Furthermore, we can leverage one of the partner's expertise in the development and usage of SCONE, greatly simplifying the adoption of virtualisation. All things considered, we decide to initially explore option two. This is, transparently lift-and-shift the local C-Cell runtime into a TEE.

## 7 Conclusions

In this deliverable, we have presented the first prototype design and implementation for Cloud-Edge Cells, the execution unit of the CLOUDSKIN cloud-edge continuum. We have started-off by listing the requirements for C-Cells, which we worked out to be: language and API independence, C-Cells must be able to execute unmodified arbitrary applications; hardware agnostic, C-Cells must support transparent execution on different hardware devices with different instruction sets; live migration, to enable the cloud-edge continuum C-Cells must support (live) migration across the cloud-edge; and adaptive virtualisation, C-Cells must support transparent execution inside TEEs when processing confidential data.

The first C-Cell design builds on Faasm, a distributed runtime that uses WebAssembly-based isolation using Faaslets. We modify Faasm to fulfil CLOUDSKIN's requirements. First, we give C-Cells (Faasm's Faaslets), optional thread- and process-like semantics to support a wide-range of applications. Second, we change the simple distributed scheduler into a centralised one which we name Planner. Third, we use the Planner to implement live migration of C-Cells.

In order to evaluate this first prototype, we port a multi-threading (OpenMP) and a multi-processing (MPI) API to CLOUDSKIN and show limited overhead when compared to native execution. In addi-

tion, we show how C-Cells isolation and transparency can be used to achieve features unsupported by the native thread/process counterpart: we perform live-migration of an MPI application and we extend an OpenMP execution across VMs. Both features are achieved without having to change the application code.

The prototype we present here does not tick all the requirements yet. In the coming deliverables we will work on supporting execution on heterogeneous devices, implementing adaptive virtualisation using TEEs, and integrating the planner with the learning and orchestrating plane.

## References

[1] GeeksForGeeks, "What is a Linux Process?." https://www.geeksforgeeks.org/processes-in-linuxunix/, 2023.

[2] Docker, "What is a container?." https://www.docker.com/resources/what-container/, 2023.

[3] VMWare, "What is a Virtual Machine?." https://www.vmware.com/topics/glossary/content/virtual-machine.html, 2023.

[4] Cloudflare, "How workers work." https://developers.cloudflare.com/workers/learning/how-workers-works/, 2023.

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

[6] Docker, "Multi-platform," 2023.

[7] Wikipedia, "Java virtual machine (jvm)," 2023.

[8] CRIU, "Checkpoint-resotre in userspace," 2023.

[9] Docker, "Docker checkpoint-restore," 2023.

[10] IBM, "Migrating virtual machines using vmware vmotion," 2023.

[11] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, "Portable support for transparent thread migration in java," in Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, (Berlin, Heidelberg), p. 29–43, Springer-Verlag, 2000.

[12] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in 2017 USENIX Annual Technical Conference (USENIX ATC 17), (Santa Clara, CA), pp. 645–658, USENIX Association, July 2017.

[13] Wikipedia, "Software Guard Extensions." https://en.wikipedia.org/wiki/Software_Guard_Extensions, 2023.

[14] AMD, "AMD Secure Encrypted Virtualization (SEV)." https://www.amd.com/en/developer/sev.html, 2023.

[15] Intel, "Trust Domain Extensions." https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2023.

[16] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch, "SGX-LKL: securing the host OS interface for trusted execution," CoRR, vol. abs/1908.11143, 2019.

[17] Bytecode Alliance, "The WebAssembly Micro Runtime." https://github.com/bytecodealliance/wasm-micro-runtime, 2020.

[18] Wikipedia, "Graphical processing unit," 2023.

[19] Wikipedia, "Tensor processing unit," 2023.

[20] G. Project, "GSC: Gramine Shielded Containers." https://github.com/gramineproject/gsc, 2023.

[21] F. Besson, S. Blazy, A. Dang, T. Jensen, and P. Wilke, "Compiling sandboxes: Formally verified software fault isolation," in Programming Languages and Systems (L. Caires, ed.), (Cham), pp. 499–524, Springer International Publishing, 2019.

[22] J. Lo, E. Wohlstadter, and A. Mesbah, "Live migration of javascript web apps," in Proceedings of the 22nd International Conference on World Wide Web, WWW '13 Companion, (New York, NY, USA), p. 241–244, Association for Computing Machinery, 2013.

[23] 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.

[24] Docker, "Docker + wasm (beta)." https://docs.docker.com/desktop/wasm/, 2023.

[25] Dfinity, "Architecture of the internet computer," 2023.

[26] Docker, "Docker + wasm (beta)," 2023.

[27] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Trust but verify: SFI safety for native-compiled Wasm," in Network and Distributed System Security Symposium (NDSS), Internet Society, February 2021.

[28] C. Watt, "Mechanising and Verifying the WebAssembly Specification," in ACM SIGPLAN International Conference on Certified Programs and Proofs, 2018.

[29] W. Fu, R. Lin, and D. Inge, "TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly," arXiv preprint arXiv:1802.01050, 2018.

[30] D. Lehmann and M. Pradel, "Wasabi: A Framework for Dynamically Analyzing WebAssembly," in ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.

[31] LLVM, "The llvm compiler infrastructure." https://llvm.org/, 2023.

[32] Assemblyscript, "AssemblyScript." `https://github.com/AssemblyScript/assemblyscript`, 2020.

[33] Mozilla, "Pyodide." `https://github.com/pyodide/pyodide`, 2020.

[34] SwiftWasm, "SwiftWasm." `https://swiftwasm.org/`, 2023.

[35] A. Andreev, "TeaVM." `http://www.teavm.org/`, 2020.

[36] WebAssembly, "WebAssembly Specification." `https://github.com/WebAssembly/spec/`, 2020.

[37] Bytecode Alliance, "WebAssembly Binary Toolkit." `https://github.com/WebAssembly/wabt`, 2022.

[38] WebAssembly Working Group, "Roadmap." `https://webassembly.org/roadmap/`, 2022.

[39] WebAssembly, "WebAssembly Dynamic Linking." `https://webassembly.org/docs/dynamic-linking/`, 2020.

[40] C. Reference, "The C++ language." `https://en.cppreference.com/w/cpp/language`, 2022.

[41] Bytecode Alliance - WAMR, "Understanding the WAMR heaps." `https://bytecodealliance.github.io/wamr.dev/blog/understand-the-wamr-heaps/`, 2023.

[42] B. ALliance, "Cranelift." https://cranelift.dev/, 2023.

[43] L. Project, "Clang: a C language family front-end for LLVM." https://clang.llvm.org/, 2022.

[44] The Rust Language, "Rust Toolchains." https://rust-lang.github.io/rustup/concepts/toolchains.html, 2020.

[45] Google, "The Go Programming Language." https://go.dev/, 2020.

[46] Bytecode Alliance, "WASI: WebAssembly System Interface." https://wasi.dev/, 2022.

[47] A. Scheidecker, "WAVM." https://github.com/WAVM/WAVM, 2020.

[48] Bytecode Alliance, "Wasmtime." https://wasmtime.dev/, 2020.

[49] F. D. R. Thoughts, "WebAssembly Benchmark 2023." https://00f.net/2023/01/04/webassembly-benchmark-2023/, 2023.

[50] Google, "V8 Engine." https://github.com/v8/v8, 2020.

[51] Mozilla, "SpiderMonkey." https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey, 2018.

[52] Microsoft, "ChakraCore." https://github.com/microsoft/ChakraCore.

[53] Apple, "WebKit." https://webkit.org/.

[54] B. Alliance, "Bytecode Alliance." https://bytecodealliance.org/, 2023.

[55] "Intel software guard extensions developer guide." https://software.intel.com/en-us/sgx-sdk/documentation, 2014.

[56] "Architecture specification: Intel® trust domain extensions (intel® tdx) module." https://cdrdv2.intel.com/v1/dl/, 2014.

[57] "Amd sev-snp: Strengthening vm isolation with integrity protection and more." https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf?ref=blog.humanode.io.

[58] V. Costan and S. Devadas, "Intel sgx explained." Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[59] T. L. kernel development community, "Control Groups." https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html, 2020.

[60] Linux Manual Page, "network namespaces." https://man7.org/linux/man-pages/man7/network_namespaces.7.html, 2022.

[61] The Knative Authors, "Knative - Enterprise-grade Serverless on your own terms.." https://knative.dev/docs/, 2021.

[62] Amazon Web Services, "AWS Lambda." https://aws.amazon.com/lambda/, 2020.

[63] The Linux Foundation, "Kubernetes." https://kubernetes.io/, 2020.

[64] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in ACM European Conference on Computer Systems (EuroSys), 2013.

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

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

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

[68] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation - Volume 2, NSDI'05, (USA), p. 273–286, USENIX Association, 2005.

[69] M. Emani, I. Laguna, K. Mohror, N. Sultana, and A. Skjellum, "Checkpointable mpi: A transparent fault-tolerance approach for mpi," 10 2017.

[70] R. Hat, "Checkpoint and Restore Kubernetes." `https://developers.redhat.com/articles/2021/10/07/checkpoint-and-restore-kubernetes`, 2023.

[71] OpenMP, "The OpenMP API specification for parallel programming." `https://www.openmp.org/specifications/`, 2021.

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

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

[74] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," Journal of Computational Physics, 1993.

[75] Sandia National Laboratories, "LAMMPS Molecular Dynamics Simulator." `https://lammps.sandia.gov/index.html`, 2020.

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

[77] Sandia National Laboratories, "Benchmarks - LAMMPS Documentation." `https://docs.lammps.org/Speed_bench.html`, 2022.

[78] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and performance of munin," ACM SIGOPS Operating Systems Review, no. 5, 1991.

[79] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," Computer, no. 8, 1991.