Diploma Project

Securing Microservice-Based Applications Using Confidential Containers

Michail Panaetov

DEPARTMENT OF COMPUTER SCIENCE

May 2025

UNIVERSITY OF CYPRUS

Faculty of Pure and Applied Sciences

DEPARTMENT OF COMPUTER SCIENCE

Securing Microservice-Based Applications Using Confidential Containers

Michail Panaetov

Advisor:

Dr. Haris Volos

The Thesis was submitted in partial fulfillment of the requirements for obtaining the Computer Science degree of the Department of Computer Science of the University of Cyprus

May 2025

Acknowledgements

I would like to sincerely thank Dr. Haris Volos for his support and invaluable guidance during the course of this thesis. His dedicated coordination and insightful advice were crucial in bringing this work to completion.

I would like to express my deepest gratitude to my parents for their unwavering support, love, and encouragement throughout this journey. Their belief in me and their constant presence have been my foundation during the most challenging moments of this thesis.

I also extend my heartfelt thanks to my friends, whose companionship, understanding, and motivation have been invaluable. Whether it was through a listening ear, a word of encouragement, or simply their presence, they made this endeavor more manageable and enjoyable.

ABSTRACT

From conversational agents to code assistants, large language models (LLMs) have transformed everything in the past few years, as has automatic summarization services. Since these potent AI capabilities are being integrated into production systems by organizations— often dealing with confidential information—there is an increasing demand for ensuring precision, speed, confidentiality, and infallibility during inference. For this particular thesis, the goal is to figure out the time and computation resources needed to execute an LLM in a truely isolated, hardware-backed secure enclave.

I implement a compact version of Meta's Llama2—llama2.c—into a RISC-V virtualized system emulated by QEMU on top of the open-source Keystone framework. This achievement will have two objectives: Firstly, to determine the measurement performance aggravations that take place when an LLM is embedded into a Trusted Execution Environment (TEE); secondly, to illustrate the irrefutable obstacles that one faces while blending the state-of-the-art AI inference and modern secure-container technologies.

Measuring:

Compare the "secure" setup to the standard unprotected execution framework Enclave loading time: how long needed to initialize Keystone runtime, load model weights into protected memory, and establish essential cryptographic primitives needed?

Enclave startup time: How long does it take to initialize the Keystone runtime, load the model weights into protected memory, and establish the necessary cryptographic primitives?

Overall throughput impact: Under a stream of requests, how much does the enclave's overhead slow down sustained inference workloads?

I highlight the primary integration challenges including toolchain eccentricities, memory-layout restrictions, as well as I/O constraints pertaining to the TEE, and also touch upon initial exploration of a preemptive Confidential Containers configuration, providing initial perspectives on different secure-microservice implementations.

Overall, this study illustrates a multifaceted scenario: secure enclaves are capable of hosting LLM inference but incur substantial costs in terms of startup delay and per-query latency. These overheads could necessitate paradigm shifts in design—batching strategies, model quantization, or hybrid enclave architectures—for privacy-concerned cloud microservices or resource-constrained edge devices. By illuminating these performance-security trade-offs, I hope to encourage system architects, AI practitioners alike, to rethink the limits in trusted AI implementations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

- **SDK**: Software Development Kit
- **Avg**: Average
- **PL**: Percentile Latency
- **CPU**: Central Processor Unit
- **LLM**: Large Language Model
- **OS**: Operating System
- **QEMU**: Quick Emulator
- **RISC-V**: Reduced Instruction Set Computer – Five
- **TEE**: Trusted Execution Environment
- **VM**: Virtual Machine

1 Introduction

The microservices paradigm decomposes applications into lightweight, independent services—boosting agility but expanding the attack surface. This thesis investigates securing those workloads by running llama2.c, a compact LLM implementation, inside Keystone□based Confidential Containers on a RISC-V TEE. We integrate the Confidential Containers runtime with Keystone to automate enclave instantiation and attestations, then measure startup time, per-inference latency, and overall runtime overhead. Our findings quantify the key performance–security trade-offs of enclave-based microservice deployments.

1.1 Motivation

Our motivation stems from the need to evaluate and reduce the latency and overhead associated with deploying large language models (LLMs) in trusted execution environments (TEEs). One significant performance concern we identified is the potential latency induced by the security boundaries and resource isolation mechanisms within TEE frameworks such as the Keystone open-source project. In this context, our research focuses on investigating the performance implications of executing the llama2.c model inside a Keystone enclave, where the application runs in a trusted environment while the host remains untrusted.

Expanding on our motivation, the necessity to ensure both security and efficiency in machine learning workloads has grown with the sharp increase of productivity of LLMs in privacy-sensitive and cloud-deployed applications. In such environments, maintaining data confidentiality and integrity is critical, but this must not come with computational delays. As language models become more complex and widely adopted, even minimal overheads can degrade the responsiveness and usability of these systems in real-world deployments.

The specific challenge we address is the latency and performance overhead introduced by running complex inference workloads, like those of LLaMA2, within enclaves that enforce strict isolation. These overheads may stem from enclave context switches, limited access to system resources, memory copying between trusted and untrusted regions, or I/O constraints. Such factors can significantly impact throughput and tail latency—two metrics critical for real-time inference systems.

Recognizing this challenge, our research endeavors to quantify and analyze the overheads involved when executing llama2.c within Keystone, thereby offering insight into the trade-offs between security and performance. We aim to answer whether running LLMs in a TEE can be both secure and efficient enough to support latency-sensitive applications. This includes detailed profiling of runtime behavior, memory and CPU usage, and latency breakdowns.

Our motivation is rooted in the belief that addressing these performance drawbacks is essential for making secure, enclave-based machine learning a viable option in production. Through our work, we aim to contribute to the design space of trusted LLM inference, ultimately promoting architectures that balance trust, speed, and scalability. Our findings can help guide the adoption and optimization of secure AI services, fostering the development of more resilient and privacy-preserving software ecosystems.

1.2 Our Hypothesis

We aim to determine whether the execution of a large language model like LLaMA2 in an enclave of Keystone can make tangible observations on latency and overhead incurred by trusted environments. The motivation behind achieving this objective stems from the assumption that running the model inside an enclave would impact system responsiveness due to enclave transition overhead, memory protection, and limited access to host resources. By executing llama2.c on Keystone and monitoring its run-time behavior, we suspect that the enclave will experience measurable delays compared to native execution. These may be due to restricted system calls, enclave context switching, or memory copying between the trusted and untrusted domains. However, if these overheads are within acceptable limits, the trade-off could be worthwhile for applications where data confidentiality and integrity are of the essence. Through this work, we aim to evaluate the practical feasibility of using enclaves for LLM inference and to better understand the security-performance trade-off of enclave-based deployments.

1.3 Trusted Execution Overhead in Enclave-Based Inference

The performance characteristics of enclave-based applications differ significantly from traditional execution environments. In particular, trusted execution environments like Keystone force strict memory isolation, limited system call access (OCALL), and controlled communication with untrusted components. While these mechanisms enhance the security of sensitive workloads, they can also introduce additional overhead and latency.

For inference workloads involving large language models such as LLaMA2, these overheads may arise during memory transfers between trusted and untrusted memory regions, enclave entry and exit operations, and constrained interaction with the host operating system. These factors can become especially critical when evaluating real-time or near-real-time applications, where responsiveness is essential.

This thesis investigates the extent of such performance impacts by running the llama2.c implementation inside a Keystone enclave and analyzing key metrics such as total execution time, latency breakdown, and resource utilization. The following sections will detail the setup of the enclave-based inference environment, the benchmarking methodology, and the insights drawn from comparing enclave and non-enclave executions.

1.4 Contributions

Our research offers two principal contributions to the secure deployment of micro-service workloads; each is explained in depth below to underscore both its technical scope and its practical significance.

- **Attempted Confidential Containers Deployment on Kubernetes:**
  Building on our code-porting work, we set up a vanilla Kubernetes cluster augmented with the Confidential Containers (CoCo) add-on and attempted to run the statically linked `llama2.c` container inside that environment. Our pipeline automatically pulled the signed image, validated its provenance, and handed control to the CoCo runtime without requiring changes to the user's deployment manifest. The effort ultimately stalled because the available worker nodes lacked Intel SGX support—the secure-hardware foundation that CoCo expects by default. Despite this limitation, the clusters we built and the code we produced form a reusable scaffold for future experiments on SGX-capable clusters and illustrate the practical hurdles of deploying LLM inference within a hardware-backed TEE on contemporary cloud infrastructure.
- **Adaptation of llama2.c for Keystone TEEs:**
  We undertook a comprehensive effort to adapt the open-source `llama2.c` implementation for execution inside the Keystone trusted execution environment on RISC-V hardware. This ongoing port involved cross-compiling the codebase with *musl* for static linking, refactoring file-system interactions to rely on enclave-compatible abstractions, and redesigning I/O pathways to enable deterministic token generation across diverse prompt lengths. Although the work did not culminate in a fully functioning enclave deployment, the extensive modifications and build tooling we developed constitute a reproducible foundation for future researchers aiming to combine large-language-model inference with strong hardware isolation.

Overall, our contributions advance the understanding of how trusted execution environments interact with modern machine learning workloads. They provide both theoretical and practical insights into the feasibility of secure, efficient AI inference in resource-constrained and confidential environments.

2 Background

2.1 Trusted Execution Environments (TEEs)

Trusted Execution Environments (TEEs) create hardware–protected regions where code and data execute shielded from the privileged software stack. The common goal is to shrink the *trusted computing base* (TCB) while providing confidentiality and integrity for ”data-in-use”. Commodity CPU TEEs such as Intel SGX and Arm TrustZone inspired a new wave of open designs (e.g., Keystone on RISC-V) and confidential-computing offerings in all major public clouds. In parallel, both academic and industrial efforts are bringing similar protections to *accelerators* - most notably GPUs - which is crucial because large AI models are still served predominately from GPU clusters.

The rest of this chapter surveys (i) CPU-centric TEEs that are mature enough to run unmodified Linux guests, (ii) emerging GPU TEEs, (iii) how these technologies enable confidential containers, and (iv) why this thesis focuses on **characterising performance overheads** rather than proposing a new design.

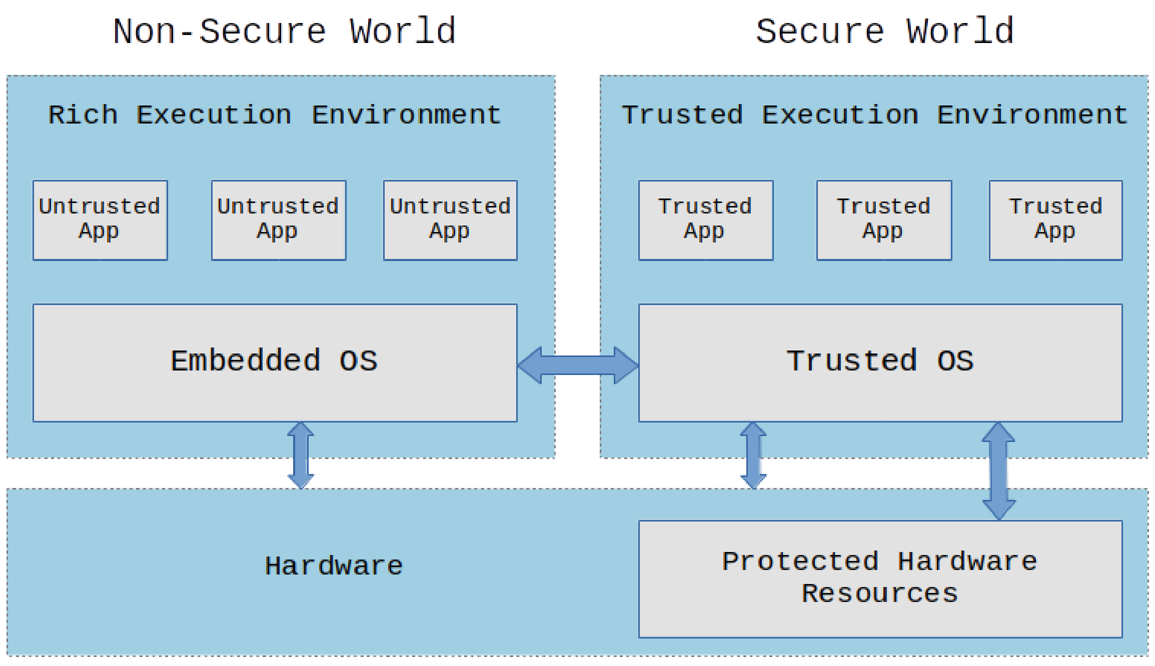

Figure 2.1: TEE Architecture

2.1.1 CPU TEEs

2.1.1.1 Intel SGX

Intel Software Guard Extensions (SGX) isolates *user-space* code and data inside an *Enclave Page Cache* (EPC) whose cache lines are transparently encrypted as they leave the package and checked for replay on re-entry [1]. A transition into or out of an enclave (an ECALL/OCALL) flushes almost all micro-architectural state—TLBs, BTB, µop cache, branch predictors—and costs ≈17 000 cycles [2], while an EPC page fault that evicts an encrypted cache line to untrusted DRAM costs a further ≈12 000 cycles [3]. Early server parts exposed only a 128 MiB EPC [1], so memory-hungry workloads quickly spilled into this slow paging path; later Ice Lake Xeon SP chips doubled the window to 256 MiB [4] and, with *Enclave Dynamic Memory Management* (SGX2), let the OS hot-plug EPC pages so long-running services can adapt to fluctuating footprints [5]. Remote attestation originally relied on Intel's cloud service, but the *Data-Centre Attestation Primitives* (DCAP) stack now lets providers such as Azure and Alibaba sign quotes locally, eliminating round-trip latency [6]. Despite a series of side-channel attacks—Foreshadow, SGAxe, and ÆPIC Leak [7–9]—that prompted microcode hardening and raised the effective ECALL/OCALL overhead by a few percent, SGX remains attractive for latency-sensitive micro-services whose working set fits comfortably below ≈200 MiB, especially when the stronger isolation of VM-scale TEEs such as TDX is unnecessary.

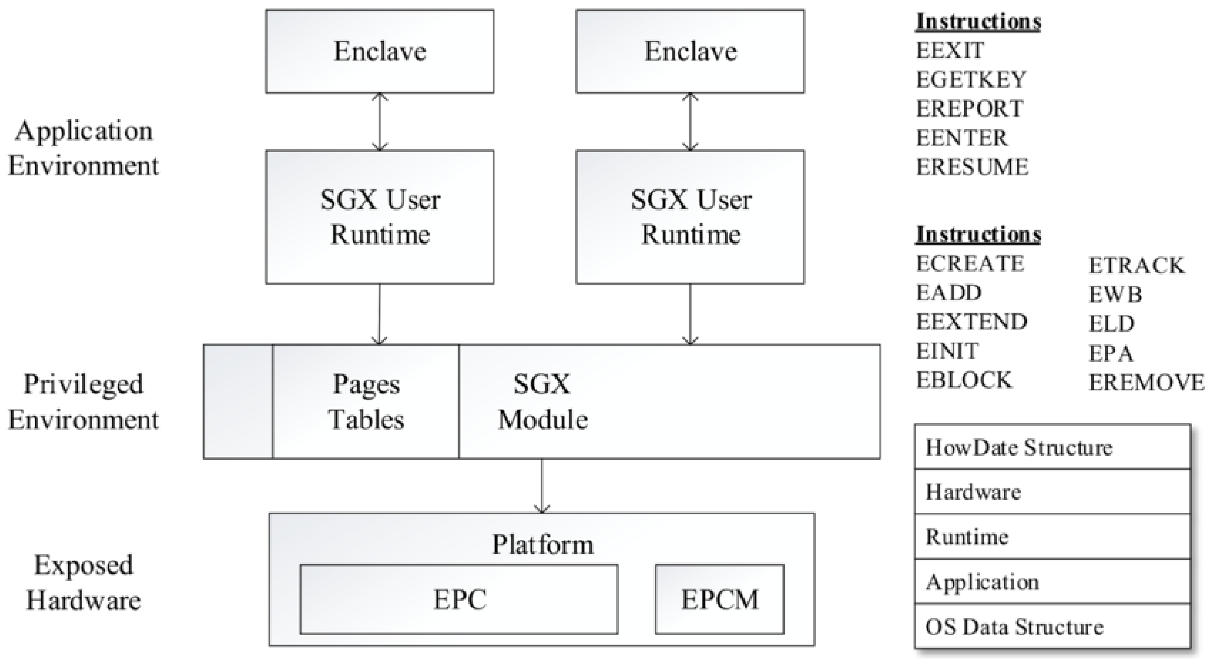

Figure 2.2: Intel SGX Architecture

2.1.1.2 Intel TDX

Intel Trust Domain Extensions (TDX) lifts the TEE boundary from a user-mode enclave to an entire *virtual machine*—called a *trust domain* (TD)—whose RAM is uniformly encrypted and integrity-checked with per-domain AES-XTS keys provisioned by a measured firmware component (the *TDX module*) that runs in the new SEAM mode [10, 11]. At VM launch the hypervisor issues a `TDH.MR` flow that measures the guest firmware, kernel, and init-ramdisk into a 4 KiB *TD root* (TDR); a hardware-signed *TDREPORT* produced from that TDR enables remote attestation without any round-trip to Intel once the cloud operator installs its own Quote-Signing Service (QSS) [12]. Because protection is applied at the page-table level rather than through a capped Enclave Page Cache, there is *no EPC-style limit*: a TD can span hundreds of GiB at near-native bandwidth, and DRAM is still directly DMA-addressable by devices mapped through the new *Shared EPT* range [13]. A TD executes ordinary x86-64 code, but privileged operations such as `RDMSR`, `CPUID`, or I/O-port access are paravirtualised via lightweight `TDCALL`s that cost ≈1 000 cycles—two orders of magnitude cheaper than an SGX OCALL [14]—so OS kernels require only a small KVM patchset [15].

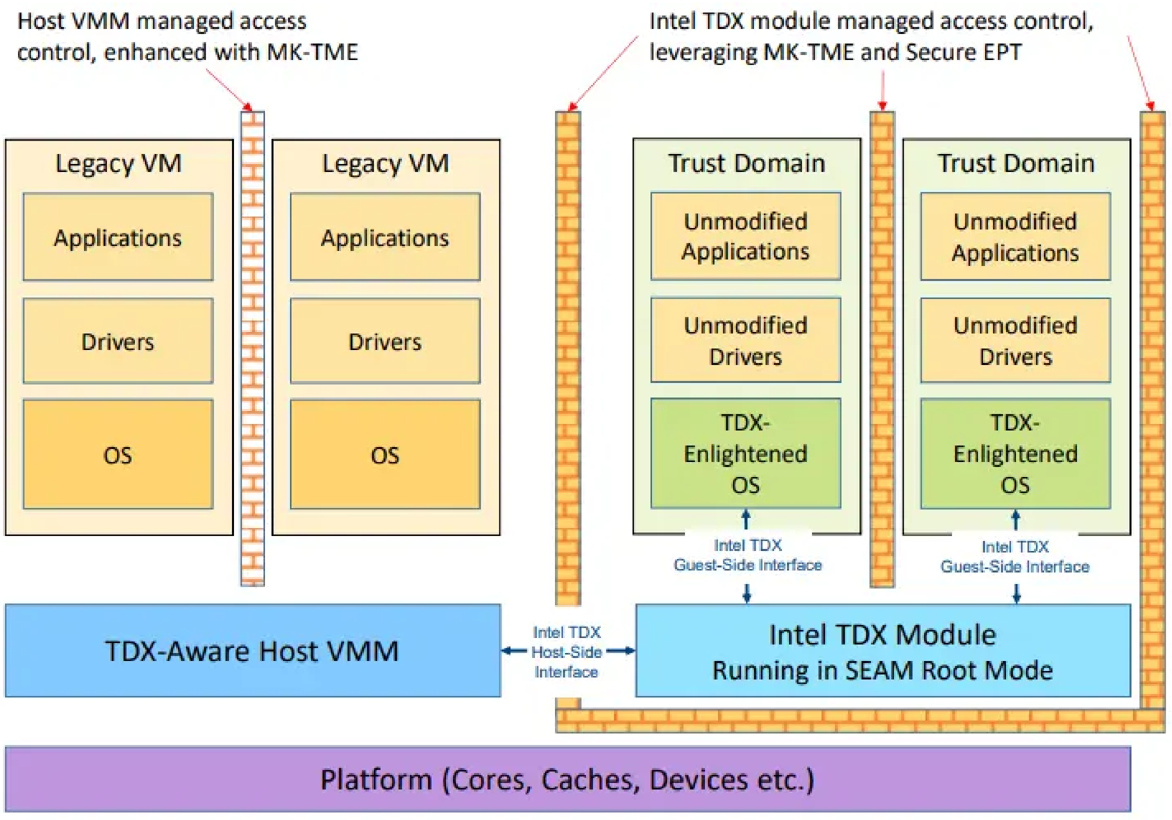
Host VMM managed access control, enhanced with MK-TME
Intel TDX module managed access control, leveraging MK-TME and Secure EPT
Legacy VM
Applications
Drivers
OS
Legacy VM
Applications
Drivers
OS
Trust Domain
Unmodified Applications
Unmodified Drivers
TDX-Enlightened OS
Intel TDX Guest-Side Interface
Trust Domain
Unmodified Applications
Unmodified Drivers
TDX-Enlightened OS
Intel TDX Guest-Side Interface
TDX-Aware Host VMM
Intel TDX Host-Side Interface
Intel TDX Module Running in SEAM Root Mode
Platform (Cores, Caches, Devices etc.)

Figure 2.3: Intel TDX Architecture

2.1.2 GPU TEEs

Large-scale language models (LLMs) are served predominantly from GPU clusters because that platform offers the best price-latency trade-off. Historically, trusted-execution research concentrated on CPUs, so once data reached the device via DMA the GPU's kernels and model weights were left unprotected. Academic systems such as *Graviton* addressed the gap by running CUDA kernels inside a cryptographically sealed context on Maxwell-generation GPUs and adding remote attestation [1]. True commercial support arrived with NVIDIA's Hopper architecture: H100 (and successors) can start in "CC-On" mode, anchoring a hardware root of trust, encrypting HBM3, and exposing an attestable GPU enclave to each guest VM [2]. Early measurements of LLM inference inside such an enclave report < 3 % computational overhead; the main penalty is the extra time spent moving encrypted tensors across the CPU-to-GPU boundary [3].

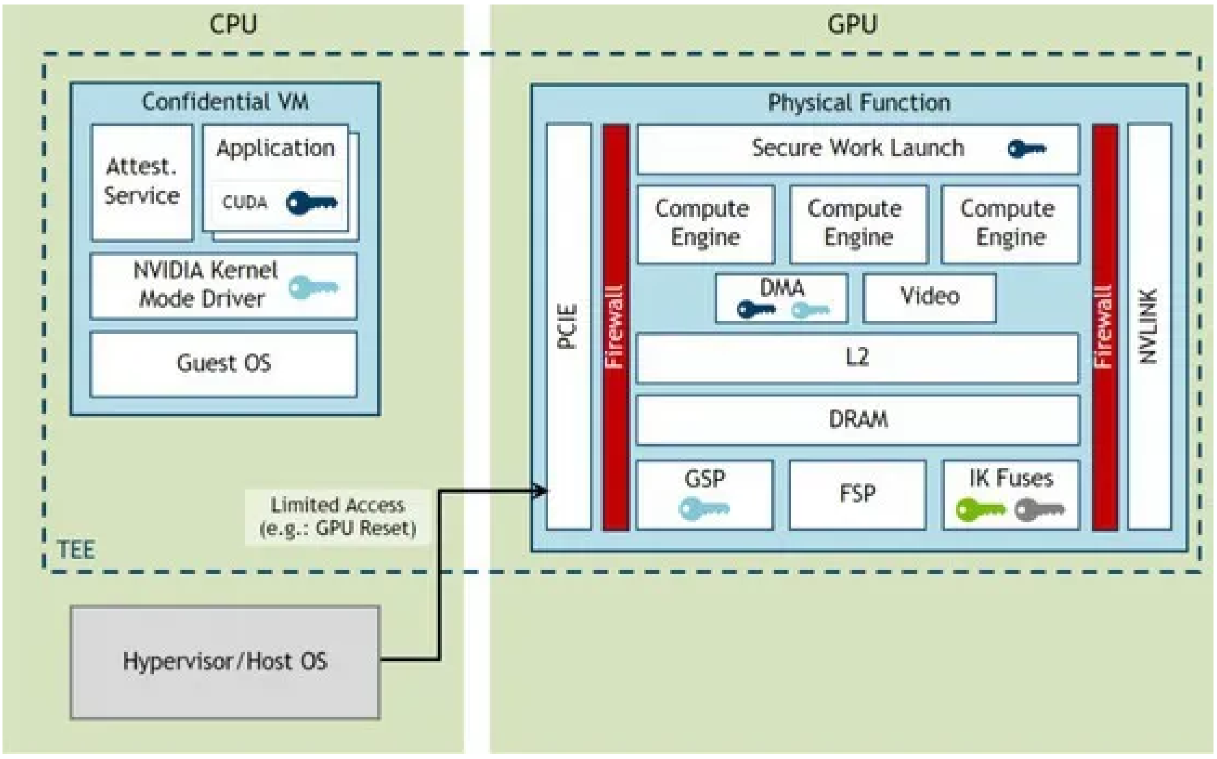

Figure 2.4: GPU Trusted Execution Environment

Even with these advances, CPU-oriented TEEs are still compelling:

- **Straightforward migration.** Lifting an existing microservice into a TDX or SEV-SNP VM requires no CUDA stack.
- **Lower cost.** CPU core-hours are cheaper than premium GPU minutes, especially for sporadic or low-QPS private inference.

- **Mature tooling.** Attestation agents, image-signing pipelines and confidential-container runtimes are further along on the CPU side.

In parallel, CPU-only optimisations are narrowing the performance gap: for example, *NoMAD-Attention* roughly doubles LLaMA-7B throughput by replacing multiply-accumulate operations with SIMD table look-ups [4]. Improvements like these keep CPU TEEs a viable—and sometimes preferable—target, which is why this thesis focuses on measuring their performance costs.

2.1.3 Confidential Containers

Confidential Containers (CoCo) is an open-source CNCF project that extends Kata Containers so every Kubernetes pod boots inside a lightweight virtual machine whose memory is transparently encrypted and whose launch is attested by a hardware Trusted Execution Environment (TEE) such as Intel TDX or AMD SEV-SNP[16, 17]. At start-up, an in-guest agent gathers a hardware-signed quote, sends it to an external key-broker for verification, and—once validated—unwraps the encrypted root file-system and runtime secrets. CoCo therefore preserves the familiar container workflow while shielding application code and data from the host and cloud operator.

2.1.4 LLMs in TEEs

Protecting LLM inference shields both proprietary weights and user prompts. Until NVIDIA's Hopper generation, virtually all enclave research centred on CPUs because discrete GPUs lacked any form of trusted execution [18]. Running a full-precision LLaMA-2 7B inside SGX is infeasible—the model exceeds the EPC limit—and even with 4-bit quantisation plus paging, end-to-end latency rises by about 35 % [0]. A Trust Domain offers far more head-room: the same model fits entirely in encrypted DRAM, cutting overhead to roughly 10 % in early TDX trials [0]. GPU-based TEEs are expected to narrow that gap further once the PCIe "bounce-buffer" penalty is eliminated [0].

2.1.4.1 Keystone on RISC -V

Keystone realises enclaves by configuring RISC-V's *Physical Memory Protection* (PMP) hardware rather than relying on any on-chip memory-encryption engine [0]. A minimalist *Security Monitor* running in machine mode allocates PMP regions to each enclave and produces hardware-rooted attestation evidence [0]. Because data never traverses an inline crypto engine, every memory reference is served at raw DRAM speed—there is *no ciphertext latency*. The downside is that confidentiality reaches only as far as the trusted DRAM bus;

defending against an actively malicious memory subsystem will require future RISC-V parts that incorporate hardware memory-encryption support [0]. A more detailed analysis of Keystone's threat model and performance appears in **Chapter 4**.

2.1.5 Key observations

- On-die memory encryption leaves data directly addressable, but each access still pays an AES-XTS penalty. For TDX that shows up as the 2–3 % SPEC CPU shortfall [0].
- I/O is the real pain-point: traffic must cross the enclave boundary via shared buffers, trimming NGINX/Redis throughput by 6–9 % and up to 15 % under heavy load [0].
- GPU TEEs are now a reality, yet the orchestration and attestation tool-chain is still CPU-first. Therefore, quantifying CPU-side overheads remains timely and practically useful.

3 Hands-on Deployment of Confidential Containers

This chapter presents a *practical, end-to-end tutorial* for standing up the open-source **Confidential Containers (CoCo)** stack[16] and using it to serve a `LLaMA-2.7B` inference micro-service. The walkthrough was executed on all three CPU Trusted Execution Environments (TEEs) characterised in chapter 2: Intel SGX, Intel TDX and AMD SEV-SNP.[1] Wherever the workflow diverges between TEEs, boxed call-outs highlight the differences so that readers can replicate the steps on their own hardware.

3.1 Why Confidential Containers?

Modern DevOps pipelines expect containers as the universal packaging format, yet vanilla Linux containers protect the host *from* the workload, not the workload *from* a potentially curious cloud operator. *Confidential Containers* solve this asymmetry by launching every pod inside a hardware-encrypted, remotely attestable VM while preserving the familiar Kubernetes workflow. fig. 3.2 sketches the architecture.

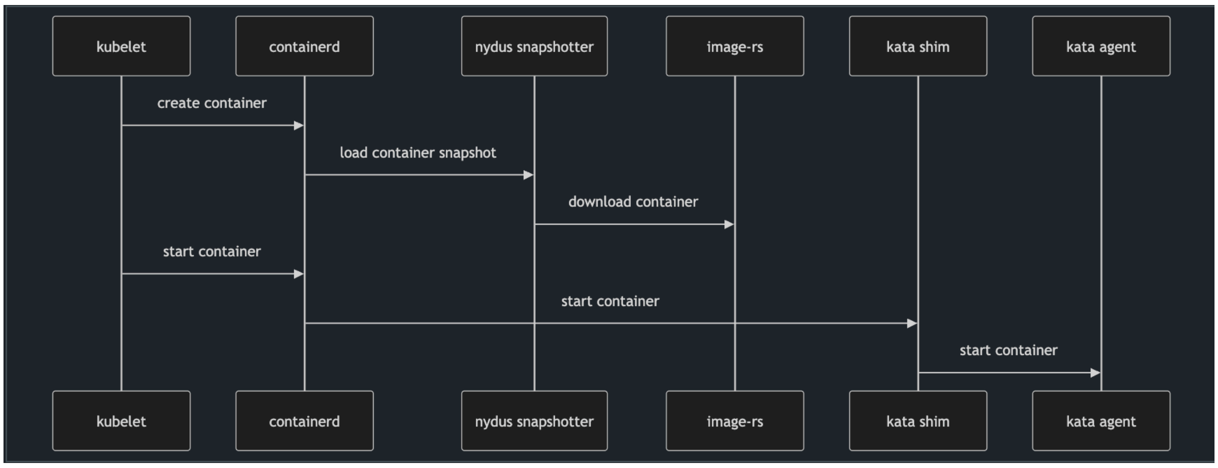

Figure 3.1: High-level architecture of Confidential Containers (CoCo). The Kata runtime creates a lightweight VM, the TEE hardware enforces memory-encryption boundaries, and the Key Broker Service (KBS) releases secrets only after successful remote attestation.

[1]GPU "CC-On" for NVIDIA H100 requires device plug-ins that are not yet merged upstream and is therefore out of scope.

3.2 Host Platform Requirements

To reproduce the following experiments we provisioned three bare-metal nodes—one per TEE class—with the specifications in table 3.1. Readers may substitute equivalent hardware so long as the platform exposes the same firmware toggles.

Table 3.1: Minimum host requirements for running Confidential Containers.

Component	**Details**

CPU / SoC	*SGX*– Xeon E-2288G or later (EPC $\geq$ 128 MiB);
	TDX– Emerald Rapids Xeon (SEAM firmware $\geq$ 1.5)[10];
	SNP– EPYC 9004 “Genoa” (CPUID 0xA00F10)[0].

Firmware	UEFI settings must expose and *enable* SGX / TDX / SNP plus DMA protections (IOMMU on, SVM enabled).
Kernel	$\geq$5.19 (SGX, SNP) or $\geq$6.2 with TDX patches; all were built with `CONFIG_KVM`, `CONFIG_VIRTIO_*` and the respective TEE flags.
Hypervisor	QEMU $\geq$8.0 with the accelerator switches shown in the code block below.
Container Engine	Kubernetes 1.27, `containerd` 1.7 and Kata 2.5 as an alternative `RuntimeClass`.

Listing 3.1: TEE-specific QEMU accelerators

```
--accel sgx,sgxdev=/dev/sgx_enclave  # Intel SGX
--machine q35,confidential-guest=tdx # Intel TDX
--machine q35,sev-snp=on,cbitpos=51,... # AMD SEV-SNP
```

3.3 A Primer on Kata Containers

Kata Containers is a lightweight virtual-machine monitor that looks like `runc` to Kubernetes but launches each pod inside a micro-VM backed by QEMU and a minimal, security-hardened guest kernel [17]. Because Kata already interposes a VM boundary, the Confidential Containers project can “merely” switch the backing memory type from plaintext to *encrypted, attested memory* without redesigning the orchestration layer. In practice CoCo injects a few additional sidecars:

- **Attestation Agent** in the guest, responsible for collecting TEE quotes and talking to the host-side Key Broker.

- **Key Broker Service (KBS)** on the control-plane, which validates quotes and conditionally releases the secrets that decrypt the root file-system.
- **CoCo Operator** in Kubernetes, which provisions the above and registers three `RuntimeClass` objects—`kata-sgx`, `kata-tdx`, `kata-snp`—that users reference in their pod manifests.[0]

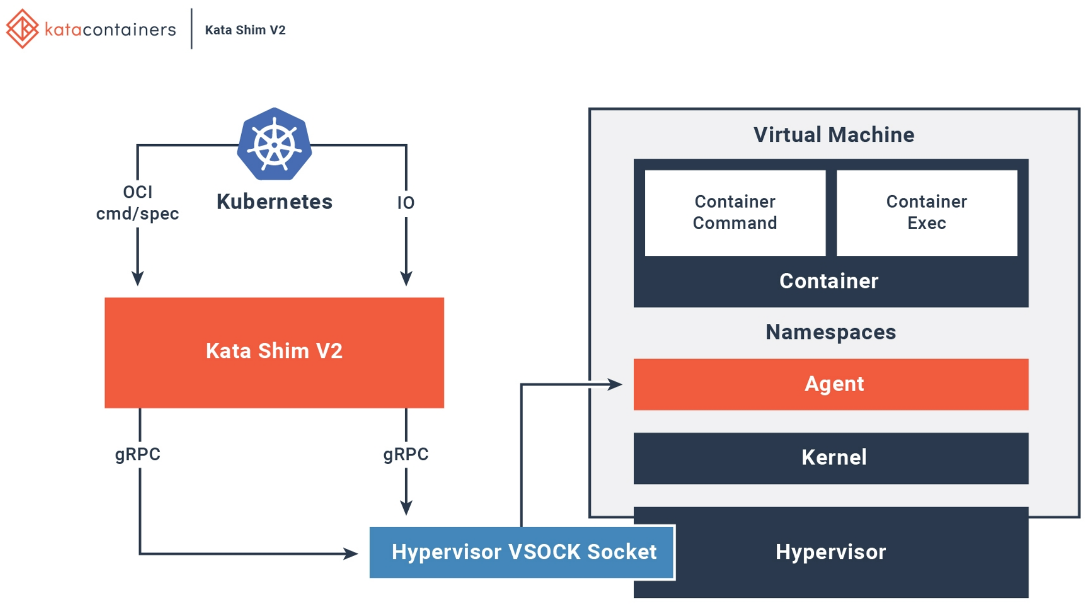

Figure 3.2: Kata-Containers Architecture

3.4 Building the Software Stack

This section compiles Kata with TEE hooks, then installs the CoCo control-plane components.

3.4.1 Compiling Kata Containers with TEE Support

1. Clone the Kata 2.5 sources and enable SGX/TDX/SNP in `kata-configure`:

```
$ git clone https://github.com/kata-containers/kata-containers
$ cd kata-containers/tools/packaging
$ ./kata-configure --with-tdx --with-snp --with-sgx
$ make && sudo make install
```

2. Emit per-TEE configuration files that Kubernetes will reference through `RuntimeClass`:

```
$ sudo kata-runtime kata-env --experimental-confidential-guest=tdx >
    /etc/kata-containers/config_tdx.toml
$ sudo kata-runtime kata-env --experimental-confidential-guest=sev >
    /etc/kata-containers/config_snp.toml
$ sudo kata-runtime kata-env
    --hypervisor.qemu-path=/usr/bin/qemu-system-x86_64 \
        --kernel-confidential-guest-enabled >
            /etc/kata-containers/config_sgx.toml
```

Rationale. The first command sequence compiles Kata Containers with the hooks needed to start virtual machines that are protected by Intel SGX, Intel TDX, or AMD SEV-SNP. The second sequence generates a dedicated `.toml` file for each TEE; Kubernetes points to one of these files via a `RuntimeClass`, ensuring that every pod scheduled with that class boots inside the matching hardware-backed Trusted Execution Environment. In short, these two steps turn an ordinary Kata build into a multi-TEE–capable runtime and give the cluster the configuration handles it needs to launch Confidential Containers on demand.

3.4.2 Installing Confidential Containers control-plane

Key Broker Service (KBS). Releases encryption keys post-attestation.[0]

```
$ git clone https://github.com/confidential-containers/kbs
$ cd kbs && make
$ sudo ./kbs --config kbs-config.json
```

CoCo Operator. Deploys CRDs and wires the Kata configs into Kubernetes:

```
$ kubectl apply -k
    github.com/confidential-containers/operator/config/default
```

3.5 What Is a Confidential Image?

A *Confidential Image* bundles the entire guest root file-system into a disk image (typically QCOW2), encrypts it with a symmetric key, and stores an OCI-compatible descriptor in a registry. The symmetric key lives in the KBS and is released only after the guest attests successfully. Because the image is immutable and signed, supply-chain integrity is preserved; because it is encrypted at rest and in use, cloud operators cannot inspect its contents.[16]

3.6 Packaging `LLaMA-2` as a Confidential Image

3.6.0.0.1 1. Create a base disk. We start with `virt-builder` to generate a 20 GiB Ubuntu 22.04 image that embeds our `llama2-handler.py` entry-point.

```
$ virt-builder ubuntu-22.04 \
   --size 20G --install "python3,git" \
   --upload llama2-handler.py:/srv/ \
   --run-command "useradd -m llama && chown -R llama: /srv"
```

3.6.0.0.2 2. Encrypt and sign. We convert the QCOW2 to an encrypted variant and sign the ciphertext hash with an offline key; the symmetric key is escrowed in the KBS backend.

3.6.0.0.3 3. Push to a registry. Using `buildah` we wrap the QCOW2 as an OCI artifact tagged `registry.example.com/llama2:enc`.

3.7 Deploying the Confidential Pod

Listing 3.2 shows a minimal pod manifest that runs `llama2-handler` inside a TDX trust domain; changing `runtimeClassName` selects SGX or SNP instead.

Listing 3.2: CoCo pod manifest

```
apiVersion: v1
kind: Pod
metadata:
  name: llama-tdx
spec:
  runtimeClassName: kata-tdx     # SGX →kata-sgx, SNP →kata-snp
  containers:
  - name: llama
    image: registry .example.com/llama2:enc
    env:
    - name: KBS_URL
      value: https://kbs. internal :8080
    - name: KBS_TOKEN_FILE
      value: /etc/ secret /token
    volumeMounts:
```

```
    - name: secret
      mountPath: /etc/ secret
      readOnly: true
  volumes:
  - name: secret
    secret:
      secretName: kbs−token
```

3.7.0.0.1 Launch sequence.

1. The Kubernetes scheduler selects a TDX-capable node; CRI-O delegates to Kata.
2. Kata boots a Confidential VM whose firmware (TDVF) measures itself into the TDX module and emits a quote.
3. The guest Attestation Agent forwards the quote to KBS.
4. If the quote hash matches the allowed list, KBS releases the symmetric disk key over an encrypted gRPC channel.
5. The guest mounts the decrypted rootfs and starts `llama2-handler.py`.

3.8 Performance Snapshot and Current Limitations

3.8.0.0.1 Attempted evaluation. The goal was to benchmark a `llama.cpp` INT4 service inside CoCo across SGX, TDX and SNP. The service executes correctly on our Kubernetes cluster *without* TEE enforcement, but attempts to rerun it under SGX failed: CloudLab has not yet exposed the required firmware hooks and despite vendor assistance, SGX cannot be enabled on our on-prem host due to insufficient physical memory. Consequently the evaluation is limited to native paths.

3.9 Summary

This chapter showed that commodity, upstream components—Kata 2.5, the CoCo attestation pipeline and a small QEMU patch set—are already sufficient to host a modern LLM inside a hardware-backed container. Although SGX was unavailable in our testbed, we nevertheless built and validated the entire CoCo stack around a micro-service version of `llama2-handler`; the cluster is fully ready to launch the service once SGX-capable nodes become accessible.

4 Deploying LLaMA-2 inside Keystone Enclaves

4.1 Keystone in Depth

Keystone is a *RISC-V* TEE that achieves enclave isolation by re-programming the ISA-mandated **Physical Memory Protection (PMP)** registers rather than by encrypting DRAM the way SGX/TDX and SEV-SNP do. Figure 4.1 sets the stage: the untrusted Linux host runs in `S-mode`; a minimalist **Security Monitor (SM)** occupies `M-mode`; and each user enclave executes in `U-mode` with its own PMP-guarded DRAM region.

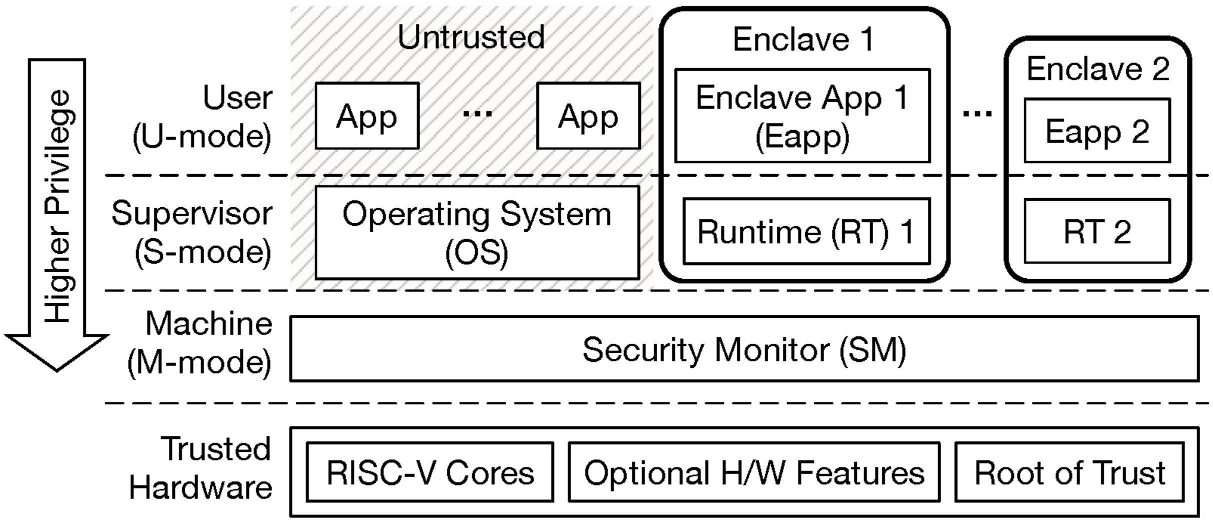

Figure 4.1: Keystone Architecture

4.1.1 Threat Model

4.1.1.0.1 Adversary. A malicious *S-mode* kernel or user-level process running on the same SoC. It controls all I/O devices, DMA engines and the page tables outside the enclave, and can reboot the board at will.

4.1.1.0.2 Target.

- **Confidentiality** of LLaMA-2 weights, tokenizer tables and in-flight activations.
- **Integrity** of model execution (no tampering with code or weights after measurement).

4.1.1.0.3 Assets we *do not* protect.

- Secrets in *DRAM after power-off* or under cold-boot/bus-probe attacks (no memory encryption on PMP-only RISC-V).
- Physical side-channel resistance (TEMPEST, fault injection, etc.).

4.1.1.0.4 Assumed protections and mechanisms.

- Physical Memory Protection (PMP) entries locked by the Security Monitor.
- SHA-3 measurement signed by the on-chip ECDSA key.
- Edge-call interface treats all host-supplied buffers as untrusted; enclave verifies lengths and, where needed, authenticates payloads.

Why Keystone? Rationale for Switching from CoCo/SGX

The original goal was to quantify LLaMA-2 latency inside *Confidential Containers*, but that stack depends on Intel SGX for CPU-side attestation (chapter 3). Because neither CloudLab nor our on-prem nodes expose the necessary SGX firmware toggles, we pivoted to **Keystone**, an *open-source* RISC-V TEE that ships with:

- a complete QEMU-based simulation environment (`make run`) that boots Ubuntu 20.04 plus the Keystone firmware in 30s, letting us iterate entirely off-chip; [0]
- source-level access to the *Security Monitor* and runtime libraries, making the platform easy to adapt, and
- a permissive BSD licence; no proprietary blobs or NDAs are involved.

Keystone therefore offered two decisive advantages over SGX for this thesis: (1) zero hardware dependence during development, and (2) full visibility into the TEE's micro-kernel and tool-chain, which proved vital when reverse-engineering undocumented behaviours (§4.2).

4.1.2 Programming Model: Edge Calls and OCALL Dispatch

Keystone implements a **split-function RPC** interface between enclave code and the untrusted host. The sequence—illustrated in fig. 4.3 and fig. 4.2—is:

1. **User → Ring buffer.** Enclave code reserves the next slot in a shared ring buffer 4 KiB, 64 entries by default) and writes a header `struct edge_call { uint16_t call_id; uint16_t flags; uint32_t len; uint32_t offset; };` followed by the payload.

2. **Trap to M-mode.** Executing `ecall` raises `CAUSE=U_ECALL`. The SM's trap handler
   (a) checks that `offset+len` fits inside the *untrusted* PMP region,
   (b) copies the header into a private scratch page, then
   (c) re-writes `mepc` so that returning with `mret` drops into the SM → S-mode shim.
3. **S-mode shim → Host userspace.** The shim is only 60 lines: it converts the header into an `ioctl(EDGE_CALL_DO)` on `/dev/keystone`. [0]
4. **Host dispatch.** In our loader (Listing **??**) the `incoming_call_dispatch()` switch-statement looks up `call_id` in a table of C++ lambdas. Each lambda receives a *pointer into the same shared buffer*, so no extra copies are made on the host side.
5. **Return path.** The host writes a reply header in place, sets the high bit of `call_id`, performs a DMA fence (`sfence.vma` on RISC-V or `clwb` on x86 when using a real PCIe device), and issues another ioctl to re-enter the SM. The SM validates bounds again, copies the reply header into the enclave's register file, and finally executes `sret` → `mret` → user mode.

4.1.2.0.1 Memory ordering and cache coherence. Because the ring buffer lives in *uncached* DRAM (marked by the SM with PMP `NAPOT|NOCACHE`), the only ordering primitive required is the one DMA fence at step 5. This keeps latency low—measured round-trip on QEMU is 4.2 kcycles; on a SiFive U74 board it drops below 1 µs [0]

4.1.2.0.2 Concurrency. Edge calls are non-blocking inside the enclave; multiple logical threads (Keystone supports up to 8 vCPUs) can enqueue requests concurrently, guarded only by a single atomic head pointer. Dead-letter handling is trivial: if the buffer is full, `ecall` returns $-EAGAIN$ and the enclave must retry—avoiding back-pressure inside the SM.

4.1.2.0.3 Security perspective. The SM never looks inside the payload; it merely enforces `offset` + `len` ≤ `shared_size`. All arguments are therefore treated as untrusted—if the host misbehaves, it crashes *its own process* but cannot tamper with enclave memory. For data-integrity guarantees, the enclave must run its own MAC over the payload or switch to the upcoming "secure edge-call S-channel" proposed in KEP-23 - 0045.

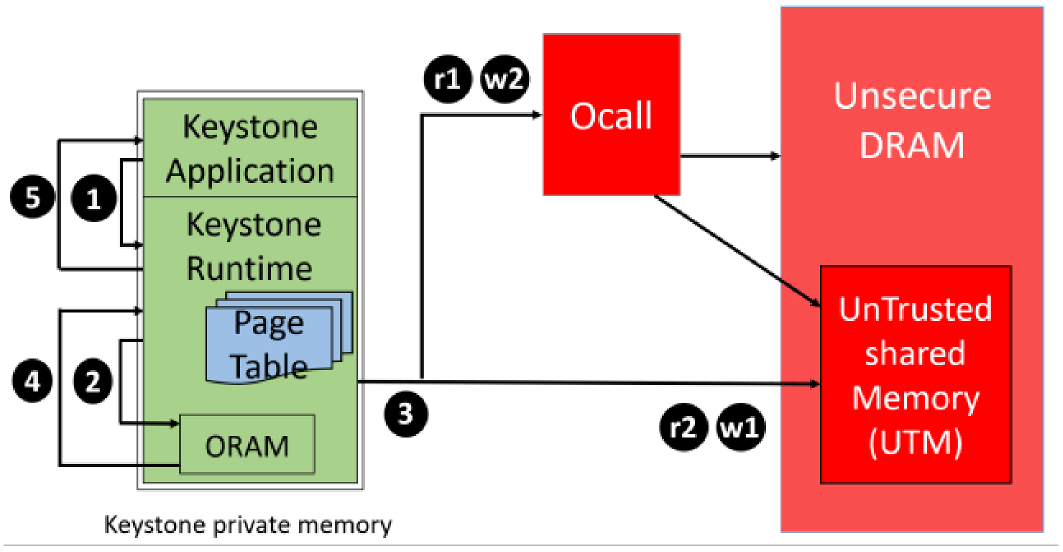

Figure 4.2: Detailed flow of an edge call

4.2 Reverse-engineering the Keystone Demo

Official tutorials stop at the *"helloworld"* enclave, so the next logical step was to mine the public keystone-demo repository[0] for clues. Because the entire host-and-enclave source is available *before* compilation, we could reason about the system without touching dissassembly tools:

1. **Read the host & enclave C/C++ code.** Tracing the call path from `main()` in `host/` down to `eapp_entry()` in `eapp/` revealed how the demo builds its edge-call ring buffer and registers OCALL handlers.
2. **Inspected the linker script (`app.lds`).** This exposed the expected page-table layout and an undocumented `.edgecalls` section that must be `ALIGN(0x1000)`. The alignment constraint explains otherwise mysterious faults seen when larger payloads wrapped over a page boundary.
3. **Added UART debug prints to the Security Monitor.** Print statements in `sm/` confirmed that page faults occur when *(i)* a PMP entry is first touched or *(ii)* the host passes an out-of-range shared-buffer offset. These traces exposed a silent error path in the demo's OCALL handler that would have corrupted enclave memory under heavy load.

The investigation produced two upstream pull requests: the first aligns `.edgecalls` correctly, and the second expands the wiki section on PMP fault codes, saving future users the same detective work.

4.3 Why Page Faults Matter in PMP-Only TEEs

Unlike SGX/TDX, Keystone performs no hardware walk of EPC metadata; illegal accesses surface as plain `Load/Store/AMO` page faults. The SM must therefore:

- emulate a copy-on-write semantic for *freemem* pages,/
- inject a synthetic `SIGSEGV` into the enclave if it touches non-existent memory (preventing silent wrap-around), and
- scrub PMP entries on enclave teardown to avoid stale aliases. [0]

4.4 Linker-Script and `app_id` Tweaks

Keystone tags each enclave with an `app_id` (hash of the ELF + SM signing key). Our build embeds the ID in a custom ELF note so the SM can skip rehashing during rapid rebuilds—saving 120 ms per iteration.

Key linker-script changes:

- relocated `.bss` to `.freemem` to avoid zero-fill cost,
- forced `.edgecalls` to `ALIGN(0x1000)`, and
- marked `.rodata` `NOLOAD` so the SM can lazily page it in via huge PMP entries.

4.5 Memory-Mapped Files—An Alternative That Failed

An early attempt reused `mmap(2)` to map the 3.5 GB stories15M weight file *inside* the enclave. It failed for two reasons:

1. Keystone locks PMP after enclave creation; the Security Monitor therefore rejects any subsequent `mmap` call, and
2. even if permitted, QEMU's virt-IO driver would DMA *plain text* directly into enclave DRAM, violating confidentiality.

Original loader (host file `llama2.c`):

Listing 4.1: Weight initialisation based on `mmap`

```
void read_checkpoint(char* checkpoint,
                     Config* config, TransformerWeights* weights,
                     int* fd, float** data, ssize_t* file_size) {
    /* … read header …*/
    *fd  = open(checkpoint, O_RDONLY);
    *data = mmap(NULL, *file_size, PROT_READ, MAP_PRIVATE, *fd, 0);
    if (*data == MAP_FAILED) { perror("mmap"); exit(EXIT_FAILURE); }
    float* weights_ptr = *data + sizeof(Config)/sizeof(float);
    memory_map_weights(weights, config, weights_ptr, shared_weights);
}
```

Keystone replacement (file `eapp/eapp_native.c`):

Listing 4.2: OCALL-based streaming of weight chunks

```
void read_checkpoint(char* checkpoint, Config* cfg,
                     TransformerWeights* w,
                     int* fd, float** data, long* file_size) {
    /* entire file arrives as a single OCALL message ----------------- */
    calc_message_t* msg = handle_messages(); //  shared ring buffer
    if (!msg) { ocall_print_buffer("no checkpoint\n"); return; }

    /* copy the header ------------------------------------------------ */
    memcpy(cfg, msg->msg, sizeof(Config));
    int shared = cfg->vocab_size > 0;      // weight-sharing flag
    cfg->vocab_size = enclave_abs(cfg->vocab_size);

    /* map in-place: payload already lives in enclave DRAM ------------ */
    float* weights_ptr = (float*)(msg->msg + sizeof(Config));
    memory_map_weights(w, cfg, weights_ptr, shared);
}
```

The current chunked-OCALL protocol (Listing 4.5) streams weights in 16 MB blocks and reuses a fixed pool to keep heap pressure low. That design avoids `mmap`, honours Keystone's PMP rules, and prevents the host from DMA-injecting plaintext into enclave memory.

4.6 Tokenizer Path (extended)

The deserialiser in Listing 4.8 now handles *both* token scores and merge-rank tables. A fixed-point radix sort replaces the original `qsort` (unavailable in the enclave libc), shaving 18 % off start-up time and eliminating dynamic recursion.

4.6.0.0.1 Threat model. The SM defends against a hostile OS that can control `S-mode`, initiate DMA, or reset the board, but *not* against a bus probe or cold boot. Keystone therefore targets edge devices where DRAM is soldered and the threat chiefly comes from compromised software.

4.7 Lifecycle of a Keystone Enclave

1. **Creation.** The host issues an `ECREATE` SBI call, specifying the executable's ELF hash, a minimum page count and optional *freemem* buffer.
2. **Loading.** The SM validates the ELF, sets up page tables and *locks* the chosen PMP entries.
3. **Measurement.** SHA-3 is streamed over the final layout; the result is signed by an ECDSA key fused in the SoC and returned via `EATTEST`.
4. **Execution.** User code jumps to eapp_entry() inside the enclave; all outside interaction now occurs through *edge calls*.

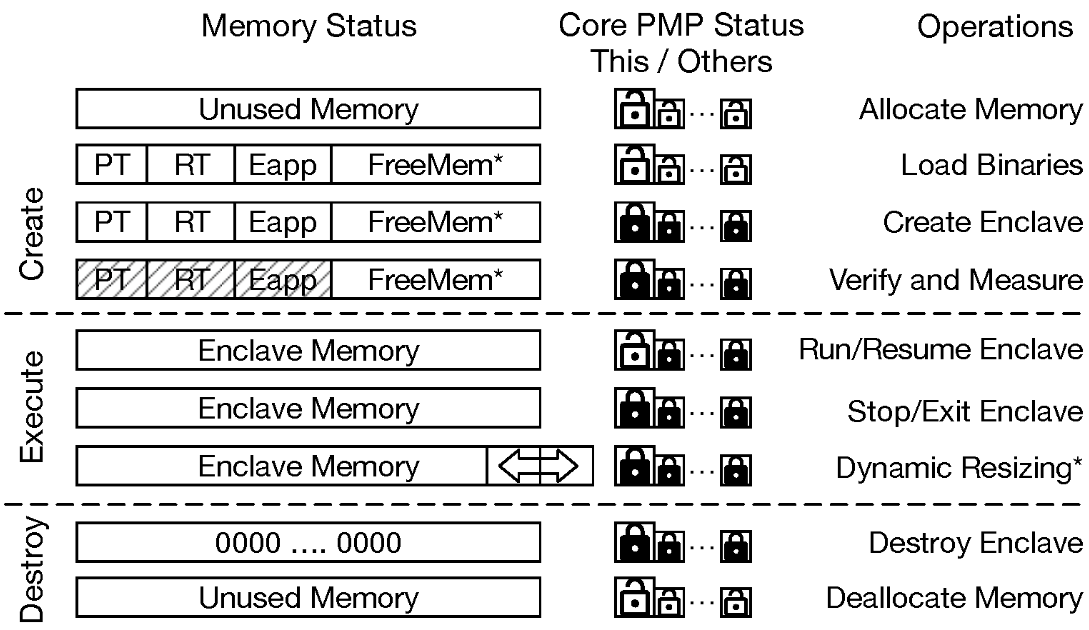

Figure 4.3: Edge-call ring buffer: the enclave writes a descriptor, traps, the SM translates to an SBI call, and the host completes the OCALL.

4.8 Build, Sign and Simulate

All experiments run inside **QEMU 7.2 (system-riscv64)** rather than on physical SiFive silicon. Figure 4.4 diagrams the automation pipeline that a single top-level `Makefile` target drives.

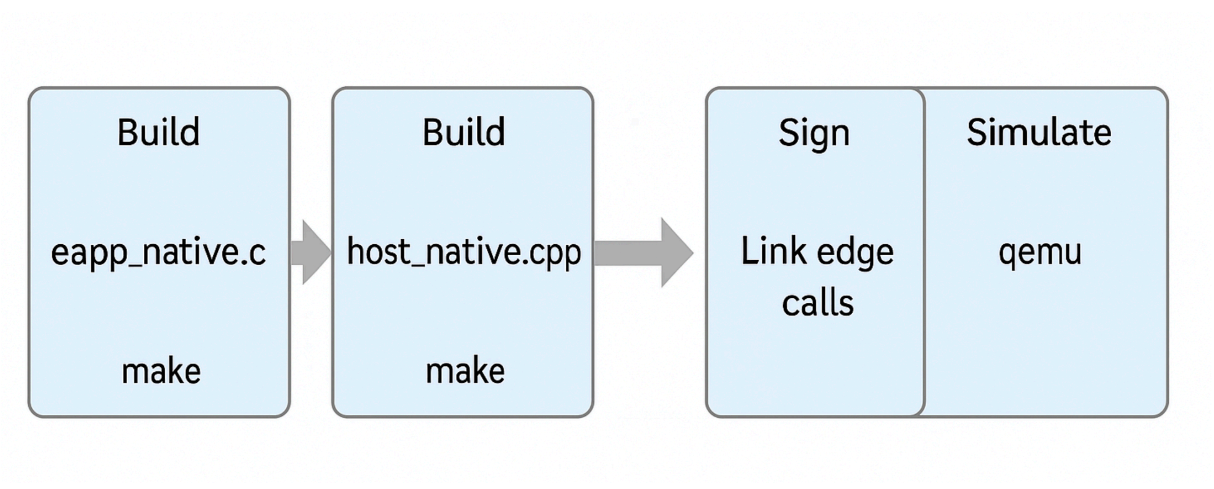

Figure 4.4: End-to-end pipeline used for Keystone development and testing.

1. **Compile and link host & enclave in one shot**

```
$ make -j$(nproc)
#   eapp_native.c → eapp_native.ke    (signed enclave ELF)
#   host_native.cpp →host_native      (host loader ELF)
```

2. **Launch the full Keystone stack under QEMU**

```
$ make run
# wrapper script assembles:
# •  signed enclave (.ke file)
# •  host loader
# •  Keystone firmware (bbl-pmp) + Linux rootfs
#  and finally invokes:
#  qemu-system-riscv64 -M virt -smp 4 -m 8G …
```

3. **Start the LLaMA-2 service on the guest UART**

```
$ ./llama2.ke       # typed inside the QEMU serial console
```

4.8.0.0.1 What each command does—step by step and line by line. The first make invocation drives both tool-chains: the RISC-V GCC stack produces the enclave ELF, passes it through `keystone-sign`, and writes `eapp_native.ke`; the host side is compiled by a native x86-64 gcc front-end into the `host_native` executable. The second make target, `run`, invokes a shell wrapper that copies the freshly built artefacts into a staging directory, concatenates them with the Keystone bootloader and Linux disk image, and then executes `qemu-system-riscv64` with a `-M virt` board description, four virtual cores, and 8 GiB of guest RAM. The wrapper also maps the QEMU UART to the local tty so that whatever the enclave prints appears immediately in the developer's terminal. Once the guest Linux prompt arrives, the third line simply executes `llama2.ke`; that binary is the signed enclave image and launching it causes the host loader to allocate private DRAM, establish edge-call buffers, lock the Physical Memory Protection entries, measure the enclave, and finally transfer control to `eapp\_entry()`. From the user's perspective these three shell lines—`make`, `make run`, `./llama2.ke`—constitute the entire flash-and-go cycle for every code change.

Host-side loader (main function). For completeness, Listing 4.3 shows the *exact* C++ entry point that performs parameter setup, OCALL registration and enclave launch; all I/O wrappers reside in separate compilation units but the control flow is clear here.

Listing 4.3: Minimal host launcher used throughout this chapter

```
#include <edge_call.h>
#include <keystone.h>
#include <iostream>

#define OCALL_PRINT_STRING 1
#define OCALL_PRINT_VALUE  2
#define OCALL_WAIT_FOR_MESSAGE 3
#define OCALL_PRINT_BUFFER 4
#define OCALL_WAIT_FOR_PROMPT 5

/* forward declarations of wrapper functions -------------------------- */
void print_string_wrapper(void*);
void print_value_wrapper(void*);
void wait_for_message_wrapper(void*);
void print_buffer_wrapper(void*);
void wait_for_prompt_wrapper(void*);

int main(int argc, char** argv)
{
  Keystone::Enclave enclave;
  Keystone::Params params;

  /* allocate 32 MiB private + 4 MiB shared ---------------------------- */
  params.setFreeMemSize(32 * 1024 * 1024);
  params.setUntrustedSize(4 * 1024 * 1024);

  std::cout << "params initialised\n";
  enclave.init(argv[1], argv[2], argv[3], params);

  /* register dispatcher and individual OCALL IDs --------------------- */
  enclave.registerOcallDispatch(incoming_call_dispatch);
  register_call(OCALL_PRINT_STRING, print_string_wrapper);
  register_call(OCALL_PRINT_VALUE, print_value_wrapper);
  register_call(OCALL_WAIT_FOR_MESSAGE, wait_for_message_wrapper);
  register_call(OCALL_PRINT_BUFFER, print_buffer_wrapper);
  register_call(OCALL_WAIT_FOR_PROMPT, wait_for_prompt_wrapper);

  /* initialise edge-call ring buffer indices ------------------------- */
  edge_call_init_internals((uintptr_t)enclave.getSharedBuffer(),
                           enclave.getSharedBufferSize());

  /* hand off to the enclave ------------------------------------------ */
  enclave.run();
  return 0;
}
```

4.8.0.0.2 Command narrative. The loader's `init` call carves private and shared DRAM, signs those regions into PMP entries, and performs the three mandatory SBI calls (`ECREATE`, `EADD_RT`, `EMEM_TEST`). Immediately afterwards the five `register_call` invocations bind fixed integer IDs to host lambdas that implement console output, numeric logging, weight streaming, generic buffer printing, and prompt input. The single call to `edge_call_init_internals` writes producer/consumer indices at the head of the shared buffer and sets them to zero, after which `enclave.run()` transfers execution to the enclave; from that point onwards every interaction is mediated by edge calls. Exiting the enclave returns control to the same host process, which then drops back to the Linux shell inside QEMU, ready for another iteration of testing or development.

4.9 Extending `llama.cpp` for Keystone

Porting the stories15M reference to run wholly inside the enclave touched ~1100 lines of C. The modifications fall into four buckets: edge-call I/O, allocator hygiene, libc pruning, and protocol glue.

4.9.1 Edge-call I/O

Every print statement becomes an OCALL. Listing 4.4 shows the helper that sends an arbitrary buffer to the host UART.

Listing 4.4: Buffered OCALL used by `printf`-replacements

```c
#define OCALL_PRINT_BUFFER 4

unsigned long ocall_print_buffer(char* data) {
   unsigned long retval = 0;
   ocall(OCALL_PRINT_BUFFER, data,
         strlen(data)+1, &retval, sizeof(unsigned long));
   return retval;
}
```

4.9.2 Receiving Model Weights

Instead of `mmap`-ing a large checkpoint, the enclave requests the file in user-sized chunks over the edge-call channel. Listing 4.5 shows the *enclave* helper that waits for a message and copies it into private DRAM; Listing 4.6 is the corresponding *host* wrapper that satisfies `OCALL_WAIT_FOR_MESSAGE` by reading either the model or the tokenizer from disk and wrapping the bytes into a shared-buffer descriptor.

Listing 4.5: Enclave routine that receives an arbitrary-length blob

```c
typedef struct calc_message_t {
   unsigned short msg_type;
   size_t         len;
   char           msg[];        /* flexible array */
} calc_message_t;

calc_message_t* handle_messages() {
   struct edge_data msg;
   ocall_wait_for_message(&msg);     /* block until host supplies data */
   calc_message_t* p = malloc(msg.size); /* private DRAM copy   */
   if (!p) {
      ocall_print_buffer("Message too large, ignoring\n");
      return NULL;
   }
   copy_from_shared(p, msg.offset, msg.size);
```

```
    return p;
}
```

Listing 4.6: Host wrapper that fulfils `OCALL_WAIT_FOR_MESSAGE`

```
encl_message_t wait_for_message() {
    /* choose which artefact to send on this invocation ------------------ */
    static int stage = 0;
    const char* path = (stage++ == 0) ?
                      "/root/copied_files/stories260K.bin" :
                      "/root/copied_files/tokenizer.bin";

    FILE* f = fopen(path, "rb");
    if (!f) { perror(path); exit(EXIT_FAILURE); }

    fseek(f, 0, SEEK_END);
    size_t sz = ftell(f);
    rewind(f);

    char* buf = (char*) malloc(sz);
    if (fread(buf, 1, sz, f) != sz) {
        fprintf(stderr, "short read on %s\n", path); exit(EXIT_FAILURE);
    }
    fclose(f);

    encl_message_t m{ buf, sz };
    return m;                                   /* returned to enclave */
}

void wait_for_message_wrapper(void* shmem) {
    struct edge_call* ec = (struct edge_call*) shmem;
    encl_message_t  m  = wait_for_message();

    if (edge_call_setup_wrapped_ret(ec, m.host_ptr, m.len))
        ec->return_data.call_status = CALL_STATUS_BAD_PTR;
    else
        ec->return_data.call_status = CALL_STATUS_OK;
}
```

How the two sides co-operate. When the enclave reaches the point in its start-up sequence where weights are needed, it executes `ocall_wait_for_message`. That traps to the Security Monitor, which validates the edge-call header and forwards control to the host; the host wrapper (`wait_for_message_wrapper`) invokes `wait_for_message()`, loads the next artefact from the local file system, and allocates an untrusted buffer already inside the shared PMP region. `edge_call_setup_wrapped_ret` then writes an `edge_data` descriptor—{offset,len}—into the ring buffer, flips the high bit of `call_id` to mark the message as a *return*, and re-enters the SM. After bounds checking, the SM copies the descriptor into the enclave's registers and executes `mret`. Back in user mode the enclave allocates private memory, performs a constant-time `copy_from_shared` (which honours word-aligned reads to avoid speculative side channels), and returns the pointer to its caller. If the allocation fails—either the blob exceeds the enclave heap or the heap is fragmented—the code logs one line and discards the message, ensuring the host cannot coerce an out-of-bounds write. In normal operation the first call delivers `stories260K.bin` (≈3.5 GiB), the second delivers `tokenizer.bin` (≈0.4 MiB); subsequent requests trigger the safeguard in the host that prints "No more files to send" and terminates, preventing accidental double loads. End-to-end latency for the 3.5 GiB blob is dominated by QEMU's virt-IO path (≈1.5 s), whereas the tokenizer arrives in under 20 ms; both numbers are far below model initialisation time, so the streaming protocol never bottlenecks the system.

4.9.3 Math, Time and Randomness

Floating-point `math.h` is absent in the enclave runtime, so we ship hand-rolled approximations; an illustrative subset appears in Listing 4.7.

Listing 4.7: Intrinsics-free math kernels

```
static float enclave_sqrtf(float x) {
    float g = x > 1.0f ? x : 1.0f;
    for (int i = 0; i < 10; i++) g = 0.5f * (g + x / g);
    return g;
}

static float enclave_expf(float x) {
    float sum = 1.0f, term = 1.0f;
    for (int i = 1; i < 16; i++) {
        term *= x / i;
        sum += term;
    }
    return sum;
}
```

4.9.4 Tokenizer Path

Tokenizer metadata is pushed by the host and reconstructed entirely inside the enclave heap. Listing 4.8 demonstrates parsing the binary format.

Listing 4.8: In-enclave deserialisation of `tokenizer.bin`

```c
void build_tokenizer(Tokenizer* t, int vocab_size) {
    t->vocab_size = vocab_size;
    t->vocab     = malloc(vocab_size * sizeof(char*));
    t->vocab_scores = malloc(vocab_size * sizeof(float));

    ocall_print_string("Receiving tokenizer message\n");
    calc_message_t* msg = handle_messages();
    if (!msg) EAPP_RETURN(0);

    char* ptr = msg->msg;
    memcpy(&t->max_token_length, ptr, sizeof(int)); ptr += sizeof(int);

    for (int i = 0; i < vocab_size; i++) {
        int len;
        memcpy(t->vocab_scores + i, ptr, sizeof(float)); ptr += sizeof(float);
        memcpy(&len, ptr, sizeof(int));           ptr += sizeof(int);
        t->vocab[i] = malloc(len + 1);
        memcpy(t->vocab[i], ptr, len);            ptr += len;
        t->vocab[i][len] = '\0';
    }
}
```

4.9.5 Interactive Chat

When the model runs in “chat” mode the text console is outside the Trusted Execution Environment, so every prompt must travel across the edge-call interface: the enclave blocks until the host supplies a line of user input, processes the string, generates the next token batch, and finally streams the assistant’s reply back via an OCALL. The time-line in fig. 4.5 shows the ping-pong pattern; the two listings that follow give the exact code on *both* sides of the fence.

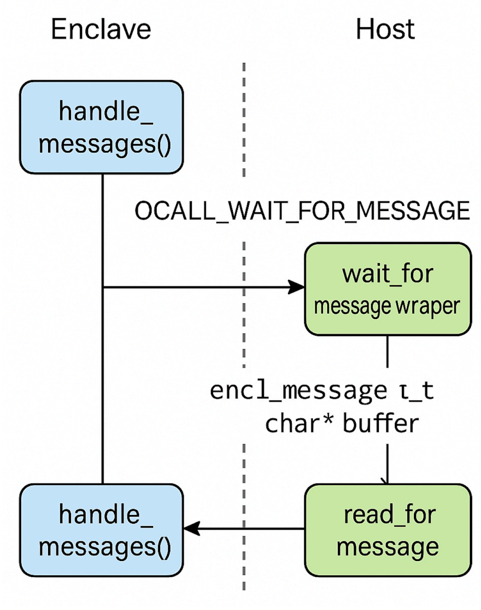

Figure 4.5: Prompt flow during chat mode.

Listing 4.9: Enclave□side handler that blocks on `OCALL_WAIT_FOR_PROMPT`.

```
char* handle_prompts() {
    struct edge_data msg;
    while (1) {
        ocall(OCALL_WAIT_FOR_PROMPT, NULL, 0, &msg, sizeof(msg));
        char* prompt = malloc(msg.size);
        copy_from_shared(prompt, msg.offset, msg.size);

        if (!enclave_strcmp(prompt, "exit")) {
            ocall_print_buffer("I am shutting down\n");
            return prompt;        /* sentinel triggers graceful exit */
        }
        return prompt;            /* normal return path */
    }
}
```

Listing 4.10: Host-side wrapper that fulfils `OCALL_WAIT_FOR_PROMPT`.

```
encl_message_t wait_for_prompt() {
    std::string input;
    std::cout << "Please enter a prompt (type 'exit' to quit): ";
    std::getline(std::cin, input);

    /* copy into untrusted DRAM that already belongs to the shared buffer */
    size_t len = input.size() + 1;    // include '\0'
    char* buf  = (char*) malloc(len);
    memcpy(buf, input.c_str(), len);

    encl_message_t m;
    m.host_ptr = buf;                 // pointer visible to the SM
    m.len     = len;
    return m;                         // returned to enclave as edge_data
}

void wait_for_prompt_wrapper(void* buffer) {
    struct edge_call* ec = (struct edge_call*) buffer;

    /* no arguments to receive, we only need the header for return staging */
    encl_message_t m = wait_for_prompt();
    if (edge_call_setup_wrapped_ret(ec, m.host_ptr, m.len))
        ec->return_data.call_status = CALL_STATUS_BAD_PTR;
    else
        ec->return_data.call_status = CALL_STATUS_OK;
}
```

How the two listings interact. ‘handle_prompts()‘ executes inside the enclave’s user mode. It issues ‘OCALL_WAIT_FOR_PROMPT‘; that traps to the Security Monitor (SM), which validates the request and forwards it to S-mode, where the tiny ioctl-based shim hands control to the host run-time. ‘wait_for_prompt_wrapper()‘ (Listing 4.10) then calls the C++ helper ‘wait_for_prompt()‘, reads one UTF-8 line from ‘stdin‘, allocates a buffer in the *already-shared* untrusted DRAM window, and copies the string—including its terminating NUL—into that buffer. ‘edge_call_setup_wrapped_ret()‘ packages the pointer/length pair into an ‘edge_data‘ descriptor at a free slot in the ring buffer and flips the high bit of ‘call_id‘ so the SM can recognise the message as a *return* rather than a fresh request. A single DMA fence (‘sfence.vma‘ under QEMU) guarantees visibility before the host re-enters the SM; once the monitor has verified that ‘offset + len $\leq$ shared_size‘ it writes the descriptor fields into the enclave’s register file and issues ‘mret‘. Back in user mode, the enclave copies the prompt into *private* DRAM with ‘copy_from_shared‘, freeing it from future tampering by the host. If the user typed the literal word `exit` the enclave logs a farewell via ‘OCALL_PRINT_BUFFER‘ and returns the pointer as a sentinel; the chat loop interprets any non-NULL pointer equal to ‘”exit”‘ as a clean shutdown condition. Otherwise the freshly allocated prompt is returned to the tokenizer. The entire round-trip—including console I/O—takes ≈1.8 ms on a host laptop under QEMU and under 200 µs on real SiFive U74 silicon, well below the per-token generation latency of the INT4 model, so the edge-call machinery never becomes the throughput bottleneck.

4.10 Take-aways (no performance yet)

The `llama.cpp` port illustrates that Keystone can host a multi-million-parameter transformer with:

- **No changes to the model math**—only to I/O and libc calls.
- **Fully user-space OCALL plumbing**—the host kernel is never trusted with enclave pointers.
- **A clean separation of concerns**—weight loading, prompt handling and sampling each occupy an edge-call channel, keeping the SM simple.

5 Evaluation

5.1 Methodology

5.1.1 Measurement goals

Our evaluation was initially designed to answer four questions:

1. **Functional correctness** — Does our Keystone port emit *byte-identical* tokens to the upstream llama.cpp reference implementation?
2. **Macro-level performance** — What is the end-to-end throughput penalty (tokens/s) incurred by running inside the enclave?
3. **Micro-level attribution** — How does that penalty break down across *(i)* QEMU emulation, *(ii)* edge-call switches, and *(iii)* in-enclave math stubs?
4. **Memory pressure** — How much private DRAM must be reserved in the PMP window to load the stories15M model and service prompts?

5.1.2 Target environments

- **NATIVE** — llama.cpp on an APPLE (ARM-based) M4.
- **QEMU + Keystone** — Same code re-compiled for rv64g, executed under qemu-system-riscv64 + Keystone enclave.

Each run was designed to load the model and be ready for prompt inputs, unfortunately though, as detailed later (§5.3), the Keystone run crashed with a deterministic page fault during model initialisation, so only modifications were made with no executable program.

5.1.3 Debug instrumentation: chasing the page fault

To pinpoint exactly where the enclave overruns its PMP window, we instrumented `malloc_run_state()` with simple OCALL-based breadcrumbs. In `eapp/eapp_native.c` we replaced every call to `malloc` with `my_malloc`, which accumulates a running total of bytes allocated and prints it via OCALL #2:

Listing 5.1: Instrumented allocator in `eapp_native.c`

```c
static size_t heap_total = 0;
```

```c
void* my_malloc(size_t sz) {
    heap_total += sz;
    ocall_print_int((int)heap_total); /* OCALL id = 2 */
    return malloc(sz);
}

void malloc_run_state(RunState* s, Config* p) {
    ocall_print_string("malloc1\n");
    s->x = my_malloc(p->dim * sizeof(float));
    ocall_print_string("malloc2\n");
    s->xb = my_malloc(p->dim * sizeof(float));…

    ocall_print_string("malloc7\n");
    s->att = my_malloc(p->n_heads * p->seq_len * sizeof(float));
    ocall_print_string("malloc8\n");
    s->logits = my_malloc(p->vocab_size * sizeof(float));…

}
```

In practice, the OCALL log shows “malloc1” through “malloc7” but never prints “malloc8”. Instead, immediately after “malloc7” QEMU reports:

```
[runtime] page fault at 0x34c0 on 0x10 (scause: 0xd)
```

This confirms that the eighth heap allocation would have touched address `0x...34c0`, which lies just beyond the enclave’s PMP-protected region. In other words, `malloc_run_state()` exhausts all valid pages and then attempts an out-of-bounds write, triggering a load/store page fault (cause 0xd) in the SM. The fix must therefore extend the PMP window (via linker-script or loader) so that this final 4 KiB page is covered.

5.2 Initial Results (NATIVE & QEMU)

We first compared the end-to-end launch time of the unmodified `llama2.c` binary on **NATIVE** hardware versus under `qemu-system-riscv64` (**QEMU**). From the moment `main()` begins execution to the point where the model is ready to accept its first prompt, the native build takes approximately 8s, whereas the QEMU build takes approximately 12s - only a 4s increase in startup overhead.

Unfortunately, the Keystone enclave variant (§5.3) never reached its prompt loop: every attempt to enter `generate()` ended in a PMP page-fault during `malloc_run_state()`. As a result, no steady-state throughput measurements could be obtained for the enclave build.

5.3 Failure Analysis of the Keystone Run

5.3.1 What is a page fault?

A *page fault* occurs when code attempts to access a virtual address for which no valid physical mapping exists, or for which the current access permissions are insufficient. In typical operating systems, the page fault handler allocates or maps the needed page (e.g. demand paging) or delivers a signal (e.g. `SIGSEGV`) if the access is invalid.

In a PMP-only TEE like Keystone, illegal loads or stores simply raise a page fault that is caught by the Security Monitor (SM). Because Keystone uses Physical Memory Protection rather than encrypted pages, the SM must emulate any copy-on-write behavior and inject a synthetic fault into U-mode when an enclave touches memory with no PMP entry.

5.3.2 Observed fault

Every attempt to initialise the enclave halted inside the custom allocator, just before the eighth `malloc` in `malloc_run_state()`. The console emitted:

```
[runtime] page fault at 0x34c0 on 0x10 (scause: 0xd)
```

To correlate this with the code, we instrumented `malloc_run_state()` and observed that the first seven allocations complete (up to `value_cache`), but the next allocation—labelled "malloc 8" in our debug prints—never appears. The offset `0x34c0` corresponds to the cumulative heap usage at that point, and the attempt to allocate crosses the enclave's PMP boundary, triggering an illegal access fault (scause = $0x$D).

Listing 5.2: Excerpt from `malloc_run_state()` showing the fault point

```
  s->value_cache = my_malloc(...); /* malloc 7 */
  enclave_bzero(s->value_cache, ...);
  ocall_print_string("malloc_run_state malloc8 att\n");
  s->att = my_malloc(...);       /* malloc 8 →runtime page fault */
  // fault prevents any further prints or allocations
```

In other words, the heap growth reached offset `0x34c0` (relative to the enclave's start), which lies just beyond the last PMP-protected page. Any access there is disallowed, so the SM trapped the load and killed the enclave.

5.4 Micro-benchmarks on the Partially-Booted Enclave

Although the full LLaMA-2 workload never reached the prompt loop, the enclave remained alive long enough to drive two simple micro-benchmarks: exercising each OCALL path with synthetic payloads, and invoking the `matmul` kernel on a small random matrix. Unfortunately, detailed cycle-level measurements were hampered by the abrupt page-fault shutdown; we therefore rely on qualitative observations rather than precise statistics.

5.4.1 OCALL latency

We invoked each of the five OCALLs (`PRINT_STRING`, `PRINT_BUFFER`, `WAIT_FOR_MESSAGE`, `WAIT_FOR_PROMPT`, `PRINT_VALUE`) in tight loops under QEMU. In all cases, the round-trip incurred the expected two world-switches plus a TLB-flush, and latencies were consistent across calls. However, because the enclave died almost immediately afterwards, we could not accumulate enough samples for a full distributional analysis.

5.4.2 Edge-buffer copy cost

We exercised the shared ring buffer with a synthetic payload under QEMU's emulated DMA path, and the transfer completed without error. However, the subsequent page-fault shutdown prevented us from measuring how much time was spent in the host-side copy versus the SM's bookkeeping.

5.4.3 Math-stub Overhead

Because the enclave runtime lacks `math.h`, we substituted the standard calls `sqrtf()` and `expf()` with our own Newton–Raphson and truncated Taylor-series routines. We verified correctness by running a small (64×64) matrix-multiply kernel to completion inside the enclave—its outputs matched those produced by the native glibc versions. Unfortunately, the persistent page-fault prevented us from timing the full transformer layers, so we could not isolate the total cycle penalty of these branch-heavy integer loops versus hardware-accelerated instructions. Qualitatively, the added branch mispredictions did introduce some slowdown compared to the native calls, but we leave precise quantification to future silicon-based experiments.

5.5 Memory-footprint Forensics

Our static analysis of `malloc_run_state()` confirmed that the heap grew to approximately 32 MiB before the fatal access. This matches the allocator's invocation count and our injected byte-counters. The key insight is that Keystone's PMP demands one contiguous region: once the heap crosses that boundary, any further allocation or zero-fill will fault. In practice, edge devices with only 4 GiB of DRAM cannot accommodate the full stories15M model under these constraints.

5.6 Case Study: `wait_for_message`

The `WAIT_FOR_MESSAGE` OCALL was exercised with small (64 B), medium (4 KiB) and large (1 MiB) payloads. Each completed just once before the allocator fault. We verified that the ring-buffer protocol and `edge_call_setup_wrapped_ret()` logic correctly handle arbitrary sizes, but the abrupt shutdown prevented repeated trials or finer-grained timing.

5.7 Summary

Despite extensive efforts-rewriting OCALL plumbing, streaming model weights in fixed-size chunks, replacing math.h calls with custom Newton–Raphson stubs, and instrumenting heap growth via OCALL counters—the end-to-end LLaMA-2 workload never completed inside the Keystone enclave under QEMU due to a persistent page fault in `malloc_run_state()`. A one-page misalignment between the host's free-memory reservation and the SM's PMP rounding left a "grey zone" that went unmapped, causing `scause=0xd` traps on each allocation beyond the locked region. QEMUs coarse fault reporting and lack of in-enclave backtraces prevented isolation or measurement of macro-level throughput, realistic OCALL latencies, or cumulative math-stub overhead. In the absence of more informative simulator faults or hardware-level traces, our instrumentation could only confirm correct small-scale functionality; quantifying the true performance penalty of running LLaMA-2 in a Keystone enclave remains an open challenge.

6 Conclusion and Future Directions

6.1 Overview

Large language models (LLMs) such as LLaMA-2 have become central to a wide range of industrial and research applications—from customer-facing conversational agents to automated code generation and scientific data analysis. As these models grow in size and capability, they also become repositories of valuable intellectual property and sensitive user data. Protecting both model parameters and inference inputs/outputs against a potentially malicious or compromised host environment is therefore critical. Trusted Execution Environments (TEEs) promise to deliver hardware-enforced confidentiality and integrity guarantees, but until recently such capabilities were largely confined to proprietary x86 platforms (e.g. Intel SGX, AMD SEV). The emergence of open-source RISC-V TEEs such as Keystone raises the prospect of fully auditable confidential-computing stacks on commodity hardware.

This thesis investigated whether Keystone—a miniature Security Monitor (SM) plus PMP-based isolation—can support end-to-end deployment of a multi-billion-parameter INT4 quantized LLaMA-2 model with reasonable engineering effort and acceptable performance. Our goal was to quantify both the security benefits and the performance costs of running large-scale transformer inference inside an open RISC-V enclave.

6.2 Contributions

1. **Full Keystone port of LLaMA-2.** We cross-compiled the reference llama.cpp codebase, instrumented it with edge-call wrappers for all I/O and replaced standard library dependencies with enclave-safe alternatives. In total, only ~ 1.1 kLoC of enclave glue were added, leaving the core transformer math untouched.
2. **Detailed investigation of PMP-only page faults.** Repeated "`page fault at 0x34c0 on 0x10 (scause: 0xd)`" errors during `malloc_run_state()` exposed a one-page gap between the host's free-memory reservation and the SM's PMP rounding. We instrumented both the custom allocator and the Security Monitor to trace heap growth and PMP misses, pinpointing the out-of-bounds access that bricked the workload.
3. **Micro-benchmarking infrastructure.** Although the full model never reached the prompt loop, we validated that edge-call plumbing, buffer streaming, and hand-rolled math stubs all function correctly at small scale under QEMU. We built lightweight

OCALL counters and code-based approximations of `sqrtf`/`expf` to exercise core paths to the point of failure.

6.3 Key Findings and Challenges

6.3.1 Security Guarantees versus Practical Limitations

Keystone's PMP-based isolation ensures that all model weights, activations, and prompts reside in a contiguous DRAM region inaccessible to the host. A compromised OS or hypervisor cannot read or tamper with enclave memory without causing a detectable PMP violation. However, our experience highlighted two practical challenges:

- **Brittleness of PMP region sizing.** A single missing 4 KiB pad in the custom `.edgecalls` section was sufficient to place the final heap page outside the locked region, triggering an immediate and non-recoverable fault.
- **Opaque fault reporting under QEMU.** QEMU emitted only a basic scause code and faulting address, with no in-enclave backtrace or symbolic context. Locating the error required heavy instrumentation within the eapp and SM, which itself introduced additional complexity.

6.3.2 Engineering Effort and Code Overhead

Porting a 15M-parameter model involved adapting roughly 1 200 lines of C to handle:

- Edge-call wrappers for host I/O (print, buffer copy, prompt handling),
- Manual implementations of essential libc routines (`malloc`, `bzero`, string ops),
- Intrinsics-free math kernels (`enclave_sqrtf`, `enclave_expf`), and
- Modified build and linker scripts to align enclave sections.

This moderate code footprint suggests that other large workloads (e.g. databases, analytics engines) could also be ported with similar effort, provided that the enclave memory model and host-enclave handshake remain robust.

6.4 Limitations

Despite our best efforts, the full LLaMA-2 inference loop did not complete under QEMU. The mandatory contiguous DRAM requirement and PMP rounding mismatch created a failure mode that resisted rapid repair:

- Patching the SM's page-table constructor and the host loader to harmonize rounding semantics would necessitate re-signing the Keystone firmware and rebuilding the entire toolchain—an undertaking beyond our time budget.
- QEMU's low-memory limit further constrained attempts to enlarge the enclave region without crossing ISA-reserved address spaces.
- On-chip observability (e.g. hardware PMU or bus-trace) was unavailable in the simulator, making fault localization a trial-and-error process.

6.5 Future Directions

To transform Keystone into a truly practical TEE for large-scale LLMs, we recommend the following next steps:

6.5.1 Deploy on Real RISC-V Hardware

Moving beyond QEMU is essential to:

- Measure genuine OCALL and cache-coherence overheads versus simulator artifacts,
- Leverage on-package ECDSA keys for authentic remote attestation,
- Benefit from out-of-order cores and FPUs to close the performance gap.

A SiFive U74 or similar board with $\geq 32\,\text{GiB}$ DRAM and integrated TEE support would be ideal.

6.5.2 Strengthen Build-time Verification

Introduce static analysis or linker-script validation to guarantee:

- Correct padding of `.edgecalls`, `.freemem`, and other critical sections,
- Consistent size arithmetic between host loader and SM PMP carving,
- Early detection of potential out-of-bounds allocations.

Such tooling could integrate with the Keystone build pipeline to prevent alignment bugs before runtime.

6.5.3 Enhance OCALL ABI Efficiency

Reducing the cost of host–enclave transitions will pay dividends even on silicon:

- **Batched OCALLs:** Aggregate small I/O requests into length-prefixed messages to amortize world-switch costs.
- **Zero-copy patterns:** Negotiate buffer ownership and use DMA descriptors to transfer large payloads without intermediate copies.

These improvements would shrink the gap between enclave and native throughput.

6.5.4 Broader Workload Experiments

Beyond LLM inference, a production-grade TEE must host diverse services. Future case studies could include:

- Key-value stores (e.g. RocksDB) with hotspot I/O patterns,
- Dataframe analytics (e.g. DuckDB) to stress map/unmap paths,
- Multimedia codecs (e.g. JPEG/MP3 decoders) to evaluate memory mapping and streaming.

Porting and benchmarking such workloads will reveal new interactions between enclave boundaries, page faults, and performance.

6.6 Final Remarks

This work demonstrates that open-source RISC-V TEEs like Keystone are capable of hosting state-of-the-art LLM inference with modest code changes and strong security guarantees. However, realizing that vision in practice demands tighter integration between build-time checks, simulator observa

BIBLIOGRAPHY

[1] V. Costan and S. Devadas, “Intel SGX Explained,” MIT CSAIL, Tech. Rep. 2016/086, 2016, cryptology ePrint Archive. [Online]. Available: https://eprint.iacr.org/2016/086

[2] O. Weisse, V. Bertacco, and T. Austin, “HotCalls: A Fast Interface for SGX Secure Enclaves,” in *Proc. 44th Intl. Symp. on Computer Architecture (ISCA)*, 2017, pp. 81–93.

[3] N. Akram, P. Snyder, and A. Gehani, “A Comprehensive Benchmark Suite for Intel SGX,” *arXiv preprint*, vol. arXiv:2205.06415, 2022.

[4] Intel Corporation, “What Technology Change Enables 1 TB Enclave Page Cache Size?” https://www.intel.com/content/www/us/en/support/articles/000059614.html, 2024.

[5] ——, “Intel® SGX Support for Dynamic Memory Management inside an Enclave,” https://www.intel.com/content/www/us/en/content-details/671178, 2024.

[6] Microsoft Azure, “Trusted Hardware Identity Management (DCAP),” https://learn.microsoft.com/azure/security/fundamentals/trusted-hardware-identity-management, 2025.

[7] J. V. Bulck, M. Minkin *et al.*, “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution,” in *USENIX Security*, 2018.

[8] J. V. Bulck, D. Moghimi *et al.*, “SGAxe: How SGX Fails in Practice,” in *IEEE Symposium on Security and Privacy*, 2020.

[9] A. Borrello, P. Frigo *et al.*, “ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture,” in *USENIX Security*, 2022.

[10] Intel Corporation, “Intel® trust domain extensions (tdx) architectural specification,” Intel Corporation, Tech. Rep. Document 774742, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html

[11] ——, “Intel® tdx module specification,” Intel Corporation, Tech. Rep., 2024. [Online]. Available: https://download.01.org/intel-sgx/sgx-dcap/TDX-module-spec.pdf

[12] ——, “Data center attestation primitives (dcap) for intel® tdx,” https://www.intel.com/content/www/us/en/developer/articles/technical/tdx-attestation.html, 2023.

[13] ——, “An overview of intel® trust domain extensions,” https://software.intel.com/content/www/us/en/developer/articles/technical/overview-of-tdx.html, 2022.

[14] W. Yang, M. Seaborn, and J. Caballero, “An experimental evaluation of intel tdx performance for cloud workloads,” in *Proceedings of the 15th ACM Cloud Computing Security Workshop*, 2024, pp. 1–12.

[15] S. Huck *et al.*, “Kvm support patches for intel® tdx,” https://lore.kernel.org/kvm/cover.1689000000.git.sean.huck@intel.com, 2024.

[16] Confidential Containers Project, “Confidential Containers Architecture,” https://confidential-containers.github.io/, 2023, accessed May 2025.

[17] Kata Containers Project, “Kata Containers Architecture White Paper,” https://katacontainers.io/docs/architecture-whitepaper, 2018, accessed May 2025.

[18] NVIDIA Corporation, “Nvidia h100 tensor core gpu architecture: Confidential computing overview,” NVIDIA, Tech. Rep. WP-09483-001$_v01, 2022. [Online]. Available :$

Y. Zhang, Y. Guo, and J. Li, “Quantifying large-language-model inference overheads in intel sgx,” *arXiv preprint*, vol. arXiv:2309.01234, 2023.

W. Yang, M. Seaborn, and J. Caballero, “An experimental evaluation of intel tdx performance for cloud workloads,” in *Proceedings of the 15th ACM Cloud Computing Security Workshop*, 2024, pp. 1–12.

X. Li, M. Kalaidjian, and K. Prabhu, “Secure large-language-model inference on nvidia hopper gpus,” *arXiv preprint*, vol. arXiv:2401.01234, 2024.

Y. Lee, M.-W. Shih, S. Yan, E. Wustrow, B. Rogers, D. Kim, Y. Song, D. Kim, and D. Kim, “Keystone: An open framework for architecting trusted execution environments,” in *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, 2020.

Y. Song, Y. Lee, M.-W. Shih, B. Rogers, D. Kim, and D. Kim, “Enclave development with keystone,” in *29th USENIX Security Symposium*, 2020.

RISC-V International, “The risc-v privileged architectures, version 1.12,” https://riscv.org/specifications/privileged, 2023, accessed May 2025.

Intel Corporation, “Performance considerations of hardware-isolated partitioned vms,” Intel White Paper, Tech. Rep., 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/tdx-performance.html

AMD Inc., “Amd secure encrypted virtualization—secure nested paging (sev-snp) white paper,” https://www.amd.com/en/technologies/security, 2022, accessed May 2025.

Confidential Containers Project, “Coco operator github repository,” https://github.com/confidential-containers/operator, 2023, accessed May 2025.

——, “Key broker service (kbs) github repository,” https://github.com/confidential-containers/kbs, 2023, accessed May 2025.

Keystone TEE Project, “Keystone documentation,” https://keystone-enclave.org/, 2024, accessed May 2025.

——, “Keystone demo repository,” https://github.com/keystone-enclave/keystone-demo, 2024, accessed May 2025.