Private Distributed Inference on Consumer Hardware

Enabling Confidential Computation on Third-Party Apple Silicon via Software Access Path Elimination

Gajesh Naik

Eigen Labs

gajesh@eigenlabs.org

April 2026

Abstract

We present a system and method for private machine learning inference on a distributed network of consumer hardware. The core problem: a user wants to run an AI model on someone else's Mac, but the machine's owner—who has root access and physical custody—should not be able to see the user's prompts or the model's responses. The machines lack hardware Trusted Execution Environments (TEEs) for application-level memory isolation.

Our approach eliminates every software path through which inference data could be observed. The inference engine runs directly inside a single hardened Swift process—no subprocess, no local server, no inter-process communication—using `mlx-swift-lm` on the Apple Silicon GPU. The macOS kernel blocks all external access to this process's memory: debuggers are denied at the system call level (PT_DENY_ATTACH), memory-reading APIs are blocked by Hardened Runtime, and these protections cannot be disabled without rebooting, which terminates the process and erases its data. We prove that System Integrity Protection, once verified, is immutable for the process lifetime (Theorem 1).

A multi-layer attestation architecture verifies each machine in the network: (1) a Secure Enclave signature proves hardware-bound identity, (2) Apple's MDM framework independently confirms the machine's security configuration, (3) Apple's Managed Device Attestation (MDA) produces an Apple-signed certificate chain proving the device is genuine hardware, with an MDA nonce binding the provider's signing key to the attested device, (4) periodic challenge-response confirms security posture has not degraded since enrollment, and (5) APNs code-identity attestation proves the provider is running the genuine, team-signed Darkbloom binary by pushing an encrypted nonce that only a properly provisioned app can receive and requiring a Secure Enclave signature on the returned nonce.

After eliminating all software access paths, the only remaining attack requires physically probing the memory chips—which are soldered directly into Apple's System-on-Chip package. This is the same residual threat model accepted by Apple's Private Cloud Compute [1] for Siri and Apple Intelligence. We validate the system end-to-end on production hardware with live inference, demonstrating negligible security overhead (12 ms per request).

1 Introduction

When a user sends a prompt to a cloud AI service, that prompt travels to a server owned and operated by someone else. The user's privacy rests on a contractual promise—a privacy policy—not a technical guarantee. Hardware Trusted Execution Environments (TEEs) such as Intel TDX [6], AMD SEV-SNP [7], and NVIDIA Confidential Computing [8] can provide cryptographic memory isolation, but these technologies exist only on enterprise server hardware costing $14,000 or more.

Meanwhile, over 100 million Apple Silicon Macs have been sold since 2020. With 64–256 GB of unified memory and 273–819 GB/s memory bandwidth, they can run models with up to 235 billion parameters at interactive speeds. These machines sit idle most of the day. A distributed network that lets Mac owners contribute their idle compute for AI inference—while guaranteeing that they cannot see the prompts being processed—would make private inference available at a fraction of the cost of cloud TEEs.

The challenge is that Apple Silicon provides no hardware TEE accessible to third-party applications:

- The **Secure Enclave** provides hardware-bound P-256 key generation and ECDSA signing, but cannot encrypt main system memory or provide isolated execution for arbitrary code.
- **App Attest** (`DCAppAttestService`) returns `false` for `isSupported` on macOS—it is available only on iOS and iPadOS. We therefore introduce APNs code-identity attestation as a macOS-compatible replacement for binary-identity proof (Section 6.5).
- No public API exposes boot measurements or PCR-style remote attestation.

We present a system that makes this possible. Rather than encrypting memory (which Apple Silicon cannot do for third-party code), we *eliminate every software mechanism* through which the machine's owner could observe inference data—the same design philosophy Apple employs in Private Cloud Compute [1]. Apple's PCC removes all software interfaces (shell, debugger, persistent storage) that could access inference data. We apply this principle to a harder problem: in PCC, Apple owns the hardware and controls physical access; in our system, the hardware owner is the adversary.

Contributions. This paper makes the following contributions:

1. A **distributed private inference architecture** that enables confidential AI computation on a network of third-party Apple Silicon machines, where each machine's owner is assumed to be adversarial.

2. A **formal security model** for this setting, including a proof that System Integrity Protection provides runtime immutability under standard assumptions (Theorem 1), and a complete enumeration of the software attack surface.

3. An **in-process inference design** that embeds the inference engine directly in a hardened Swift process using `mlx-swift-lm`, eliminating all inter-process communication channels that could be observed, with optional **hypervisor memory isolation** reported as telemetry for future RDMA hardening.

4. A **multi-layer attestation architecture** combining Secure Enclave signatures, MDM-based independent verification, Apple MDA hardware attestation with Apple-signed certificate chains, MDA nonce-based SE key binding, periodic challenge-response with fresh security state, and APNs code-identity attestation proving the running binary is genuine.

5. A **combined enrollment protocol** that unifies MDM enrollment and an ACME `device-attest-01` payload into a single configuration profile; the active hardware-provenance path today is MDA-over-MDM rather than ACME, with ACME retained for future transport-layer use.

6. **End-to-end validation** on production hardware (Apple M2 and M4 Max) with live inference, demonstrating that the security mechanisms introduce negligible overhead.

7. A **cost analysis** showing that Mixture-of-Experts models create 6–32× cost advantages on unified-memory hardware versus cloud GPUs, establishing economic feasibility.

2 Related Work

Hardware TEE-based inference. Intel TDX [6] and AMD SEV-SNP [7] encrypt VM memory and provide remote attestation. NVIDIA Confidential Computing [8] extends this to GPU memory. These approaches provide strong isolation but require enterprise server hardware. Our work achieves comparable privacy guarantees on consumer hardware without memory encryption.

Distributed compute networks. Akash [2] provides general compute with TEE support in development. io.net [3] aggregates GPU resources without hardware security guarantees. Ritual [4] uses zero-knowledge proofs and Intel SGX. Bittensor [5] employs stake-weighted consensus for result verification. None address inference privacy on consumer hardware lacking TEEs.

Apple Private Cloud Compute. PCC [1] achieves inference privacy on Apple Silicon servers (M2 Ultra) without memory encryption through five requirements: stateless computation, enforceable guarantees, no privileged runtime access, non-targetability, and verifiable transparency. Implementation includes: no persistent storage, no SSH or shell access, immutable OS via Signed System Volume, cryptographic erasure on reboot, and OHTTP anonymization. Our work differs in that PCC operates in *Apple-controlled data centers* with physical security (access controls, tamper detection), while our system must defend against the hardware owner as adversary.

Cryptographic approaches. Fully Homomorphic Encryption [9] introduces 10^4–$10^6\times$ computational overhead. Secure Multi-Party Computation [10] requires multiple non-colluding servers. zkLLM [11] and DeepProve-1 [12] verify inference integrity via zero-knowledge proofs but do not protect input privacy during computation. These remain impractical for interactive LLM inference at current performance levels.

2.1 Comparison with Apple Private Cloud Compute

Table 1 compares the system's security properties with Apple PCC.

The key distinction is who owns the hardware. In PCC, Apple owns the servers and controls physical access. In our system, the hardware belongs to an independent provider who may be adversarial. The multi-layer attestation architecture (Section 6) compensates for the absence of physical security by providing multiple independent verification paths, culminating in Apple-signed hardware provenance certificates. The result is that consumers get the same residual threat model—physical memory probing only—whether they use Apple's private cloud or our distributed network.

Table 1: Comparison with Apple Private Cloud Compute [1].

Property	Apple PCC	This Work
Hardware owner	Trusted (Apple)	Adversarial
Physical security	Facility controls	Soldered LPDDR5x
Coordinator TEE	N/A (Apple infra)	AMD SEV-SNP
OS immutability	SSV	SIP + ARV + SSV hash
Shell/debug access	Removed entirely	Blocked at kernel level
Memory encryption	None	None
DMA/RDMA isolation	Facility controls	IOMMU default-deny; hypervisor Stage 2 planned
Provider verification	Apple-signed	SE + MDM + MDA + APNs code identity
Hardware provenance	Apple supply chain	Apple MDA certificates
Residual attack	Physical probing	Physical probing

3 Threat Model and Assumptions

The fundamental challenge in distributed inference is that the person running the computation is not the person who owns the data. A provider contributes their Mac's compute, but should learn nothing about the prompts processed on it. This section formalizes what we assume about each party and what a malicious provider can attempt.

3.1 Actors

Definition 1 (System Actors)**.** *We define four classes of actor:*

- $\mathcal{C}$ *(Consumer): The party submitting inference requests. Trusted—they have no incentive to attack their own data.*
- $\mathcal{K}$ *(Coordinator): The routing and verification service. Trusted via hardware: runs in a Confidential VM (AMD SEV-SNP [7]) with cryptographic attestation of the deployed container image. The coordinator's memory is encrypted by the TEE hardware; neither the cloud operator nor the system operator can access it.*
- $\mathcal{P}$ *(Provider): The Mac owner contributing compute.* ***Assumed adversarial****—they own the hardware, have root access, and may actively attempt to read inference data. This is the strongest practical assumption: we guarantee privacy even if the provider is trying to break it.*
- $\mathcal{A}$ *(Apple): Trusted for the correctness of Secure Enclave hardware, SIP implementation, Kernel Integrity Protection, Hardened Runtime enforcement, IOMMU policy, code signature verification, and the Managed Device Attestation certificate authority.*

3.2 Adversary Capabilities

Definition 2 (Threat Model)**.** *The adversarial provider $\mathcal{P}$ can:*

- *Execute arbitrary code as root on the Mac*
- *Install any user-level software (subject to SIP restrictions)*
- *Reboot the machine at any time*
- *Physically access the machine exterior and ports*
- *Enable RDMA over Thunderbolt 5 (requires Recovery OS boot)*
- *Attempt to intercept or modify network traffic*

The adversary $\mathcal{P}$ cannot*:*

- *Exploit unpatched kernel vulnerabilities (Assumption 3)*
- *Modify Secure Enclave firmware or extract its keys*
- *Desolder LPDDR5x memory from the SoC package without destroying it*
- *Compromise Apple's Managed Device Attestation signing infrastructure*

Definition 3 (Assumptions)**.** ***Assumption 1*** *(Kernel Integrity): The macOS kernel has no unpatched vulnerabilities that allow SIP bypass, Hardened Runtime bypass, or KIP circumvention. Apple has patched all known SIP bypass CVEs (e.g., CVE-2022-22583, CVE-2022-42821, CVE-2023-32369) within weeks of disclosure.*

Assumption 2 *(Secure Enclave Integrity): The Secure Enclave hardware and firmware correctly implement key generation, signing, and key isolation. This is analogous to trusting Intel SGX or ARM TrustZone hardware.*

Assumption 3 *(Apple Attestation CA Integrity): Apple's Enterprise Attestation Root CA private key is not compromised, and Apple's attestation servers correctly verify device properties before issuing certificates. This is analogous to trusting any certificate authority in the Web PKI.*

3.3 Security Goal

In plain terms: a provider processes a prompt on their Mac, returns the result, and learns nothing about what the prompt said or what the model replied. More precisely:

Definition 4 (Privacy Property)**.** *Let x be an inference input (prompt) and $y = f(x)$ the model output. The system satisfies the* privacy property *if, for all polynomial-time adversaries $\mathcal{P}$:*

$$\Pr[\mathcal{P} \text{ learns } x \text{ or } y] \leq \mathit{negl}(\lambda) \tag{1}$$

where λ is the security parameter, under Assumptions 1–3, excluding physical memory probing attacks requiring destructive hardware modification.

4 System Architecture

The distributed network comprises three principal components: a *coordinator* that routes consumer requests to providers and verifies their attestations, *provider agents* running on Apple Silicon Macs that execute inference, and a *Secure Enclave module* that gives each provider a hardware-bound cryptographic identity. The provider is a Swift CLI (`darkbloom`) that runs `mlx-swift-lm` in-process on the Apple Silicon GPU. Figure 1 depicts the complete architecture.

4.1 In-Process Inference

Traditional inference serving architectures (vLLM, Ollama, llama.cpp) run the engine as a separate HTTP server process. This creates two attack surfaces: (a) localhost TCP traffic can be captured via `tcpdump`, which functions even with SIP enabled; and (b) the subprocess binary can be replaced with a version that logs requests.

Our architecture eliminates both attack surfaces by embedding the inference engine directly in a single hardened Swift process. The provider uses `mlx-swift-lm` (a fork maintained under `libs/mlx-swift-lm`) to load model weights and run generation inside the same address space that holds the WebSocket client, attestation signer, and encryption keys:

```
import MLXLMCommon

// Model loaded in-process
let container = try await LLMModelFactory.
    shared.loadContainer(
    configuration: config)

// Inference in protected memory
let result = try await container.perform { _,
    tokenizer in
    try generate(promptTokens: tokens,
                 parameters: params,
                 model: $0)
}
```

Listing 1: Inference executes inside the hardened Swift process address space. No subprocess, HTTP server, socket, or pipe is created.

Model weights (up to 76 GB for a 122B-parameter model at 4-bit quantization), tokenizer, and all intermediate activations reside in a single process address space. This space is protected by three kernel-enforced mechanisms described in Section 4.2. Because the Swift binary is code-signed and SIP prevents execution of modified signed binaries, the only binaries that can act as decryption endpoints are the genuine team-signed build (further proved by APNs code-identity attestation, Section 6.5) and any blessed variants tracked in a transparency log.

4.1.1 Memory Sanitization

After each inference request completes, buffers containing prompts and model outputs are zeroed using `memset_s`, which is guaranteed not to be optimized away by the compiler, ensuring sensitive data is overwritten before the buffers are released.

4.2 Access Path Elimination

Three kernel-enforced mechanisms jointly eliminate all software paths to the inference process's memory:

1. **PT_DENY_ATTACH** (ptrace constant 31): Invoked at process startup before any sensitive data is loaded. Instructs the macOS kernel to permanently deny all `ptrace` requests against this process, including from root. This blocks `lldb`, `dtrace`, and Instruments.

2. **Hardened Runtime**: The binary is code-signed with hardened runtime options and explicitly *without* the `com.apple.security.get-task-allow` entitlement. The kernel denies `task_for_pid()` and `mach_vm_read()` from any external process.

3. **System Integrity Protection (SIP)**: Enforces both of the above at the kernel level. With SIP enabled, root cannot circumvent Hardened Runtime protections, load unsigned kernel extensions, or modify protected system binaries. Section 9.1 proves that SIP, once verified, is immutable for the process lifetime.

4.3 Coordinator Trust Boundary

The coordinator runs in a Confidential VM (AMD SEV-SNP) on cloud infrastructure. The container image is cryptographically attested—consumers can verify that the coordinator executes unmodified code. The coordinator's memory is encrypted by the TEE hardware; neither the cloud infrastructure operator nor the system operator can access it.

The coordinator performs request routing, attestation verification, billing, and key management. Consumers may optionally seal requests to the coordinator's long-lived X25519 key; in all cases the coordinator decrypts the request body inside its Confidential-VM memory for routing and billing. Prompt content is never logged or retained, and is immediately re-encrypted with a fresh per-request NaCl Box to the selected provider's attested X25519 key (Section 5). The coordinator is therefore a trusted plaintext intermediary for routing, not a blind relay.

5 End-to-End Encryption

Encryption is hop-by-hop rather than strictly end-to-end. The system uses NaCl Box (X25519 key agreement + XSalsa20-Poly1305 authenticated encryption) [16] for both legs.

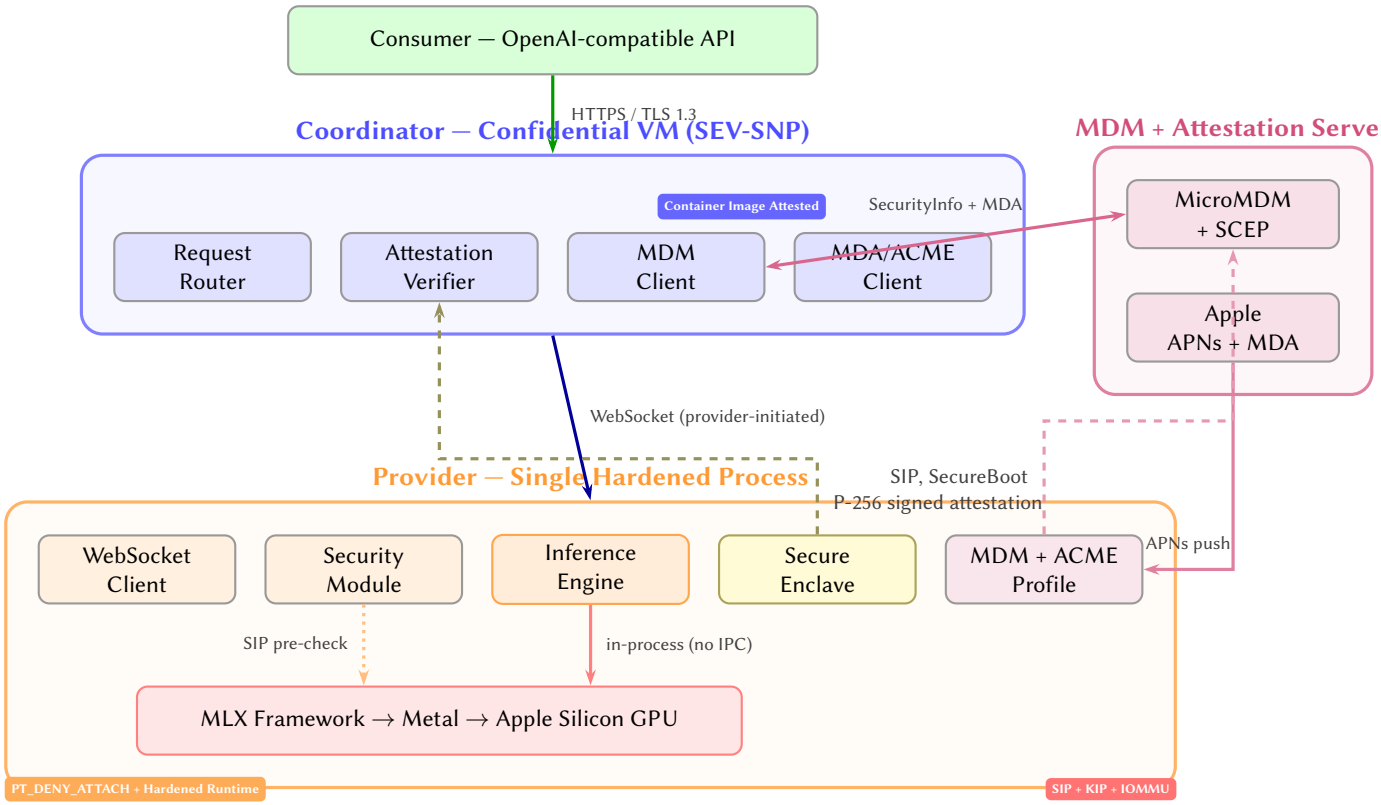

Figure 1: System architecture. The consumer connects via HTTPS (optionally NaCl Box sealed) to a coordinator running in a Confidential VM. The coordinator routes requests to provider Macs over provider-initiated WebSocket connections, re-encrypting each request with a per-request NaCl Box to the provider's attested X25519 key. Each provider runs inference in a single hardened Swift process protected by PT_DENY_ATTACH, Hardened Runtime, SIP, KIP, and IOMMU. The Secure Enclave provides hardware-bound P-256 identity; APNs code-identity attestation proves the binary is genuine. MDA-over-MDM (not ACME) is the operative hardware-provenance path today.

5.1 Two-Leg Model

Consumer → coordinator. By default the consumer sends plaintext over TLS. Optionally, the consumer may seal the request body to the coordinator's long-lived X25519 public key (advertised at GET `/v1/encryption-key`). The coordinator decrypts the body inside its Confidential-VM memory for routing and billing.

Coordinator → provider. For every inference request the coordinator:

1. Generates an ephemeral X25519 key pair (sk_e, pk_e)
2. Generates a random 24-byte nonce n
3. Computes the shared secret and encrypts the request body: $c = \text{Box}(m, n, pk_{\text{provider}}, sk_e)$
4. Sends $(pk_e, n \| c)$ to the provider over the outbound WebSocket

The provider decrypts using its in-memory X25519 key K (bound to its Secure Enclave identity via the attestation blob, Section 6.1). Response SSE chunks are encrypted back to the coordinator's ephemeral public key and decrypted by the coordinator before being relayed to the consumer (re-sealed if consumer sealing is enabled).

5.2 Forward Secrecy

Each request uses a fresh coordinator ephemeral key pair. Compromise of any single ephemeral key does not enable decryption of past or future requests. The provider's X25519 key K lives only in the hardened process's protected memory; its binding to the Secure Enclave identity is verified by the coordinator.

6 Five-Layer Attestation Architecture

The attestation architecture provides five independent layers of verification, each addressing distinct threat vectors. Table 2 summarizes the layers and their properties.

6.1 Layer 1: Secure Enclave Attestation

On first execution, the provider generates a P-256 ECDSA signing key inside Apple's Secure Enclave via CryptoKit. The private key never leaves the Secure Enclave hardware—the `dataRepresentation` stored on disk is an opaque handle that functions only on the device that created it. The provider constructs a signed attestation

Table 2: Five-layer attestation architecture. Each layer addresses threats not covered by the others.

Layer	Mechanism	Threat Addressed
1	SE P-256 ECDSA	Hardware-bound identity; replay prevention
2	MDM SecurityInfo	Independent SIP/SecureBoot verification
3	MDA-over-MDM	Apple-signed hardware provenance
4	Challenge-response	Liveness; continuous security state
5	APNs code identity	Genuine binary proof (v0.6.0)

blob:

Table 3: Attestation blob fields. JSON-encoded with deterministic key ordering (alphabetical) and signed by the Secure Enclave P-256 key.

Field	Type	Purpose
`authenticatedRootEnabled`	bool	Sealed system volume (ARV)
`binaryHash`	hex string	SHA-256 of running binary
`chipName`	string	Hardware identity
`encryptionPublicKey`	base64	X25519 key bound to this identity
`hardwareModel`	string	Machine model identifier
`hypervisorActive`	bool	Hypervisor VM active (Section 10)
`osVersion`	string	macOS version
`publicKey`	base64	P-256 public key (raw X‖Y)
`rdmaDisabled`	bool	RDMA status via `rdma_ctl`
`secureBootEnabled`	bool	Secure Boot status
`secureEnclaveAvailable`	bool	SE hardware present
`serialNumber`	string	Device serial for MDM cross-ref
`sipEnabled`	bool	SIP status
`systemVolumeHash`	hex string	APFS snapshot hash
`timestamp`	ISO 8601	Freshness

Three fields merit particular attention:

serialNumber. The hardware serial number enables the coordinator to correlate the device with its MDM enrollment record and request independent security verification. This field bridges Layer 1 (self-reported) to Layers 2 and 3 (independently verified).

binaryHash. The SHA-256 hash of the running provider binary. Because this value is self-reported by the measurer—the potentially adversarial provider—it cannot serve as a standalone trust signal: a modified binary can simply report the hash of a blessed version. Starting in v0.6.0, `binaryHash` is used only for drift telemetry and transparency-log matching; the active code-identity proof is APNs code-identity attestation (Section 6.5). The hash is retained in the SE-signed status canonical so that a blessed-build policy can be re-enabled as a rollback or emergency gate.

encryptionPublicKey. Binds the provider's X25519 encryption key (used for hop-by-hop encrypted inference, Section 5) to the same Secure Enclave identity. The coordinator verifies this matches the key in the registration message, proving the same physical device controls both the signing and decryption keys.

6.1.1 Cross-Language Signature Verification

The attestation protocol spans two programming languages: Swift (Secure Enclave signing and provider agent) and Go (coordinator verification). Both must produce byte-identical JSON representations for signature verification to succeed. We enforce this through:

- Struct fields declared in alphabetical order (by JSON key name) in both languages
- Swift's `JSONEncoder` configured with `.sortedKeys` output formatting
- Go coordinator preserves raw attestation bytes (`json.RawMessage`) for verification, with a `marshalSortedJSON` fallback using `map[string]interface{}` (Go 1.12+ marshals map keys in sorted order)
- `omitempty` handling: nil fields omitted (not set to null)
- ISO 8601 timestamp format with consistent precision
- Base64 standard alphabet (no URL-safe variant); raw P-256 key format (64 bytes: X‖Y, no 0x04 prefix)

6.1.2 Coordinator Verification Procedure

The coordinator verifies each attestation through the following steps:

1. Decode the P-256 public key from the attestation blob (64-byte raw X‖Y format)
2. Hash the original JSON blob bytes with SHA-256
3. Verify the DER-encoded ECDSA signature against the hash
4. Enforce: `secureEnclaveAvailable` = true, `sipEnabled` = true, `secureBootEnabled` = true
5. Verify: `encryptionPublicKey` matches the registration message
6. Verify: `timestamp` is within acceptable freshness window
7. Record `binaryHash` for drift telemetry and transparency-log matching
8. If all pass, proceed to Layer 2 verification

6.2 Layer 2: MDM-Based Independent Verification

Secure Enclave attestation (Layer 1) has two inherent limitations: (a) the coordinator cannot prove the P-256 signing key resides in the Secure Enclave rather than in software, since both produce identical ECDSA signatures; and

(b) the SIP status in the attestation blob is obtained via `csrutil status`, a software check that could theoretically be spoofed on a compromised system.

Layer 2 addresses both limitations through *independent verification*: the coordinator queries the device's security posture through Apple's MDM framework, which returns data from the OS MDM subsystem—not from the provider software.

6.2.1 SecurityInfo Command

Apple's MDM `SecurityInfo` command [13] returns the following fields, as reported by the operating system's MDM client:

Table 4: Fields returned by the MDM `SecurityInfo` command. The coordinator requires SIP enabled, Secure Boot "full," and Authenticated Root Volume enabled.

Field	Verification Purpose
`SIPEnabled`	SIP active—root cannot modify `/System` or bypass Hardened Runtime
`SecureBootLevel`	"full": only Apple-signed code at boot
`AuthRootVolume`	Signed System Volume Merkle tree intact
`FDE_Enabled`	FileVault disk encryption active
`RecoveryLock`	Recovery Mode requires password
`FirewallEnabled`	Application firewall active

The coordinator cross-checks these values against the self-reported attestation blob. Any discrepancy results in immediate rejection.

6.2.2 Authenticated Root Volume

The Authenticated Root Volume (ARV) seals the macOS system volume with a Merkle tree of SHA-256 hashes. Any modification to any system file—including `mdmclient`, `csrutil`, and kernel caches—breaks the seal and prevents the volume from mounting. ARV provides a stronger guarantee than SIP alone: SIP prevents *runtime* modifications, while ARV proves the *entire system volume* is Apple's unmodified build.

The `systemVolumeHash` field in the attestation blob contains the cryptographic hash of the APFS system volume snapshot, uniquely identifying the exact macOS build. The coordinator can cross-reference this against known-good builds published by Apple.

6.3 Layer 3: MDA-over-MDM Hardware Attestation

Layers 1 and 2 verify security *posture* (is SIP enabled?) but do not provide cryptographic proof of hardware *identity* (is this a genuine Apple device?). Layer 3 addresses this through Apple's Managed Device Attestation (MDA) protocol, requested over MDM via the `DeviceInformation` command with `DevicePropertiesAttestation` [15].

6.3.1 Protocol Overview

After the device enrolls in MDM, the coordinator sends a `DeviceInformation` command requesting `DevicePropertiesAttestation`. The device contacts Apple's attestation servers and returns a DER-encoded X.509 certificate chain:

- **Leaf**: Device certificate with Apple-assigned OIDs encoding hardware properties
- **Intermediate**: Apple Enterprise Attestation Sub CA 1 (P-384)
- **Root**: Apple Enterprise Attestation Root CA (P-384, valid until 2047)

The coordinator verifies the chain against the embedded Apple Enterprise Attestation Root CA, extracts the OID-encoded properties, and cross-checks the serial number against the provider's self-reported attestation. The complete certificate chain is stored and exposed via a public API endpoint, enabling consumers to independently verify each provider against Apple's publicly available root CA certificate.

6.3.2 Certificate OID Fields

The leaf certificate in Apple's attestation chain encodes device properties as X.509 extensions with Apple-assigned OIDs:

Table 5: Apple-assigned OIDs in MDA leaf certificates. These values are signed by Apple's Enterprise Attestation Root CA and cannot be forged.

OID Suffix	Property	Verification Use
100.8.9.1	Serial Number	Device identity
100.8.9.2	UDID	Device identity
100.8.10.1	OS Version	OS currency
100.8.10.2	SepOS Version	SE firmware version
100.8.10.3	LLB Version	Boot chain integrity
100.8.11.1	Freshness Code	Replay prevention / SE key binding
100.8.13.1	SIP Status	SIP independently verified
100.8.13.2	Secure Boot	Boot security level
100.8.13.3	Kext Status	Kernel extension policy

All OIDs prefixed with 1.2.840.113635.

6.3.3 SE Key Binding via Freshness Nonce

When requesting MDA, the coordinator supplies a `DeviceAttestationNonce` equal to SHA-256 of the provider's SE public key. Apple embeds that hash in the `FreshnessCode` OID of the signed leaf certificate. The coordinator verifies that the returned freshness code matches the hash of the SE public key it already accepted, cryptographically binding the SE signing key to Apple-attested genuine hardware.

6.3.4 ACME `device-attest-01` Limitations

The enrollment profile still contains an ACME `device-attest-01` payload, but it is *not* the operative production trust anchor today. Hardware-bound ACME identities on macOS land in the data-protection keychain inside a restricted system access group that is inaccessible to third-party applications, including the Darkbloom provider; only Apple system services such as Wi-Fi EAP-TLS and Safari mTLS can use them. In addition, the ACME leaf issued by our setup does not currently carry the SIP/SecureBoot OIDs, and the mTLS ingress path that would populate the verifying headers is not functional in production. Consequently, the active hardware-provenance path is MDA-over-MDM, with ACME retained for future transport-layer use.

6.3.5 Comparison with SecurityInfo

Table 6 compares the two independent verification mechanisms.

Table 6: MDM SecurityInfo (Layer 2) vs. MDA-over-MDM (Layer 3).

Property	SecurityInfo	MDA
Trust anchor	OS MDM client	Apple Root CA
Forgery requires	SIP disabled	Apple CA compromise
Proves identity	No	Yes (serial, UDID)
Proves security	Yes (SIP, boot)	Yes (SIP, boot, SepOS)
Latency	Seconds	Minutes (Apple servers)
Availability	Always	Requires MDM enrollment

6.4 Layer 4: Continuous Challenge-Response

Layers 1–3 establish trust at enrollment time. Layer 4 provides continuous assurance that the provider's security posture has not changed since enrollment.

Every 5 minutes, the coordinator generates a random 32-byte nonce n and timestamp t, and sends an `attestation_challenge` message to the provider. The provider must:

1. Check current SIP status via `csrutil status`
2. Check current Secure Boot status
3. Check current RDMA status via `rdma_ctl status`
4. Check current hypervisor status
5. Compute signature $\sigma = \text{Sign}_{\text{SE}}(n \| t \| \text{pk})$ where pk is the registered public key
6. Return $(\sigma, n, \text{pk}, \text{sip\_enabled}, \text{secure\_boot\_enabled}, \text{rdma\_disabled}, \text{hypervisor\_active})$ within 30 seconds

The coordinator verifies:

- Nonce n matches the sent value
- Public key pk matches the registered key
- ECDSA signature σ is valid
- `sip_enabled` = true (immediate untrust if false)
- `secure_boot_enabled` = true (immediate untrust if false)
- Records `rdma_disabled` and `hypervisor_active` for telemetry and future policy enforcement

Immediate untrust policy: If SIP or Secure Boot is reported as disabled, the provider is immediately marked untrusted with no grace period. Disabled SIP or Secure Boot indicates a deliberate reboot with weakened security (by Theorem 1, SIP cannot be disabled without rebooting). RDMA status and hypervisor activity are currently reported as telemetry; strict RDMA-without-hypervisor routing enforcement is reserved for future deployment.

Failure escalation: Three consecutive signature verification failures result in untrust marking. This catches providers experiencing connectivity issues versus providers that have been replaced or compromised.

Reconnection scenario: If a provider reboots (e.g., to disable SIP), disconnects, and reconnects, the reconnection triggers a fresh registration with a new attestation blob. If the new attestation reports SIP as disabled, the provider is rejected. If the provider lies about SIP status, the next MDM SecurityInfo query independently reveals the true state.

6.5 Layer 5: APNs Code-Identity Attestation

Layers 1–4 establish hardware identity and continuous security posture, but they do not prove *which binary* is running on the device. A malicious provider could run a fork that logs prompts while presenting a valid SE key and MDA certificate. Layer 5 closes this gap through APNs code-identity attestation (v0.6.0).

This is a practical breakthrough because macOS does not expose `DCAppAttestService` to third-party apps—`isSupported` returns `false` on macOS—so App Attest cannot be used to prove binary identity. A self-reported `binaryHash` is equally insufficient, because the measurer is the potentially adversarial provider. APNs code-identity attestation solves the problem without a new Apple API: it repurposes Apple's existing push-delivery infrastructure as a non-self-reportable code-identity oracle.

Why APNs proves code identity. Only a process that satisfies three conditions can receive a push for our topic `io.darkbloom.provider`:

1. It is signed with our Developer ID team certificate.
2. It carries our globally unique App ID `io.darkbloom.provider`.

3. It is authorized by an Apple-signed provisioning profile containing the `aps-environment` entitlement.

These conditions are enforced at launch by the `AppleMobileFileIntegrity.kext` kernel extension (AMFI) working with the user-space `amfid` daemon. AMFI validates the code signature, entitlements, and provisioning profile before the process is allowed to register for remote notifications with Apple's push service. A modified binary, a re-signed binary under a different Team ID, or an app lacking the entitlement cannot obtain a valid APNs device token for our topic. This makes APNs delivery a strong, Apple-mediated statement that the receiving process is the genuine Darkbloom build.

Protocol. At registration the provider sends its APNs device token T alongside its X25519 public key K and SE public key. The coordinator:

1. Binds $T \leftrightarrow K$ to the WebSocket connection.
2. Generates a fresh 32-byte nonce n.
3. Encrypts the nonce to K using NaCl Box: $c = \text{Box}(n, K)$.
4. Sends an APNs background push to T carrying c.

Only the genuine Darkbloom binary receives the push. It decrypts n with K, signs the nonce with its SE P-256 key, and returns (n, σ) over the same WebSocket. The coordinator verifies:

- The returned nonce matches the one it pushed.
- The SE signature σ is valid against the public key bound at registration.

On success the connection is marked `CodeAttested`. Private text routing requires this flag once the code-attestation enforcement deadline has passed. The flag is per-connection and in-memory; reconnecting forces re-attestation, because a SIP downgrade requires a reboot which drops the WebSocket.

MDM remains required. APNs code identity proves *which binary* is running, but it does not prove the device's security posture (SIP, Secure Boot, hardware genuineness). Those properties still require Layer 2 (MDM `SecurityInfo`) and Layer 3 (MDA-over-MDM). The provider must therefore still install the combined MDM enrollment profile; APNs is not a replacement for MDM, but rather the missing code-identity piece that MDM and MDA do not provide.

Limits. Background push delivery is best-effort and device-budget-throttled; the coordinator can switch to alert mode if reliability becomes an issue. APNs requires a logged-in macOS GUI Aqua session, so headless or login-screen Macs cannot become code-attested and fail closed for private traffic. APNs binds App ID/Team ID, not exact binary version; version/downgrade control is intended to come from reproducible builds plus a public transparency log of blessed `cdhash` values.

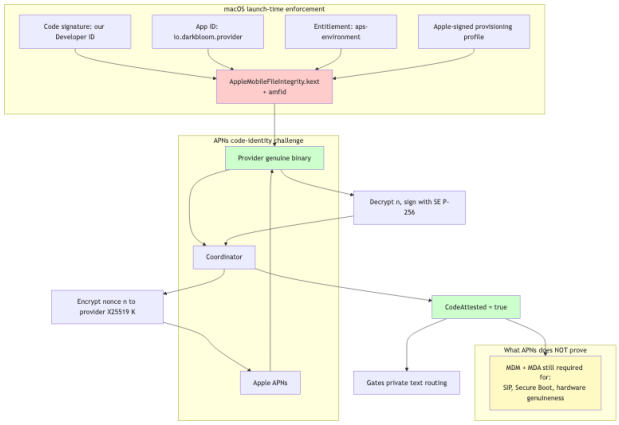

Figure 2: APNs code-identity attestation. AMFI (`AppleMobileFileIntegrity.kext`) enforces code signature, App ID, and `aps-environment` entitlement at launch, so only the genuine Darkbloom binary can register for our push topic. The coordinator pushes an encrypted nonce to that topic; the provider decrypts it with its bound X25519 key and returns a Secure Enclave signature, proving the WebSocket session is backed by genuine code holding the decryption key. MDM and MDA remain required for SIP, Secure Boot, and hardware provenance.

7 Combined Enrollment Protocol

A practical concern in deploying multi-layer attestation is user friction: requiring providers to install multiple configuration profiles creates opportunities for partial enrollment and increases abandonment. We address this with a *combined enrollment protocol* that unifies MDM enrollment, SCEP device identity, and an ACME payload into a single Apple configuration profile. The active hardware-provenance path today is MDA-over-MDM (Layer 3) rather than ACME, with ACME retained for future transport-layer use.

7.1 Profile Structure

The coordinator generates a per-device `.mobileconfig` containing three payloads:

1. **SCEP payload** (`com.apple.security.scep`): Generates an RSA-2048 identity certificate via Simple Certificate Enrollment Protocol. This certificate authenticates the device to the MDM server.
2. **MDM payload** (`com.apple.mdm`): Enrolls the device with the MDM server using the SCEP identity certificate. The `AccessRights` bitmask is set to 1041 ($2^0 + 2^4 + 2^{10}$), granting only:
   - Bit 0: Inspect device (basic query)
   - Bit 4: Query device information (model, OS, serial)

- Bit 10: Query security information (SecurityInfo command)

The remaining 10 capability bits are explicitly unset. The MDM server **cannot** erase the device, lock it, change the passcode, install or remove profiles or applications, modify settings, or query network information. The only commands the coordinator sends are `SecurityInfo` and `DeviceInformation` (for MDA).

3. **ACME payload** (`com.apple.security.acme`): Initiates the `device-attest-01` challenge. The payload specifies a P-384 hardware-bound key with Apple attestation. This payload is retained for future use but is not the operative production trust anchor today because the generated key lands in a platform-restricted keychain inaccessible to third-party apps and the resulting certificate does not carry the SIP/SecureBoot OIDs (Section 6.3).

7.2 Enrollment Flow

1. Provider runs the installation script, which detects the device serial number via `system_profiler`
2. Script requests a per-device profile from the coordinator: `POST /v1/enroll` with `{serial_number}`
3. Coordinator generates the combined profile with fresh UUIDs
4. macOS displays the profile contents and requested permissions to the user for review and consent
5. On installation, macOS processes all three payloads atomically:
   - SCEP certificate issued and stored in System Keychain
   - MDM enrollment completes; device checks in with MDM server
   - ACME challenge initiated (future use; not the active trust anchor)
   - MDA certificate chain obtained via MDM `DeviceInformation` when the coordinator requests `DevicePropertiesAttestation`
6. Provider is now eligible for all five layers of verification; Layer 5 (APNs code identity) completes automatically when the provider connects over its outbound WebSocket

7.3 Minimal Permission Principle

The MDM enrollment requests the minimum permissions necessary for security verification. Table 7 enumerates the complete MDM `AccessRights` bitmask.

Providers can unenroll at any time via System Settings, which removes the profile and all associated certificates immediately and completely.

Table 7: Apple MDM AccessRights bitmask. The system requests only bits 0, 4, and 10 (bold).

Bit	Permission	Ours	Typical
0	**Inspect device**	✓	✓
1	Install/remove configuration profiles	–	✓
2	Device lock and passcode removal	–	✓
3	Erase all data on device	–	✓
4	**Query device information**	✓	✓
5	Query network information	–	✓
6	Inspect provisioning profiles	–	✓
7	Install/remove provisioning profiles	–	✓
8	Inspect installed applications	–	✓
9	Restriction-related queries	–	✓
10	**Query security information**	✓	✓
11	Change device settings	–	✓
12	App management	–	✓

8 Trust Tiers and Routing Policy

The multi-layer attestation architecture (Section 6) and the combined enrollment protocol (Section 7) establish a spectrum of verifiable trust guarantees. We formalize this spectrum into *trust tiers* that consumers can select at inference time, allowing each request to specify the minimum acceptable level of hardware assurance. The coordinator's routing algorithm filters the candidate provider set by trust tier before applying the cost-minimization scheduler described below.

8.1 Tier Definitions

Table 8: Consumer-selectable trust tiers. Consumers specify a minimum tier; the coordinator routes only to providers meeting that threshold. Private text traffic additionally requires APNs code-identity attestation once enforcement is enabled.

Tier	Name	Verification Guarantee
0	`none`	No attestation
1	`self_signed`	SE-signed attestation blob
2	`hardware`	MDM SecurityInfo + Apple MDA + SE key binding via MDA nonce
–	`code_attested`	Tier 2 + APNs code-identity proof (private text gate)

8.2 Consumer-Side Selection

Consumers specify their minimum trust tier as a query parameter (`trust_level`) in the inference request. The coordinator's routing algorithm filters the candidate provider set by trust tier before applying the cost-minimization scheduler below. In production deployments, the coordinator enforces `hardware` as the minimum trust tier, and private text traffic is additionally gated on APNs code-identity attestation once the enforcement deadline passes.

Table 9: Trust properties by tier. "Crypto" = Apple CA-verified; "OS" = OS verified via MDM; "Self" = self-reported; "Apple" = proved via Apple's push infrastructure.

Property	none	self_signed	hardware	code_attested
Device is real Apple hardware	—	—	Crypto	Crypto
SIP / Secure Boot enabled	—	Self	OS + MDM	OS + MDM
SE key identity	—	Self	Self + MDA nonce	Self + MDA nonce
Genuine binary proof	—	—	—	Apple
Response integrity (signed)	—	—	SE signature	SE signature
Device serial verified by Apple	—	—	Crypto	Crypto
Periodic security re-check	—	—	Every 5 min	Every 5 min

8.3 Trust Property Summary

8.4 Verified Provider Requirements

Definition 5 (Verified Provider)**.** *A provider achieves* verified *status if and only if all of the following conditions hold:*

1. *Secure Enclave P-256 attestation signature is valid (Section 6.1)*
2. *Device is enrolled via the combined enrollment profile (Section 7)*
3. *MDM SecurityInfo confirms: SIP enabled, Secure Boot at "full" level, Authenticated Root Volume enabled*
4. *Apple MDA certificate chain is valid and serial number matches self-reported attestation (Section 6.3)*
5. *MDM-reported security posture matches self-reported attestation with no discrepancies*
6. *Periodic challenge-response succeeds with fresh SIP and Secure Boot status (Section 6.4)*
7. *APNs code-identity attestation succeeds, proving the running binary is the genuine team-signed build (Section 6.5)*

Providers that fail any verification receive no private inference requests. `binaryHash` *is retained as drift telemetry and for transparency-log matching but is not itself a hard trust gate (Section 6.1).*

The coordinator selects providers using a cost-minimization scheduler. For each request it collects every eligible provider, computes an estimated completion time, and selects the lowest-cost candidate.

8.5 Routing Gates

Before a provider becomes a candidate it must pass the routing gates applied by `providerPassesRoutingGatesLocked`:

1. Catalog membership: the provider advertises an allowed build of the requested model.
2. Dispatch-load cooldown: skip a provider-model pair that recently failed to load with "insufficient memory."
3. Inference-error cooldown: a shape-keyed circuit breaker for repeated provider-side 5xx failures.
4. Status not `offline` or `untrusted`.
5. Private-only machines are excluded from public traffic unless serving their owner's self-route request.
6. Hardware-trust floor: public traffic must meet the configured minimum trust level; self-route to an owned machine relaxes this.
7. Runtime verified: the provider's reported runtime hashes match the known-good manifest.
8. Private-text support: `providerSupportsPrivateTextLock` requires an attested X25519 key, encrypted response chunks, anti-debug, core-dump disabling, environment scrubbing, and—once enforcement begins—APNs code-identity attestation.
9. Challenge freshness: the last attestation challenge verified within the maximum age window.
10. Trait eligibility: vision-capable builds for media requests, version floors for tool requests, and render-broken build fences.

8.6 Cost Function

For each eligible provider the scheduler builds a candidate with estimated completion time

$$\text{costMs} = \text{stateMs} + \text{queueMs} + \text{pendingMs} + \text{backlogMs} + \text{thisReqMs} + \text{hea} \tag{2}$$

where:

- stateMs: slot-state penalty (0 for `running`/`idle`, 30 s for `unknown`, 20 s for `idle_shutdown`, ineligible for `crashed`/`reloading`)
- queueMs: effectiveQueue $\times$ 3,000 ms
- pendingMs: totalPending $\times$ 750 ms
- backlogMs: tokens already committed divided by effective decode TPS
- thisReqMs: promptTokens/prefillTPS + maxTokens/effectiveTPS
- healthMs: penalty from memory pressure, CPU utilization, thermal state, and GPU utilization

Effective decode TPS is resolved in priority order from provider-reported observed EWMA, fleet median TPS for the same model and chip family, and a load-scaled benchmark fallback. The winner is the candidate with lowest cost; near-ties are broken by effective queue depth, then total pending load, then randomization among equivalent candidates to avoid hot-spotting. Capacity is reserved atomically before the provider is returned.

8.7 Reputation and Health

Provider reputation and health feed into the health term of the cost function rather than a multiplicative score. The health term captures memory pressure, CPU utilization, thermal throttling, and GPU utilization reported in provider heartbeats.

9 Formal Security Analysis

9.1 SIP Runtime Immutability

The security of our architecture depends on the property that SIP, once verified at process startup, cannot be disabled during the process lifetime.

Lemma 1 (SIP State Persistence)**.** *SIP state* $S \in \{\textsc{enabled}, \textsc{disabled}\}$ *is stored in NVRAM and is immutable to all userspace and kernel code in the normal macOS boot environment.*

Proof. The SIP state variable resides in the machine's NVRAM and is read by the Boot ROM during the secure boot chain. The only utility that modifies this variable is `csrutil`, which Apple restricts to execution within the Recovery Mode boot environment $\mathcal{R}$. In the normal macOS environment, `csrutil disable` returns an error. No other userspace or kernel API exposes write access to the SIP NVRAM variable when SIP is enabled, as SIP protects the NVRAM variables that control it—a self-reinforcing property. □

Theorem 1 (SIP Runtime Immutability)**.** *Under Assumption 1 (no unpatched kernel vulnerabilities), if SIP is verified as enabled at process startup time* t_0*, then SIP remains enabled at all times* $t \in [t_0, t_{end}]$ *during the lifetime of that process.*

Proof. By Lemma 1, SIP state is immutable in the normal boot environment. The only mechanism to change SIP state requires:

1. Reboot into Recovery Mode environment $\mathcal{R}$
2. Execute `csrutil disable` in $\mathcal{R}$
3. Reboot back to the normal environment

Step 1 requires a hardware reboot. Let $\mathcal{E}(t)$ denote the set of all running processes at time t. A reboot at time t_r terminates every process:

$$\forall\, p \in \mathcal{E}(t_r^-) : \ \neg\,\text{alive}(p,\, t_r^+) \tag{3}$$

The provider process $p^* \in \mathcal{E}(t_0)$. For p^* to observe $S = \textsc{disabled}$, a reboot must occur. But a reboot terminates p^*. Therefore:

$$S(t_0) = \textsc{enabled} \implies \forall\, t \in [t_0,\, t_{\text{end}}] : \ S(t) = \textsc{enabled} \tag{4}$$

□

Corollary 1 (Single Verification Sufficiency)**.** *A single SIP verification at process startup provides a security guarantee for the entire process lifetime. Per-request SIP checks serve as defense-in-depth but do not detect state changes, because none can occur under the theorem's conditions.*

9.2 Complete Attack Surface Analysis

Table 10 enumerates all known software-based attacks against the provider process and their corresponding defenses.

9.3 Self-Reinforcing Security Properties

Several verification mechanisms exhibit *circular dependency* that strengthens rather than weakens the security model:

Proposition 1 (MDM Verification Circularity)**.** *Spoofing the MDM* `SecurityInfo` *response to report SIP as enabled when it is disabled requires modifying system frameworks in* `/System/Library/`*, which requires SIP to be disabled. Therefore, a provider cannot simultaneously (a) have SIP disabled and (b) cause the MDM framework to report SIP as enabled.*

This circularity ensures that the MDM verification is unforgeable precisely in the cases where it matters most. Combined with Authenticated Root Volume verification (Section 6.2.2), which proves the entire system volume matches Apple's sealed build, this creates a defense-in-depth where each layer independently validates the others.

10 Hypervisor Memory Isolation

The software-enforced protections in Section 4 defend against observation through operating system interfaces. RDMA over Thunderbolt 5 (available in macOS 26.2+) permits a physically connected Mac to read host memory at 80 Gb/s via DMA, bypassing PT_DENY_ATTACH, Hardened Runtime, and SIP entirely. RDMA is disabled by default and requires a Recovery OS boot to enable. The provider reports `rdmaDisabled` and `hypervisorActive` in every attestation challenge response, and the coordinator records these values for telemetry and future policy enforcement.

10.1 Current Status

In the current system, `hypervisorActive` is informational rather than a hard routing gate. Single-node inference is the supported security boundary today: a provider with RDMA disabled relies on the kernel-enforced access-path elimination described in Section 4.2. Providers that enable RDMA without hypervisor isolation are not currently routed private text traffic as a matter of operational policy, but the coordinator does not yet enforce a strict hypervisor gate in the production scheduler.

10.2 Planned Hypervisor Isolation

When enabled, Apple's Hypervisor.framework creates a lightweight virtual machine with ARM Stage 2 page tables. These page tables insert a hardware-enforced translation layer between the host's virtual addresses and physical memory. Inference memory mapped into the VM

Table 10: Complete software attack surface on Apple Silicon. All listed attacks are blocked under Assumptions 1–2. The final row represents the residual physical attack.

Attack Vector	Defense Mechanism	Enforced By
Attach debugger (lldb, dtrace, Instruments)	`ptrace(PT_DENY_ATTACH)` at process startup	Kernel
Read memory via Mach APIs (`task_for_pid`, `mach_vm_read`)	Hardened Runtime without `get-task-allow` entitlement	Kernel
Intercept IPC between processes	No IPC exists—inference is in-process	Architecture
Modify provider binary to add logging	Code signing + SIP: macOS refuses modified signed binaries	Kernel + SIP
Replace binary with malicious version	APNs code-identity attestation: only the genuine, team-signed app can receive the push challenge	Coordinator + Apple
Inject malicious runtime package	No embedded interpreter; Swift binary links only signed `mlx-swift-lm`	Process + SIP
Load unsigned kernel extension	SIP blocks all unsigned kexts on Apple Silicon	SIP
Modify kernel code at runtime	Kernel Integrity Protection: hardware denies writes to kernel pages after boot	Hardware
Disable SIP	Requires reboot into Recovery Mode, terminating all processes (Theorem 1)	Hardware
Read physical memory via `/dev/mem`	Device node does not exist on Apple Silicon	Hardware
DMA-based extraction via Thunderbolt/PCIe	Per-device IOMMU (DART) with default-deny; unauthorized DMA triggers kernel panic	Hardware
RDMA over Thunderbolt 5 (80 Gb/s)	RDMA status reported in attestation challenge-response; single-node inference is the supported boundary today (Section 10)	Hardware + policy
Physical memory probing	*LPDDR5x soldered into SoC package; desoldering is destructive*	*Physical*

would reside at guest physical addresses that are invisible to RDMA, which operates on host physical addresses.

The intended design is:

1. Create a Hypervisor VM via `hv_vm_create()`
2. Allocate a memory pool via `mmap` sized to the model working set
3. Align the pool to 16 MB boundaries and map it in 16 MB chunks via `hv_vm_map()`
4. Serve Metal buffer allocations from this pool using `makeBuffer(bytesNoCopy:)`

A fail-closed policy would refuse inference requests if the VM-mapped pool is exhausted rather than falling back to unprotected memory. Multi-node RDMA sharding (Section 15) would require both nodes to have RDMA enabled and hypervisor isolation active, with activations transferred over encrypted RDMA.

10.3 Experimental Validation

We validated the hypervisor approach on an Apple M4 Max (128 GB, macOS 26.3) with the following results:

- **Performance overhead**: 0% on Qwen3.5-27B (50.1 GB, BF16). Baseline: 8.2 tok/s; with hypervisor and 100% memory VM-mapped: 8.0 tok/s (within measurement noise).
- **Memory coverage**: 100% for all tested quantization formats (BF16, FP16, 4-bit, 8-bit, MoE).
- **Pool mapping latency**: 60 GB pool mapped in 2.5 ms (3,840 × 16 MB chunks).
- **Metal GPU compute**: Verified cor-

rect on VM-mapped memory—Metal's `makeBuffer(bytesNoCopy:)` creates GPU-accessible buffers backed by the VM-mapped pool.

These measurements establish that hypervisor isolation can be deployed without performance penalty, but production enforcement is not yet active.

11 Implementation

The system is implemented in approximately 14,000 lines of code across four languages:

Table 11: Implementation components and languages.

Component	Language	Lines
Provider CLI	Swift	6,000
Provider core foundation	Swift	1,200
Coordinator	Go	4,000
Console UI	TypeScript (Next.js)	2,500
macOS application / AppKit host	Swift/SwiftUI	800
Total		**∼14,500**

Coordinator. Implemented in Go. Runs within a Confidential VM (AMD SEV-SNP) with cryptographic container attestation. Exposes an OpenAI-compatible HTTP API for consumers and WebSocket endpoints for provider registration and messaging. The MDM server is MicroMDM [18] with SCEP certificate issuance; MDA hardware attestation is requested over MDM. The APNs code-identity attestor is implemented in Go and pushes encrypted challenges through Apple's HTTP/2 notification service.

Provider agent. Implemented as a Swift CLI (`darkbloom`). Manages the WebSocket connection to the coordinator, Secure Enclave attestation and challenge signing, APNs registration and push handling, hardware detection, system metrics collection, model manifest downloads, and in-process inference via `mlx-swift-lm` [19]. There is no embedded Python interpreter, no Rust provider binary, and no local inference server subprocess.

Secure Enclave module. Implemented in Swift using CryptoKit's `SecureEnclave.P256.Signing.PrivateKey` and the Security framework's persistent SE-backed key. Builds the signed attestation blob and signs challenge responses and code-identity nonces.

Provider installation. A single shell command downloads a self-contained signed and notarized bundle including the Swift provider binary, AppKit host, and embedded provisioning profile. No system Python or package manager is required. The script detects the device serial number, requests a combined enrollment profile from the coordinator, and opens the profile for user review and installation.

12 Evaluation

12.1 End-to-End Validation

We validated the complete system on two Apple Silicon machines simultaneously: an AWS `mac2-m2.metal` instance (Apple M2, 24 GB unified memory, macOS 14.8.4) and a local M4 Max (48 GB unified memory, macOS 26.3). The validation sequence:

1. Both providers enrolled via combined profiles (SCEP + MDM + ACME payload, AccessRights = 1041)
2. Both generated Secure Enclave attestation blobs with serial numbers
3. Coordinator verified SE P-256 signatures → Layer 1 (self-signed trust)
4. Coordinator queried MDM; SecurityInfo confirmed SIP = true, SecureBoot = full, AuthenticatedRootVolume = true for both → Layer 2
5. Coordinator requested MDA via MDM `DeviceInformation`; Apple's attestation servers returned DER certificate chains signed by Apple Enterprise Attestation Root CA for both devices, with `FreshnessCode` matching the SHA-256 of each SE public key → Layer 3
6. Coordinator verified Apple certificate chains against embedded root CA; serial numbers cross-checked against attestation blobs
7. Both providers completed APNs code-identity attestation: encrypted nonce pushed, decrypted with the in-memory X25519 key, and signed by the SE key → Layer 5
8. Both providers reached fully verified status (all five layers)
9. Consumer inference requests routed to both providers via the cost-minimization scheduler
10. M4 Max: 92 tok/s decode (Qwen3.5 9B, Q4); M2: 22 tok/s
11. 8 concurrent requests completed with 100% success rate across both providers; 6,517 tokens generated in 76.2 s
12. Consumer disconnect triggered cancel propagation within 1 s: coordinator → provider → backend
13. Full attestation proof (including Apple MDA certificate chains) exposed at public API endpoint for independent consumer verification

12.2 Security Verification

Security overhead is negligible. The SIP check (12 ms per request) is the largest per-request cost, dominated by sub-

Table 12: Security and performance measurements on two providers.

Verification	Result
PT_DENY_ATTACH	lldb attach denied
SIP status (both devices)	Enabled
Secure Boot (both devices)	Full
Authenticated Root Volume	Enabled
MDM SecurityInfo cross-check	Match
Apple MDA certificate chain	Valid (both devices)
MDA serial cross-check	Match
MDA freshness code / SE key binding	Match
APNs code-identity attestation	Passed (both devices)
RDMA status (both devices)	Disabled
Challenge-response (5-min interval)	All passed
Performance	**Value**
Decode throughput (9B, M4 Max)	92 tok/s
Decode throughput (9B, M2)	22 tok/s
Time to first token	1.4 s
SIP check overhead	12 ms
Reconnect after disconnect	$<$1 s
Concurrent requests per provider	4
8 concurrent, 0 failures	100%
Hypervisor overhead (27B, M4 Max)	0%
Hypervisor pool mapping (60 GB)	2.5 ms
AES-256 encrypt+decrypt (16 MB)	0.37 ms

process invocation of `csrutil status`. By Corollary 1, this check could be performed once at startup without loss of security guarantee; the per-request check serves as defense-in-depth.

13 Economic Feasibility Analysis

We analyze the economic properties of the system to establish that distributed inference on consumer hardware is economically viable at the price points required for adoption.

13.1 Marginal Cost Model

For providers with hardware already purchased, the marginal cost per million output tokens is:

$$C_{\text{local}} = \frac{P_{\text{elec}}}{R_{\text{tok}} \times 3{,}600} \times 10^6 \tag{5}$$

where P_{elec} is electricity cost (\$/hr) and R_{tok} is decode throughput (tok/s). At U.S. average residential \$0.15/kWh, an Apple Silicon workstation draws $\sim$100 W under inference load (\$0.015/hr) and 5 W at idle.

13.2 Throughput Measurements

Observation: Mixture-of-Experts (MoE) models are disproportionately efficient on unified-memory hardware. Qwen3.5 35B-A3B (3B active parameters) achieves 101 tok/s on M4 Max—faster than the dense Qwen3.5 27B at 21 tok/s, despite a larger total parameter count. For MoE architectures, only active expert weights traverse the memory bus per token.

Table 13: Decode throughput (tok/s, Q4 quantization) on Apple Silicon. † = measured on live system; others bandwidth-scaled at 65% efficiency.

Model	Architecture	M4 Max 546 GB/s	M3 Ultra 819 GB/s
Qwen3.5 9B	Dense	92†	94
Qwen3.5 27B	Dense	21	32
Qwen3.5 35B-A3B	MoE (3B active)	101	152
Llama 3.1 70B	Dense	8	13
Qwen3.5 122B-A10B	MoE (10B active)	25	35
MiniMax M2.5 230B	MoE (10B active)	—	40

"—" = does not fit in memory.

13.3 Cost Comparison

Cloud inference providers price based on total parameter count. Local inference cost depends only on *active* parameters per token. For MoE models, this creates 6–32$\times$ cost advantages:

Table 14: Cost comparison: cloud inference vs. local (electricity only).

Model	Cloud \$/M	Local \$/M	Ratio
Qwen3.5 35B-A3B	\$1.30	\$0.04	32$\times$
Qwen3.5 122B-A10B	\$2.08	\$0.09	23$\times$
Qwen3.5 27B	\$1.56	\$0.20	7.8$\times$
MiniMax M2.5 230B	\$0.95	\$0.16	6$\times$
Llama 3.1 70B	\$0.30	\$0.49	0.6$\times$

Dense models exceeding 32B parameters favor cloud infrastructure: H100 clusters with 3.35 TB/s memory bandwidth achieve 6$\times$ higher throughput than M4 Max (546 GB/s) for bandwidth-bound decoding. The economic sweet spot for distributed inference is MoE models with small active parameter counts on large unified-memory hardware.

14 Limitations

Kernel vulnerability assumption. Our security model (Theorem 1) assumes no unpatched kernel vulnerabilities (Assumption 1). A macOS kernel zero-day could bypass SIP, Hardened Runtime, or KIP, undermining the trust model. This is an inherent limitation of software-enforced security boundaries; only hardware TEEs provide immunity to kernel-level attacks. We rely

on Apple's track record of rapid patching.

Timing side channels. Our architecture protects token *content*—the provider cannot observe what tokens are generated. However, token *timing* is observable. Network packet intervals reveal approximate prompt length (from prefill duration), response length (from output packet count), and relative generation difficulty (from inter-token delays). We do not implement constant-time inference. Mitigations include response buffering and random jitter. This limitation is shared with Apple PCC and all non-TEE inference systems.

APNs availability. APNs code-identity attestation requires a logged-in macOS GUI Aqua session and best-effort background push delivery. Headless or login-screen providers cannot receive pushes and therefore cannot become code-attested, so they are not routed private text traffic once enforcement is enabled. Push delivery can be throttled by Apple's device budget, although the coordinator can fall back to alert mode if necessary.

Hypervisor/RDMA. Hypervisor memory isolation and RDMA-based multi-node inference are validated experimentally but not enforced as a production routing gate today. Single-node inference with RDMA disabled is the supported security boundary.

MDM/MDA availability. Hardware attestation requires devices to be enrolled with an organization that has Apple attestation authority (MicroMDM with an Apple push certificate). Expanding to arbitrary consumer devices requires integration with Apple Business Manager or an equivalent enrollment pathway.

15 Future Work

Request anonymization. OHTTP (RFC 9458) relays and RSA blind signatures would prevent the coordinator from linking consumer identity to request content, achieving non-targetability—Apple PCC's fourth requirement.

Hypervisor enforcement. Elevate `hypervisorActive` from telemetry to a hard routing gate for providers that report RDMA enabled. This would enable secure single-node inference even when Thunderbolt 5 RDMA is turned on, and is a prerequisite for multi-device sharding.

Multi-device inference. Thunderbolt 5 RDMA ($<$50 μs latency, macOS 26.2+) enables model sharding across multiple provider machines, extending capacity to 400B+ parameter models. Single-machine validation of the required components is complete: hypervisor memory isolation (Section 10) protects inference memory from RDMA, and AES-256 activation encryption at 42.5 GB/s (0.37 ms per 16 MB tensor) is fast enough to overlap with GPU inference compute (2.9% effective overhead with double-buffered pipelining). Two-machine validation of RDMA memory visibility—confirming that RDMA cannot access VM-mapped memory across a Thunderbolt link—remains as future work.

Encrypted model execution. Model weights encrypted at rest, with decryption keys released only to providers whose full attestation chain verifies. The provider's hardened process decrypts weights in memory, performs inference, and wipes them. The same protections that guard inference inputs also guard model intellectual property.

Transparency log for binary hashes. APNs code-identity attestation proves the genuine App ID/Team ID but not the exact binary version. A public transparency log of blessed `cdhash` values, combined with reproducible builds, would let consumers pin or range-accept specific provider releases while retaining the non-self-reportable APNs gate.

General private compute. The access path elimination technique is not specific to inference. Any computation that can be embedded in a hardened process on Apple Silicon—fine-tuning on private data, evaluation on proprietary datasets, confidential data pipelines—can leverage the same security guarantees. The multi-layer attestation architecture generalizes to any workload where computation must remain confidential from the machine's owner, positioning the distributed network as a general-purpose private compute layer on consumer hardware.

16 Conclusion

We have presented a distributed inference system in which consumers get private AI computation on machines they do not own or control. The machine owners—who have root access and physical custody—cannot observe the prompts or responses processed on their hardware.

The approach works by systematically eliminating every software path to inference data: in-process Swift execution using `mlx-swift-lm` removes inter-process communication, kernel-enforced isolation blocks memory inspection, and operating system integrity guarantees are provably immutable at runtime (Theorem 1). Hypervisor Stage 2 page tables have been validated as a future hardware-level isolation mechanism for RDMA-based DMA attacks with zero measured performance overhead. A multi-layer attestation architecture—Secure Enclave signatures, MDM-based independent verification, Apple MDA with Apple-signed certificate chains, MDA nonce-based SE key binding, continuous challenge-response with fresh security state, and APNs code-identity attestation—provides defense-in-depth where each layer independently verifies properties that the others cannot.

The residual attack surface reduces to physical memory probing of LPDDR5x soldered into Apple's System-on-Chip package, even when RDMA is enabled. This is the same threat model accepted by Apple's Private Cloud Compute for Siri and Apple Intelligence queries. Our

system achieves equivalent privacy guarantees on a distributed network of independently owned machines.

The system is validated end-to-end on production hardware with live inference, demonstrating negligible security overhead. The economic analysis establishes that Mixture-of-Experts models create 6–32× cost advantages on unified-memory hardware, making distributed private inference economically viable. The access path elimination technique generalizes beyond inference to any computation requiring confidentiality on third-party consumer hardware.

References

[1] Apple Security Engineering and Architecture. Private Cloud Compute: A new frontier for AI privacy in the cloud. *Apple Security Research*, 2024.

[2] Akash Network. The Decentralized Cloud. https://akash.network/, 2024.

[3] io.net. The Internet of GPUs. https://io.net/, 2024.

[4] Ritual. Infernet: Decentralized AI Infrastructure. https://ritual.net/, 2024.

[5] Y. Rao et al. Bittensor: A peer-to-peer intelligence market. *arXiv*, 2023.

[6] Intel Corporation. Intel Trust Domain Extensions (TDX). *Intel Architecture Specification*, 2023.

[7] D. Kaplan et al. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. *AMD White Paper*, 2020.

[8] NVIDIA. Confidential Computing on H100 GPUs. *NVIDIA Technical Blog*, 2023.

[9] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of STOC*, pages 169–178, 2009.

[10] A. C. Yao. Protocols for secure computations. In *Proceedings of FOCS*, pages 160–164, 1982.

[11] J. Sun et al. zkLLM: Zero knowledge proofs for large language models. *arXiv:2404.16109*, 2024.

[12] Lagrange Labs. DeepProve-1: First full LLM zkML proof. 2025.

[13] Apple Inc. Mobile Device Management Protocol Reference. *Apple Developer Documentation*, 2024.

[14] Apple Inc. Secure Enclave. *Apple Platform Security Guide*, 2024.

[15] Apple Inc. Managed Device Attestation. *WWDC22 Session 10143*, 2022.

[16] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT*, pages 159–176, 2012.

[17] Smallstep. step-ca: An online certificate authority and ACME server. https://smallstep.com/docs/step-ca/, 2024.

[18] MicroMDM Project. MicroMDM: A macOS MDM server. https://micromdm.io/, 2024.

[19] Apple ML Research. MLX: An array framework for Apple Silicon. https://github.com/ml-explore/mlx, 2024.

[20] Darkbloom: Distributed Private Inference on Apple Silicon. Open-source implementation. https://github.com/Layr-Labs/d-inference, 2026.