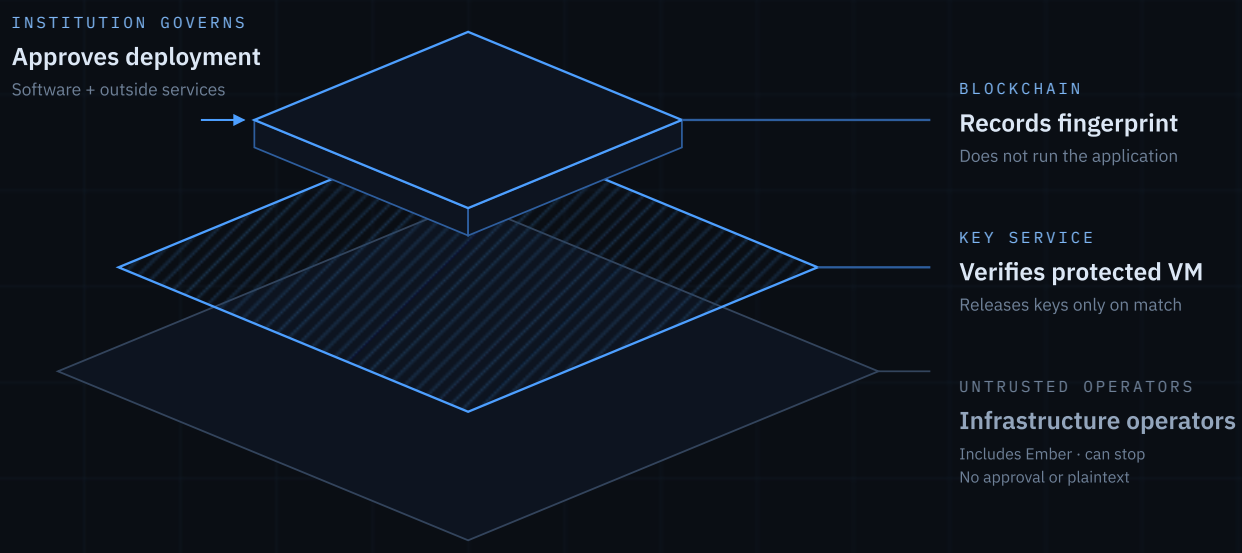
Institution-Governed AI with Cryptographic Controls

The institution approves a deployment and records its fingerprint on a blockchain under a pseudonymous address. A key service releases keys only to a matching protected virtual machine; approved outside services may read data sent by permitted operations.

THE CRYPTOGRAPHIC CONTROL PATH



VERSION	DATE	RUNTIME	CONTACT
0.6.3	July 2026	Attested CVM · Intel TDX	embersovereignty.com/briefing

ABSTRACT

The data that makes an AI product useful to an institution is often the data the institution cannot let the vendor see. Today, institutions generally choose between a vendor-hosted service, where the vendor can read the data, and a separate deployment the institution must operate itself. That tradeoff stalls otherwise valuable products in security review.

This architecture creates a third model. A vendor brings software components fixed to exact versions. The institution approves that software and its permitted outside services, and a pseudonymous account the institution controls records the deployment's digital fingerprint on a blockchain. The application itself runs in a dedicated, protected virtual machine, not on the blockchain. A key-management service releases encryption keys only to a protected virtual machine whose verified fingerprint matches the institution's recorded approval. The infrastructure operator cannot read the virtual machine's memory or storage. An approved outside service can read the data that an approved operation sends to it, and each contact creates a signed, content-free receipt.

The vendor maintains one product and proposes updates. The institution controls approvals. Infrastructure operators, including Ember, transport and run the deployment but cannot authorize software or read plaintext institutional data. They can stop the virtual machine, while underlying infrastructure retains the metadata and rollback powers disclosed in Section 15. This paper explains the architecture, its implementation status, the assumptions it relies on, and where its protections end.

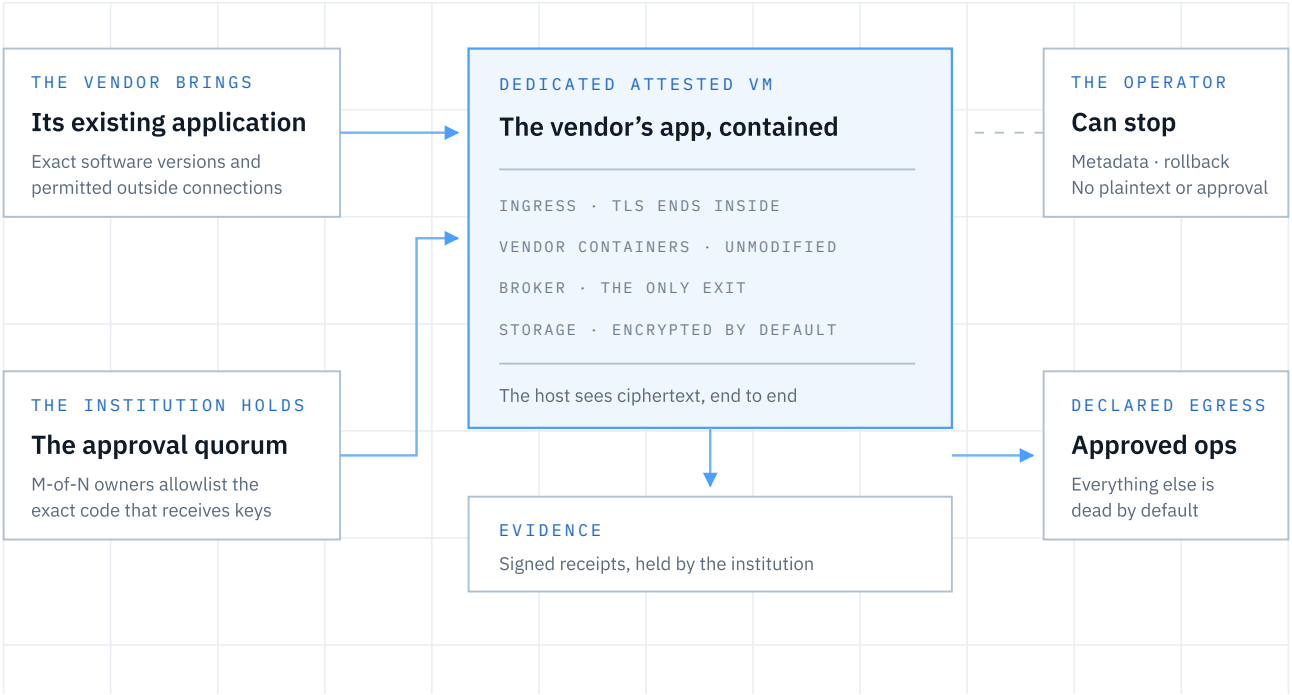
The runtime, verifier, and reproducible-build sources are referenced in Appendix D.

CONTENTS

Custody is structural, not contractual.

The vendor's existing application moves into a dedicated, protected virtual machine with four configuration changes. The institution approves the software and outside services that may handle its data. Infrastructure operators can run or stop the deployment but cannot authorize software or read plaintext.

THE VENDOR BRINGS · THE INSTITUTION APPROVES · THE PLATFORM ENFORCES



1. The institutional problem and the parties

An AI product is only institution-ready if it can answer one question with evidence: **who holds authority over our data while the product works on it?** For the vendor, answering it turns a recurring security exception into a property of the deployment rather than a promise about operations.

1.1 The parties

This paper uses five nouns deliberately. The **institution** is the governed organization whose data, policy, and audit obligations are at issue. The **vendor** builds the AI application. The **operator** runs the machines and network the application executes on. On the hosted platform, that means Ember and the datacenter beneath it. An **inference provider** operates a model endpoint the application calls. A **user** is a person or agent acting for the institution. In commercial settings the institution is often the vendor's buyer: an enterprise, a law firm, an agency, any governed organization. Calling every party a "customer" obscures who actually controls each layer.

1.2 The problem

Modern AI applications are built to read everything. Documents, repositories, deal rooms, and records flow into the vendor's service as plaintext, where retrieval pipelines, prompt assembly, and agents operate before and after each model call. The vendor's answer to the security question is contractual: no training on institutional data, zero data retention, access restricted to named roles. These commitments matter, but they remain **policy**. The architecture does not prevent the vendor, an attacker inside the vendor, a compelled disclosure, or a future acquirer from reading the data. A reviewer can audit the policy but cannot verify the property.

The traditional alternatives trade one problem for another. On-premises and bring-your-own-cloud deployments move the vendor's code inside the buyer's perimeter, but the data remains readable there, the perimeter must be built and maintained by the buyer, and the vendor's release pipeline still ships unverified code into it. Section 14 treats this comparison structurally.

The result is a familiar stall: the product wins the evaluation and loses months in security review, and the most regulated institutions are exactly the ones who cannot proceed.

1.3 The claim, stated precisely

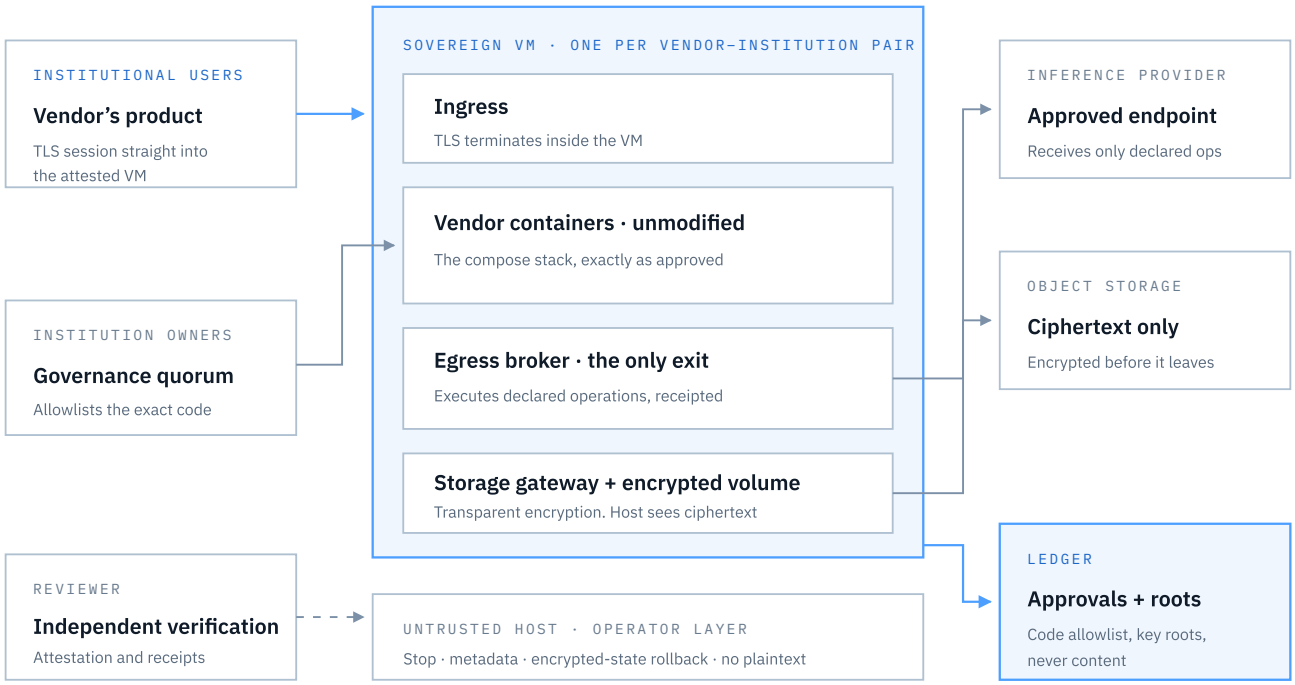
The Ember platform runs a vendor's existing application inside a sovereign VM: a dedicated, hardware-attested confidential virtual machine, provisioned per vendor–institution pair. In every institution-governed VM, the following holds:

- The vendor's application runs essentially unchanged, down to its containers, processes, filesystem, and internal networking. The modifications required are enumerated in Section 2 and fit on an index card.
- The host **provably cannot read it**. Memory is encrypted by the CPU, disks are encrypted with keys the host never holds, and TLS terminates inside the VM. What the operator keeps is scheduling and availability: it can stop a VM, refuse it resources, or snapshot and roll back its *encrypted* state, and it observes traffic and resource metadata. It never sees plaintext and never holds keys.

- The institution alone **governs what code the keys follow**. A VM's secrets are released only to code whose exact measurement an M-of-N quorum of institutional owners has approved on an append-only ledger. Neither the vendor nor Ember can substitute code and retain access to the data.
- The only way out of the box is a **declared, receipted operation**. Egress is default-deny on every network stack. An egress broker is the sole exit, and it executes only the operations the institution approved, to the recipients it approved.
- **Storage works like a regular computer**, encrypted transparently. The application writes files. The host reads ciphertext.
- Every session leaves **signed, content-free receipts**, delivered outward to the institution at session time and verifiable by a third party without trusting the vendor or Ember.

A receipt certifies only the properties its session actually used. The boundaries that remain, including what an approved inference provider receives and the storage-rollback limit, are stated in Section 15 rather than implied away.

FIG. 1 **System overview of a sovereign VM. Users reach the vendor's product over a TLS session that ends inside the attested VM. The vendor's containers run unmodified between a locked ingress and a default-deny egress. The host beneath sees ciphertext and traffic metadata, and the ledger carries approvals and key roots, never content.**



1.4 What is new here

None of the ingredients is exotic. Confidential VMs are shipping silicon, measured boot and remote attestation are standardized, and Docker Compose is a common way to describe multi-container server applications. The

contribution is the composition, aimed at a deployment problem: the platform assembles these pieces so that an *existing* application gains the custody properties above with configuration-level changes, and so that every property is either enforced by hardware, checked by a third party, or stated as a limit in Section 15. The live substrate is built from open-source, reproducible code. The platform components specified here are designed to be open source and reproducibly built, so their measurements can correspond to source a reviewer can inspect.

> **Implementation status.** The substrate this paper leans on is exercised by a live deployment: the dedicated CVM and its attestation (Sections 4–5), the on-chain code authority, and key release gated on it (Sections 6, 13). The governance transcript in Section 13.3 is read from that deployment. The platform components above the substrate (the manifest compiler, egress broker, receipts, and Tier 2 storage gateway) are specified by this paper and tracked for implementation. The paper marks which is which where the difference matters.

2. Moving the vendor's existing application

The vendor brings the multi-container application it already runs, with each component fixed to an exact version. The platform adds proof of the running software, institutional approval, protected ingress, restricted outside connections, encrypted storage, and independent audit records.

2.1 A full VM, not a function runtime

The VM is a real machine. The application keeps its processes, its filesystem, its localhost networking between containers, its schedulers and queues, its choice of what to hold in RAM versus write to disk. There is no SDK to adopt, no proprietary runtime API, no restriction to a request–response shape. This is the property that makes lift-and-shift honest: software written for an ordinary Linux host runs in the VM the way it ran outside it, because the VM *is* an ordinary Linux host, one whose memory and disk happen to be encrypted against everyone but the code inside.

2.2 The index card

Four changes, all at the configuration level:

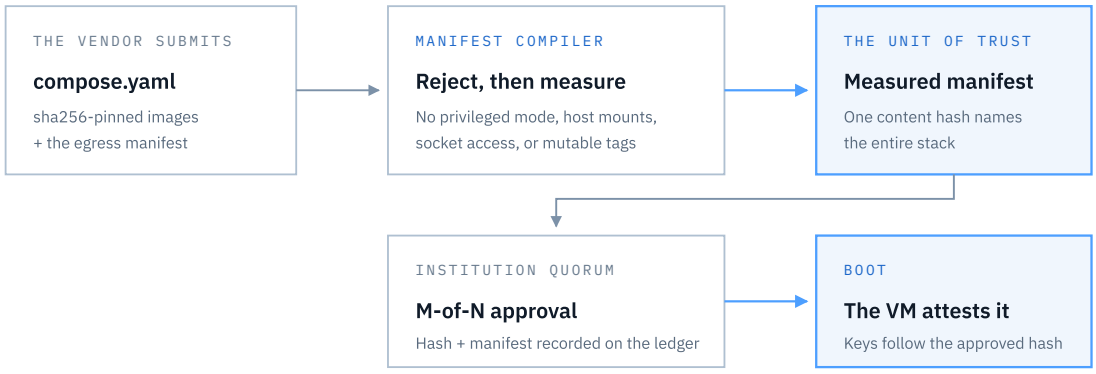
1. **Point outbound API calls at the broker.** The broker exposes provider-compatible surfaces inside the VM: an OpenAI-compatible chat-completions endpoint, the Anthropic Messages API, and pinned read-only connectors. For most applications this is one environment variable, such as `OPENAI_API_BASE`. Provider credentials are injected by the broker outside the vendor's containers, and the application's own key configuration is left empty (Section 9).
2. **Declare your egress.** The vendor writes an egress manifest: every outbound operation the application needs, by type and destination. The institution approves the manifest alongside the code itself. Anything undeclared is dead by default (Section 8).
3. **Storage: nothing.** Filesystem writes hit an encrypted volume as-is. Applications that need durable, portable data point their existing S3 configuration at the in-VM gateway (Section 11).
4. **Strip phone-home.** Telemetry, update checks, and crash reporters are disabled by configuration where possible, and the manifest compiler rejects the telemetry components declared in the stack's configuration.

The compiler cannot see inside an opaque image, so anything it misses is blocked by default-deny egress at the network layer: an undeclared phone-home fails to connect rather than failing review. An application that constantly fails outbound calls is a noisy tenant, so fix it in configuration.

2.3 What the platform compiles

Ember does not run the vendor's compose file directly. A manifest compiler transforms it into the measured production manifest the VM boots, and the compiler is opinionated: it rejects privileged mode, host namespaces and mounts, access to the container socket, custom DNS, added capabilities, mutable image tags, and telemetry or update agents declared in the stack's configuration. What survives compilation is a stack that can be measured, attested, and contained. The compiled manifest's hash is the unit of governance: it is what the institution approves, what attestation quotes, and what receipts name.

FIG. 2 **Onboarding. The compose stack and egress manifest compile into one measured manifest. The institution's quorum approves its hash on the ledger. The VM boots it, attests it, and receives keys only if the measurement matches the approval.**



2.4 What the vendor does not do

The vendor does not build attestation, ingress TLS, key management, network policy, receipts, or encrypted storage into its application. Those are the platform's job, identical for every tenant. The vendor also does not operate the deployment: there is no shell into a running VM, no live debugger, and no path to push an update into an institution's VM without that institution's recorded approval (Section 7). The honest statement of the trade is that the vendor gives up operational reach into this one deployment in exchange for a custody claim its buyers can verify.

3. Threat model and trust assumptions

A security architecture is defined by what it assumes as much as by what it prevents. This section states both, before any mechanism is described.

3.1 Adversaries considered

A1: The operator, including Ember. Whoever runs the control plane, the hosts, and the network under the VMs. A1 controls the hypervisor's scheduling, the physical machines, and every cable. It must not be able to read VM memory or disk, extract keys, forge attestation, or substitute code. Its residual powers are named in Section 15, not hidden: stopping a VM, observing traffic shape and timing, and rolling storage back to an earlier encrypted state.

A2: The vendor, honest-but-curious or compromised. The party with the most natural access in a conventional deployment: it wrote the code. A2 covers the vendor's ordinary operations, a breach of the vendor, an insider, a compelled disclosure served on the vendor, and a future acquirer. Against A2 the VM is a box the vendor shipped but cannot open: no shell, no telemetry, no unilateral update (Section 7).

A3: A compromised workload. The vendor's application itself, subverted at runtime by a supply-chain implant in a dependency or by an attacker steering the application through its own inputs. A3 executes *inside* the box with the data in plaintext. Containment bounds it: a compromised workload exfiltrates only through operations the institution approved, to recipients it approved, under receipt (Sections 8–9).

A4: A malicious upgrade. An attempt, by the vendor or whoever holds its release pipeline, to ship code into the VM that leaks content while presenting a compliant surface. Handled by the governance gate: unapproved code never receives the VM's keys (Section 13).

A5: A network attacker. Standard capability on every link: observe, replay, reorder, and inject traffic between users, the VM, providers, and the ledger.

A6: A history rewriter. An attempt to make a session un-happen by rolling the host's storage back or suppressing local logs. Receipts are delivered outward at session time, so the institution's copy survives the host (Section 12). The storage-rollback residual is disclosed in Section 15.

3.2 What is trusted

The trusted base is deliberately short, and each item is named so a reviewer can price it:

- **The silicon vendor's attestation root.** The CPU manufacturer's key signs the statement "this measurement is running inside a genuine confidential VM." This is the same root of trust that confidential-computing deployments at the major clouds rest on.[2]
- **The measured platform stack.** The VM image, ingress, broker, storage gateway, and key-management service are part of what attestation measures. They are open source and reproducibly built, so the measurement corresponds to reviewable source, and trusting them is trusting an audit, not an operator.

- **The pinned application build.** Attestation proves which code ran. It does not make that code correct. The vendor's images must be reviewed (by the vendor, the institution, or a third party), and the review is meaningful because the hash pins exactly what was reviewed.
- **The key-management service's root, as a floor.** VM keys derive from a root held inside attested hardware and governed on the ledger, never by an operator console. Its protection is itself TEE-grade, and Section 15 states this floor honestly rather than claiming it away.
- **The domain administrator, in v1.** Whoever controls the DNS zone a VM is served under (the infrastructure operator for platform-managed subdomains, the vendor for a custom product domain) is trusted not to redirect the name or obtain and serve a competing certificate. CAA narrows issuance and CT makes competing issuance publicly visible, but the zone administrator can change both. Section 10 states this as detection, not prevention.
- **The institution's endpoints and owner credentials.** A compromised user device reads content before it is sent. Endpoint security and custody of the owners' passkeys remain institutional responsibilities.
- **Approved recipients, to the extent the manifest admits them.** An inference provider that receives a declared operation reads what that operation sends, under the credential's terms. The platform makes the routing provable. It does not make an external provider blind.
- **Liveness of the ledger** used for approvals and key governance. Safety does not depend on it minute-to-minute, but auditability does.

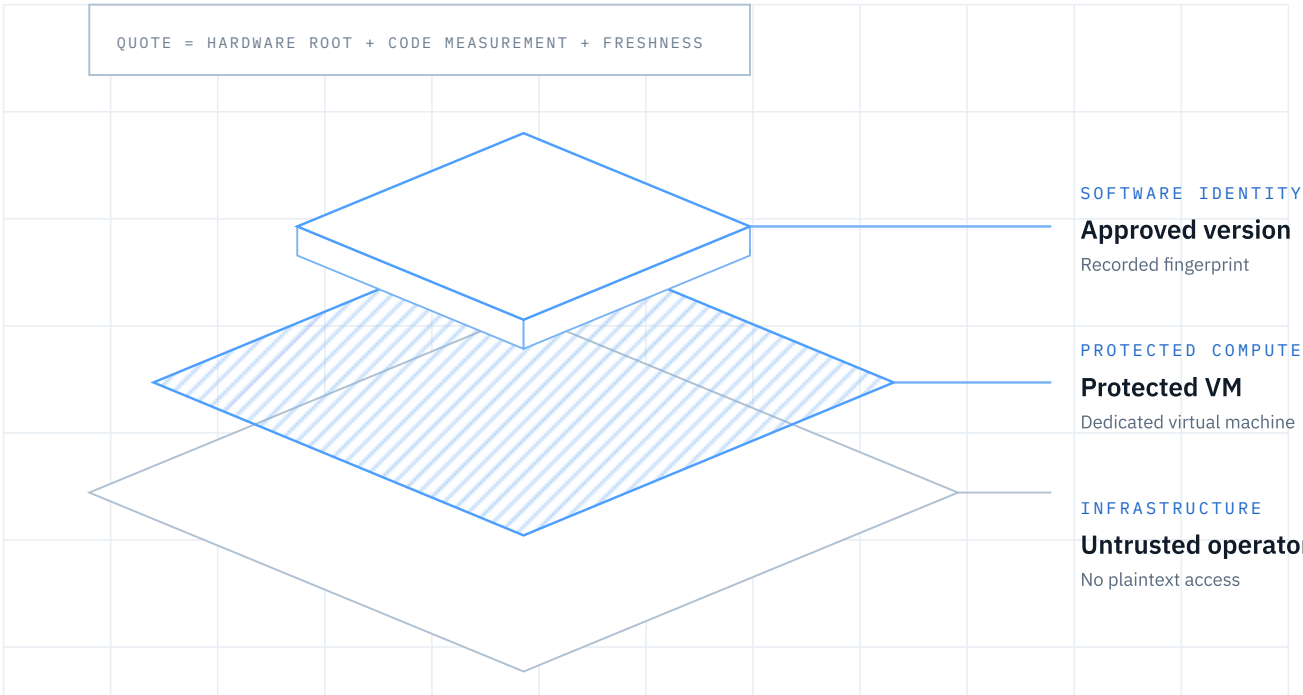
3.3 What is explicitly not assumed

No trust is placed in the operator's organization or employees, the host operating systems and hypervisors under the VMs, the vendor's backend or release pipeline (past the governance gate), or the network. None of these can read content, obtain VM keys, or produce a receipt that verifies.

Third-party compute is independently verifiable.

The protected virtual machine proves which software is running. Assurance follows independently verifiable evidence, not the operator’s brand. Encryption keys follow that evidence too.

THE VERIFIED COMPUTE STACK



4. The dedicated confidential VM

A sovereign VM is a trusted execution environment (TEE) in its virtual-machine form: a confidential VM (CVM) whose memory is encrypted by the CPU with keys the host never sees, and whose contents are inaccessible to the host operating system and hypervisor under the platform's threat model.[2][4] Remote attestation follows the evidence, appraisal, and relying-party pattern standardized by the IETF RATS architecture.[3]

The runtime profile is concrete rather than generic: sovereign VMs run on Intel TDX[2] under **dstack**,[11][12] an open-source confidential-container runtime that boots a measured OS image, runs the pinned compose stack on it, and anchors code authority and key release on the ledger. Each mechanism this paper claims is a named dstack component, or an Ember component measured alongside it.

The choice of a full CVM over a process-level enclave is what makes Section 2's lift-and-shift claim true rather than aspirational. A process enclave asks the application to be rebuilt around a restricted ABI. A CVM presents an ordinary machine (kernel, filesystem, container runtime) whose isolation boundary is drawn around the whole VM. Existing software runs unmodified, and the security argument concentrates on the small measured stack around it.

- **Memory encryption and isolation.** While the VM runs, its plaintext exists only inside CPU-encrypted memory. The operator of the machine sees ciphertext pages and cannot introspect the workload.
- **Remote attestation.** The CPU signs a *quote*, a statement rooted in the silicon vendor's key of exactly what booted inside the VM: firmware, the runtime OS image, and the compiled application manifest. A remote party verifies the quote against the manufacturer's published root of trust (Section 5).
- **Dedication.** One VM serves one vendor–institution pair. There is no co-tenant inside the trust boundary and no shared cache of another institution's sessions. Isolation between institutions is by construction, not by convention.

4.1 What the host actually sees

It is worth being concrete about the operator's view, because the entire custody claim lives there. The host sees encrypted memory pages, an encrypted block device, and TLS ciphertext entering and leaving, plus traffic metadata (packet timing, sizes, permitted destinations) and resource consumption. The host can: schedule the VM, migrate or snapshot its *encrypted* state, restore an earlier encrypted snapshot, refuse it CPU, and stop it. The host cannot: read or modify plaintext, inject code or devices that survive attestation, obtain the VM's keys, or produce a valid quote for code that is not running. Everything on the "can" list is an availability or metadata power, never a custody one. Section 15 prices each residual.

4.2 Session lifecycle

Unlike a per-session enclave, a VM is a standing deployment, and its lifecycle is the application's own. The confidentiality unit is the VM itself: attest at boot, release keys against the approved measurement, serve sessions over in-VM TLS, persist only to encrypted storage, and emit receipts outward as sessions complete. A stopped VM's plaintext ceases to exist. What remains is ciphertext on disk and the receipts the institution holds.

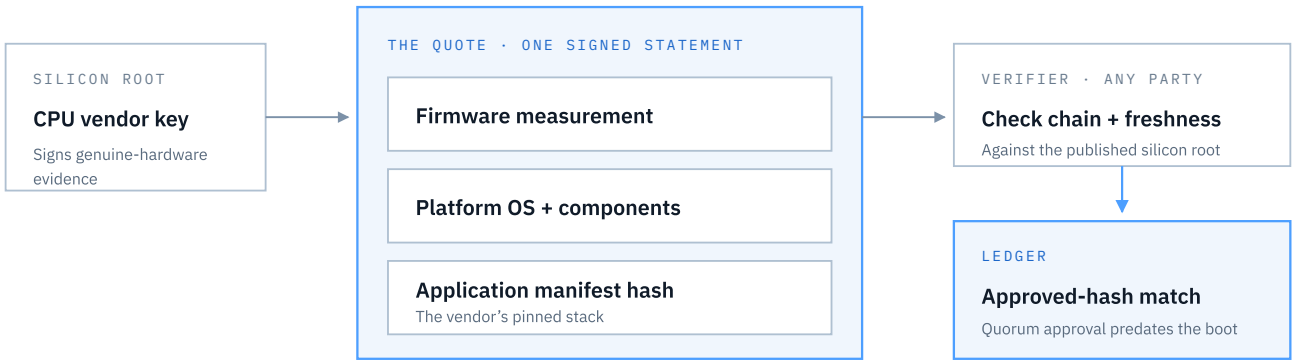
5. Attestation: proving what runs

Attestation answers two questions at once: “is this genuine confidential-computing hardware?” and “exactly what booted inside it?” The remaining question, whether *what booted* is *what was approved*, is closed by content addressing and the ledger.

5.1 The measurement chain

A VM’s quote covers a chain of measurements taken as the machine boots: the virtual firmware, then the runtime OS image and the platform components it supervises (ingress, broker, storage gateway, agents), and finally the compiled application manifest of Section 2.3, whose hash names the vendor’s exact container images and egress declarations. Change any byte at any layer and the measurement changes. The quote either names the approved stack or it does not. Because the platform layers are open source and reproducibly built, a reviewer can rebuild them from source and confirm the expected measurement independently. The operator’s word appears nowhere in the procedure.

FIG. 3 **The attestation chain. One hardware-signed quote covers firmware, the open-source platform stack, and the application manifest hash. A verifier checks the signature against the silicon vendor’s root and the measurement against the institution’s recorded approval.**



5.2 The evidence, field by field

On the shipping profile (dstack on Intel TDX), the quote a reviewer pulls is not an abstraction. A verifier checks, in order:

- **The TDX quote and its chain.** The quote’s signature chain validates against Intel’s published attestation root, and carries the TCB status of the platform that produced it. Out-of-date microcode or firmware surfaces in this field, not in a footnote.
- **The runtime measurement.** The measurement registers (MRTD and RTMR0–2) name the virtual firmware and the runtime OS image: kernel, initramfs, and the container supervisor. The image is open source and reproducibly built,[11] so the expected value is recomputable from source per release.

- **The application event log.** Boot-time events extended into the runtime register (RTMR3) record the application identifier (`app-id`), the **compose hash** (the digest of the compiled application manifest), the instance identity, and the key source. The verifier replays the log, confirms it reproduces the quoted register value, and reads each claimed field from the log rather than from anyone's word.
- **Freshness.** The quote binds a caller-supplied nonce or the VM's own public key in its report data. A stale or replayed quote fails this check: the nonce does not match a fresh challenge, or the replayer cannot prove possession of the bound key.

The compose hash is where the vendor's stack enters the hardware evidence. The compiled manifest pins every container image by content digest, and the manifest's own digest is what the event log records: change an image, a command, a default, or an egress declaration and the compose hash changes: a different measurement, and no keys (Section 6). Two things are deliberately *not* measured directly. The image bytes are named by digest inside the measured manifest and verified against that digest when pulled. Injected secrets arrive encrypted to the application's KMS-derived key, so they can rotate without a governance event, and they gate nothing.

Every VM serves its attestation evidence over a public verification endpoint, and verification is open tooling end to end: an Intel DCAP quote-verification library for the hardware chain, the runtime's open-source tooling for the event-log replay,[11] and a single ledger read (Section 13.3) to compare the recovered compose hash against the institution's allowlist.

5.3 The reviewer's three checks

Before the first institutional payload enters the VM, the institution's security team verifies it independently. Three checks require no trust in the vendor or Ember:

1. **Pull the attestation quote.** Every VM exposes one through its verification endpoint.
2. **Verify the quote's signature** against the silicon vendor's published root of trust. This proves genuine confidential hardware, not an emulator or a mock.
3. **Compare the measurements** in the quote to the reproducible platform build and to the application-manifest hash the institution's quorum approved on the ledger. Equal hashes prove the approved code is exactly what is running.

Appendix A walks through the procedure step by step. The receipt chain repeats the code identity for every subsequent session, so drift is visible immediately: what a reviewer verifies once, the architecture re-proves continuously.

6. Only approved software receives decryption keys

Attestation alone proves what is running. It does not stop an operator from booting different code next to the data. The enforcement mechanism is key management: every secret a VM holds derives from a key-management service (KMS) that releases keys only to VMs whose attestation it has verified against the institution's on-ledger allowlist. Unapproved code does not receive a policy exception. It receives ciphertext it cannot read.

6.1 The key-management service

The KMS is itself a measured workload inside attested hardware, not an operator console. A key release is an attestation check, not an authorization decision by an employee: the requesting VM presents its quote, the KMS verifies the hardware chain and looks up the measurement against the governance contract the institution controls on the ledger, and a match derives keys while a mismatch derives nothing. There is no path by which Ember, the vendor, or the datacenter requests a VM's keys, because the protocol has no requester role, only measurements.

Concretely, the platform uses the dstack KMS in its on-chain mode.[11][12] The governance object is a per-application contract on the ledger: a **DstackApp** contract whose state is the application's identity and an allowlist of approved compose hashes, and whose owner is the institution's governance account (Section 13). The KMS nodes are themselves attested workloads, governed by the same contract family, replicating a root key that never exists outside attested memory.

6.2 The release protocol

A key release runs in four steps, and each fails closed:

1. **The VM proves what it is.** The booted CVM produces a fresh TDX quote whose event log names its application identity and compose hash (Section 5.2), and whose report data binds the ephemeral public key of the requesting instance. The quote and the channel key are bound together: a quote replayed by anyone else authenticates a channel they do not hold, and yields nothing they can read.
2. **The KMS verifies the hardware.** Signature chain to the silicon root, TCB status, event-log replay. A quote from an emulator, an unpatched platform, or a tampered log stops here.
3. **The KMS consults the institution's contract.** One read: is this compose hash on this application's allowlist? Anyone can perform the same read, and Section 13.3 shows it against a live deployment.
4. **Match derives, mismatch starves.** On a match, the KMS derives the application's keys and returns them encrypted to the quoted channel key. On a mismatch, nothing is derived. An operator-pushed build absent from the allowlist boots, attests a different compose hash, reads `false`, and holds ciphertext it cannot open: no storage, no receipt signing, no TLS identity.

Rotation and revocation follow the same authority. The KMS root and its node set are governed on the ledger, and root rotation is a governance action with a controlled handover. Removing a compose hash from the allowlist, or

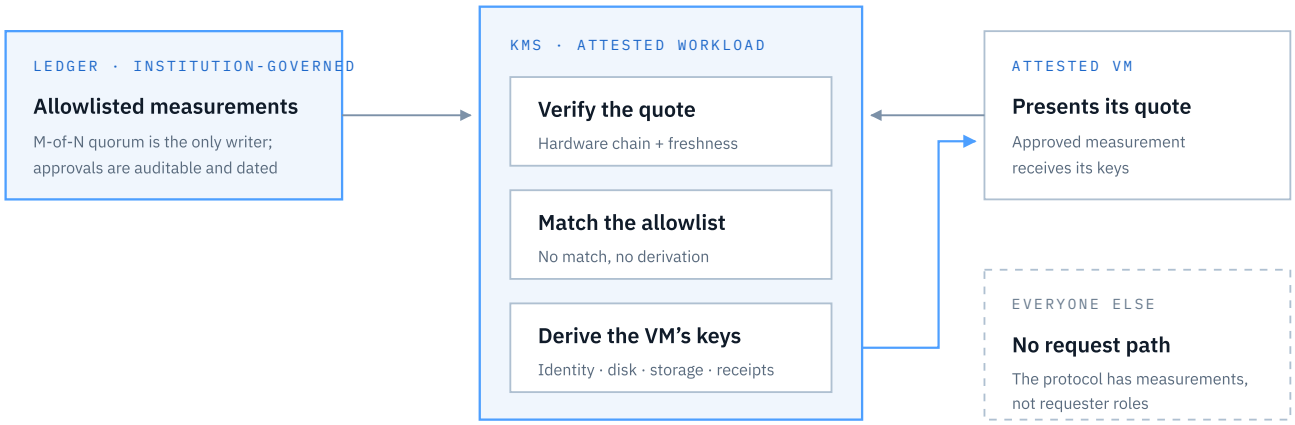
declining to add the next one, strands the code it names at step 3. The floor beneath all of this, who protects the KMS root itself, is stated as limit L3 rather than assumed away.

6.3 Derived keys and what they bind

Derived keys split into **app-stable** keys, derived from the application identity alone so they survive approved upgrades and re-provisioned instances, and **instance-bound** keys, which die with the specific VM. The split is deliberate: an approved upgrade keeps the institution's data and receipt chain, and a dead instance takes its local scratch state with it.

- **The application identity key** derives from the application identifier recorded in attestation and signs its receipts. A verifier walks the signature chain from any receipt back to the KMS root registered on the ledger, and checks the code hash independently, so a receipt from unapproved code cannot verify (Section 12).
- **The disk-encryption key** derives from the application identity *and* the specific VM instance. The local volume mounts transparently inside the VM and is ciphertext to everyone else. The instance binding is a disclosed caveat: a replacement VM cannot decrypt its predecessor's local volume, so local data has the lifetime of the instance (Section 11).
- **The storage-gateway key** derives from the application identity alone, portable across instances by construction. It encrypts everything the gateway writes to durable object storage before the bytes leave the VM (Section 11).
- **The ingress private key** is derived inside the VM and never leaves it. The TLS certificate binds the public half (Section 10).

FIG. 4 **Keys follow code. The KMS verifies a VM's hardware quote, matches its measurement against the institution's ledger allowlist, and derives keys on a match. Operators and vendors do not appear in the protocol. Unapproved code receives nothing but ciphertext.**



> This is the sentence the rest of the paper leans on: **the institution's allowlist, not any operator, decides which code the keys follow.** Sections 7 and 13 describe the governance that maintains the allowlist, and limit L3 names the custody floor.

7. The two locks: operator and vendor

Confidential computing is usually presented as protection against the cloud. The institutional problem needs a second lock, because in a conventional deployment it is the *vendor*, the party that wrote the application, who holds the most natural access. A sovereign VM is locked against both, by different mechanisms, and the difference is worth stating exactly.

7.1 Locked from the operator, by hardware

The operator's exclusion is physical. Memory encryption removes the hypervisor's read path, disk encryption under KMS-derived keys removes the storage path, and in-VM TLS termination removes the wire path. Attestation removes the substitution path, because swapped code produces a different measurement and receives no keys. What remains to the operator is enumerated in Section 4.1: scheduling, observation of encrypted traffic and resource metadata, snapshot and rollback of encrypted state, and stop. Ember operates VMs it cannot open, and the point of the architecture is that this claim is checked by the reviewer's procedure in Section 5.3, not taken from Ember.

7.2 Locked from the vendor, by governance

The vendor's exclusion is procedural, enforced by the key path. The vendor wrote the code, but a running VM offers it no shell, no debugger, no telemetry channel, and no update lever. The only way vendor code reaches an institution's VM is the onboarding path of Section 2: compile, measure, and pass the institution's M-of-N approval. A vendor (or anyone holding its release pipeline) can push whatever it likes at the platform. What it cannot do is cause that code to receive the VM's keys. An unapproved build boots into a machine that cannot decrypt the institution's storage, cannot sign valid receipts, and cannot serve the VM's TLS identity: a brick with a different hash.

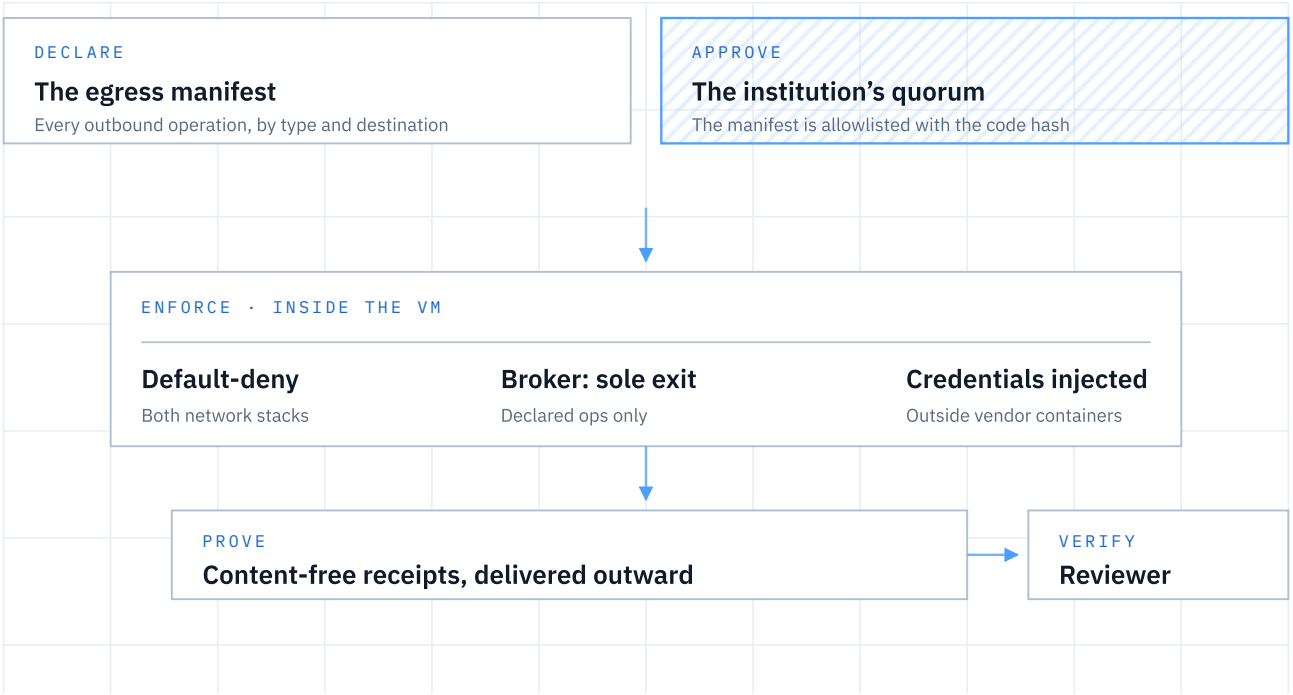
The composition of the two locks is the product. Hardware makes the operator irrelevant to custody, governance makes the vendor a proposer rather than an administrator, and the institution's recorded approval is the single authority both locks resolve to. Section 13 specifies the governance ceremony itself, including how an institution takes ownership and how upgrades happen without either lock weakening.

PARTY	CAN	CANNOT
OPERATOR / EMBER	Provision, schedule, meter, and stop VMs. Snapshot or roll back encrypted state. Observe traffic and resource metadata.	Read memory or disk, obtain keys, forge attestation or receipts, substitute code that retains access.
VENDOR	Propose new builds. Run its application. See its own product's metadata surfaces.	Open a shell, receive telemetry, read the data inside, or push code into a VM without the institution's recorded approval.
INSTITUTION	Approve code and egress manifests by quorum, verify attestation and receipts, stop use.	Reach inside the running VM any more than the others. Its power is governance and evidence, not a back door.

The only way out is a declared operation.

The vendor declares the application's egress, the institution approves it with the code, the broker enforces it as the sole exit, and receipts prove which declared operation ran and where it went. Everything undeclared is dead by default.

DECLARE · APPROVE · ENFORCE · PROVE



8. Default-deny containment

Sections 4–7 lock the box against parties outside it. Adversary A3 is already inside: the vendor's own application, subverted by a poisoned dependency or steered through its inputs, running with the institution's data in plaintext. Containment is the answer, and it is engineered as ordinary, testable network discipline rather than novel cryptography.

8.1 Two stacks, both closed

Egress is denied by default at both layers the workload touches: the container network inside the VM, and the VM's own firewall beneath it. The vendor's containers can reach each other and the platform's in-VM services (the broker, the storage gateway). They can reach nothing else. There is no direct route from a vendor container to the internet, to the host's network, or to a raw storage endpoint. The compiler of Section 2.3 already rejected the configurations that would create one.

8.2 The disciplines that keep "closed" closed

- **DNS is pinned.** Workload DNS resolves only through the platform's resolver, and only for declared destinations. Custom DNS configuration is rejected at compile time, so tunneling data through lookups of attacker-chosen names fails with the query.
- **TLS goes to pinned hosts.** The broker connects to the exact hosts the manifest names and refuses redirects that would carry an approved request to an unapproved recipient.
- **Credentials live outside the workload.** Provider credentials are injected by the broker at the VM boundary (Section 9.3), so a compromised workload has no key to steal and replay from elsewhere.
- **Denied traffic is counted.** Every refused connection increments content-free counters the institution can see. A workload that suddenly probes its walls is visible without anyone reading its data.

8.3 The claim, parameterized

What containment buys is a bounded blast radius, stated per application: **a compromised workload exfiltrates only through operations the institution approved, to recipients it approved, under receipt.** The bound is only as tight as the manifest is narrow, which is exactly why the manifest is an institutional approval object rather than vendor configuration, and why the standing limit that a prompt-injected workload can still misuse an *allowed* operation is stated in Section 15 rather than footnoted. The claim also has a floor: it holds while the attacker remains inside the vendor-container boundary. An escape into the measured platform itself (kernel, runtime, broker, gateway, KMS) is a TCB failure, stated as limit L10.

9. The egress broker and the manifest

The broker is the VM's single door. It runs inside the VM as a platform component (measured, attested, outside the vendor's containers) and is instantiated per application from the approved egress manifest. The platform ships a library of typed operations. The manifest selects and parameterizes them.

9.1 Provider-compatible surfaces

Inside the VM, the broker exposes the wire formats the ecosystem already standardized on: an OpenAI-compatible chat-completions endpoint, the Anthropic Messages API, and read-only HTTP connectors with pinned destinations. For most applications, adopting the broker is one environment variable:

```
# The application's provider config points at the in-VM broker.
export OPENAI_API_BASE=http://broker.internal/openai
export ANTHROPIC_BASE_URL=http://broker.internal/anthropic
# Provider keys are injected by the broker. The app's own key config stays empty.
```

The application's model-calling code does not change. This is the same compatibility that makes lift-and-shift real at the network layer: the VM meets the application at interfaces it already speaks.

9.2 The manifest is the policy

The egress manifest declares every outbound operation the application needs: which provider operations, which connectors, which destinations. The institution approves it alongside the code hash: the two are allowlisted together, and a manifest change is a governance event exactly like a code change. There is no separate policy engine to configure and drift: **the manifest is the policy.** An application that should not ship embeddings to a third party simply does not declare an embeddings operation, and no runtime state can re-open what was never declared. Appendix B shows a complete example.

9.3 Credential admission, then injection

Provider credentials, whether the institution's or the vendor's, are held by the platform's secret path and injected by the broker at dispatch, outside the vendor's containers. The workload never observes a provider key.

Two distinct proofs govern a credential's life, and the platform keeps them separate. The first is a **pre-dispatch admission gate**. Before a credential may be used at all, it is admitted against a versioned **provider-policy profile**: the provider and account class, the retention and training terms in force, and the evidence that the provider's console cannot retrieve content sent under it. That evidence is what it honestly is: a provider API or account-configuration snapshot where the provider exposes one, a contractual account class (a zero-data-retention agreement, for example) where it does not, and manual review of the terms. Its hash is recorded with the profile at admission. A credential whose provider account logs prompts for later reading fails admission, whoever owns it, and the gate fails closed: no admitted credential, no dispatch. Because it runs *before* dispatch, the gate is the control that keeps content from leaving under an unacceptable relationship. A receipt, produced after dispatch,

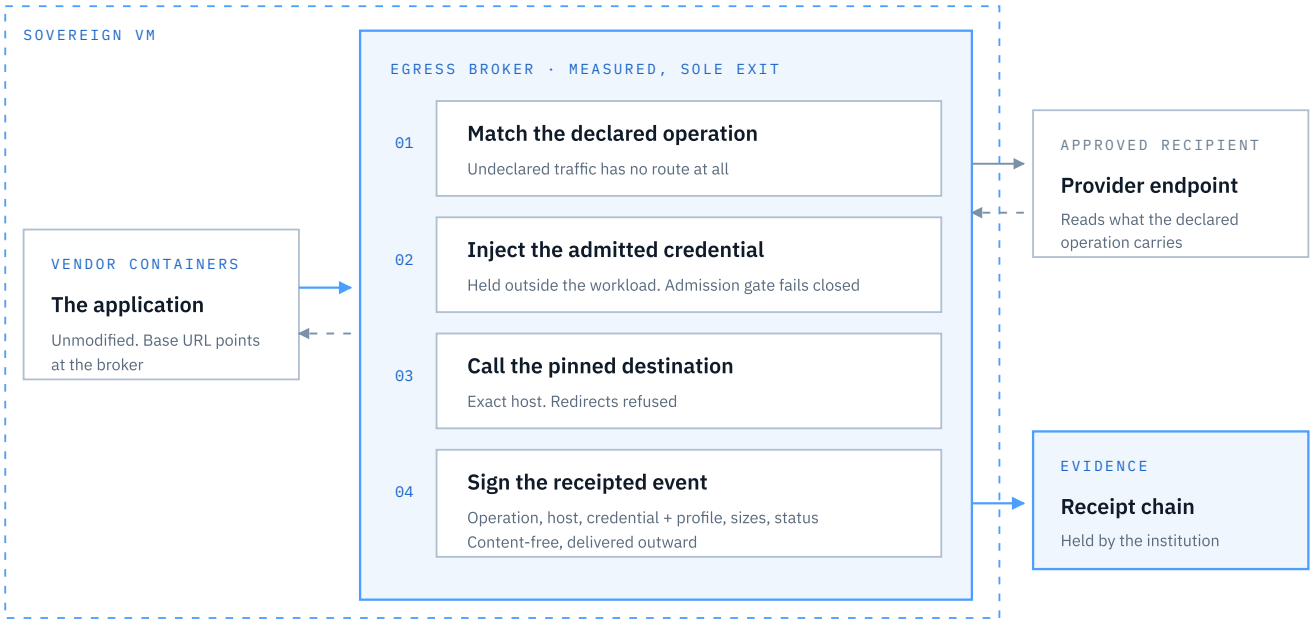
cannot be, because by the time it exists the provider has the payload. Profiles carry expiry, and are revalidated when the provider's configuration or terms change. An expired or invalidated profile closes the gate again.

The second proof is the **per-call receipt** (Section 12): every dispatched operation records the credential's fingerprint and the admitted policy-profile identifier, so a verifier ties each call to the exact relationship, terms, and evidence under which it was admitted. The boundary between the two proofs is stated rather than blurred: the receipt proves which admitted relationship a call used. It does not independently prove what the provider retained after the call, or the provider's continuing compliance (Section 15, L6).

9.4 What leaves, and what is recorded

A dispatched operation produces a content-free receipted event: the operation type, the destination host, the credential fingerprint and its policy-profile identifier, sizes, timing, and status. It never includes the payload (Section 12). The receipt proves which declared operation ran and where it went. It does not claim the recipient cannot read what an approved operation, by design, sent it. An inference provider reads the prompts a declared inference operation carries, under the credential's terms. The platform makes that boundary provable and keeps it in Section 15's limits rather than implying it away.

FIG. 5 **The broker's dispatch path. The application's outbound call must match a declared operation. The broker injects an admitted credential the workload never sees, calls the exact pinned host, and signs a content-free receipted event before the response returns.**



10. Ingress: TLS ends inside the box

Users reach the vendor's product in the VM directly. The TLS session that begins in the user's browser or client terminates inside attested hardware: the certificate's private key is derived inside the TEE boundary from KMS-released secrets and never exists outside encrypted memory. The operator relays ciphertext it cannot terminate. No load balancer, CDN, or relay holds a key in the plaintext path.

10.1 The path, concretely

The shipping profile's zero-trust TLS,[11][12] serves a VM in one of two shapes. On a **platform-managed subdomain**, the default, the VM is reachable at a name derived from its application identity under the platform's base domain. TLS for that name terminates in the runtime's gateway: a reverse proxy that is itself a measured, attested CVM governed like any other workload, holding the certificate key only inside attested memory, and forwarding traffic to the application VM over an attested, encrypted channel between the two TEEs. On a **vendor custom domain**, the vendor's DNS points the product's own name at the deployment, and the certificate key is derived and held inside the application VM itself. In both shapes the claim is the same: no private key for a VM's name ever exists outside attested hardware.

- **Attested issuance.** Certificates are obtained through the standard ACME protocol,[6] with the ACME account key held inside the TEE, so issuance under that account corresponds to an attested workload rather than an operator request.
- **CAA-narrowed.** DNS CAA records[7] restrict which authority may issue for the VM's names and, where the CA supports account binding, restrict issuance to the TEE-held ACME account specifically. Both narrow who could obtain a competing certificate.
- **CT-monitored.** Certificate Transparency logs[8] make every issuance for the VM's names public. CT prevents nothing, but it guarantees a mis-issuance cannot happen silently, and monitoring alarms on any certificate for a governed name that does not correspond to an attested key.

10.2 The trusted domain administrator

This chain has an honest v1 boundary, stated here rather than implied away: **whoever administers the DNS zone is trusted.** For a platform-managed subdomain that is the infrastructure operator. For a custom domain it is the vendor and its registrar. A zone administrator can, by construction of the DNS, repoint the name or replace the CAA record and obtain a competing certificate from a public CA. Against that party the architecture provides detection, not prevention: the competing issuance is public in CT and trips the monitor, and the impostor endpoint cannot present the attested key, so it fails independent verification even as it passes a browser's.

That last distinction is load-bearing. A browser automatically verifies only that the name resolved to a holder of a CA-issued certificate. The property this section claims, that the served key lives inside the governed, attested deployment, is checked on the independent verification surface: the attestation evidence includes a second chain for the served key, rooted in the application's own KMS-derived certificate authority, so a reviewer checks the served certificate against that chain and then verifies the quote behind it (Section 5.2 and Appendix A, step 5). Users relying on the browser check alone inherit the v1 assumption. Limit L11 records it.

11. Storage that works like a computer

Applications assume a disk. The platform's storage design preserves that assumption (the application writes files and objects as on any machine) while guaranteeing nothing readable ever rests outside the VM. Two tiers, both generic, both invisible to the application.

11.1 Tier 1: the encrypted local volume

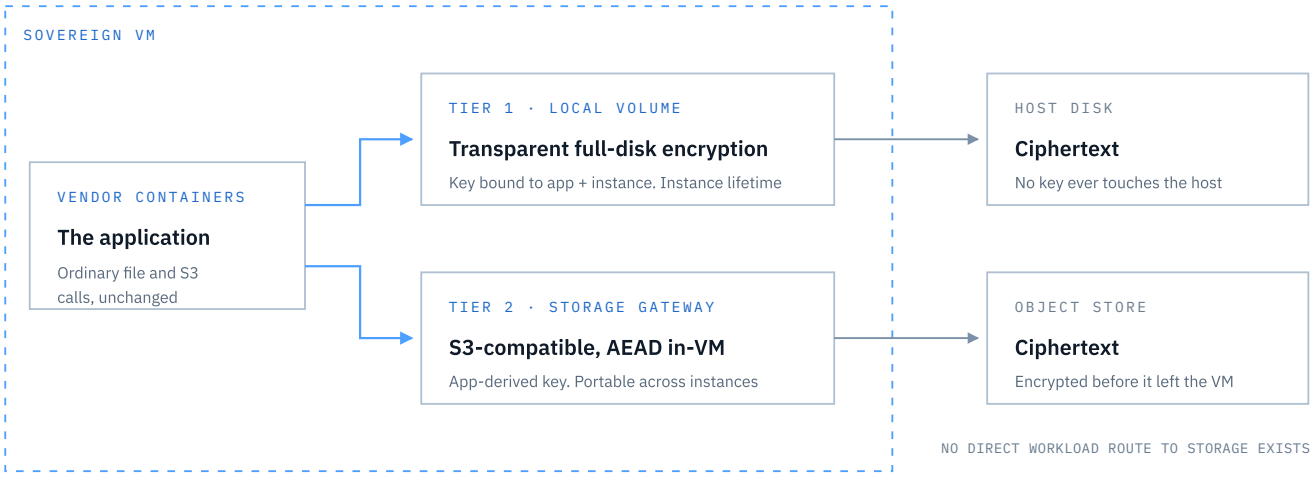
The VM's filesystem sits on full-disk encryption with a key derived from the application identity and the VM instance (Section 6.3). Writes are transparent to the application, and the host holds ciphertext. The instance binding is the disclosed caveat (L2): a replacement VM cannot decrypt its predecessor's volume. Scratch space, caches, and rebuildable indexes belong here. Anything the application cannot afford to lose belongs in Tier 2.

11.2 Tier 2: the encrypted object-storage gateway

For durable, portable data, a platform sidecar exposes an S3-compatible endpoint inside the VM. The gateway transparently applies authenticated encryption with keys derived from the application identity, portable across instances by construction, before anything leaves the VM, then writes ciphertext to ordinary object storage. The application's existing S3 client simply points at the in-VM endpoint.

One design constraint is load-bearing: the vendor workload reaches storage only through this gateway, never via its own network route. A raw network disk or direct bucket credential in the workload would be a generic exfiltration channel and would void Section 8's containment claim. Routed through the gateway, the property is unconditional: whatever a compromised workload writes out is ciphertext outside the VM.

FIG. 6 **The two storage tiers. The application uses an ordinary filesystem and an ordinary S3 endpoint. Encryption is applied transparently inside the VM, and everything at rest outside it, host disk and object store alike, is ciphertext.**



> What the platform does not claim for storage: rollback protection. A host can restore a previously valid encrypted state. This is an integrity and freshness limit, never a confidentiality one (L1).

12. Receipts

A receipt is the platform's unit of evidence: a signed, content-free statement, produced inside the VM, of what a session or an egress operation actually did. Receipts carry hashes and metadata (code identity, operation types, destination hosts, credential fingerprints and policy-profile identifiers, sizes, timing, status), never content.

12.1 Signing and verification

Receipts are signed with the application identity key of Section 6.3, over a canonical JSON serialization,[9] so independent implementations hash identical bytes. Verification walks the signature chain from the receipt's signer through the key-management service to the root registered on the ledger, and independently checks that the code hash the receipt names is one the institution's quorum approved. A receipt signed by unapproved code fails on the second check even if its chain is intact. A fabricated receipt fails on the first. The verifier is open source, and running it requires no account with the vendor or Ember.

12.2 Outward delivery is the freshness anchor

Receipts are delivered outward at session time, to the institution and, where the product surfaces it, to the client. They are never only stored on the VM's host. This is a deliberate structural choice: the institution's copy lives outside the host's rollback domain, so a host that restores its own disk to an earlier state cannot un-happen a session whose receipt the institution already holds. Within a session, receipts form a chain, each naming its predecessor, so removal or reordering is visible. Outward delivery is the platform's one rollback mitigation.

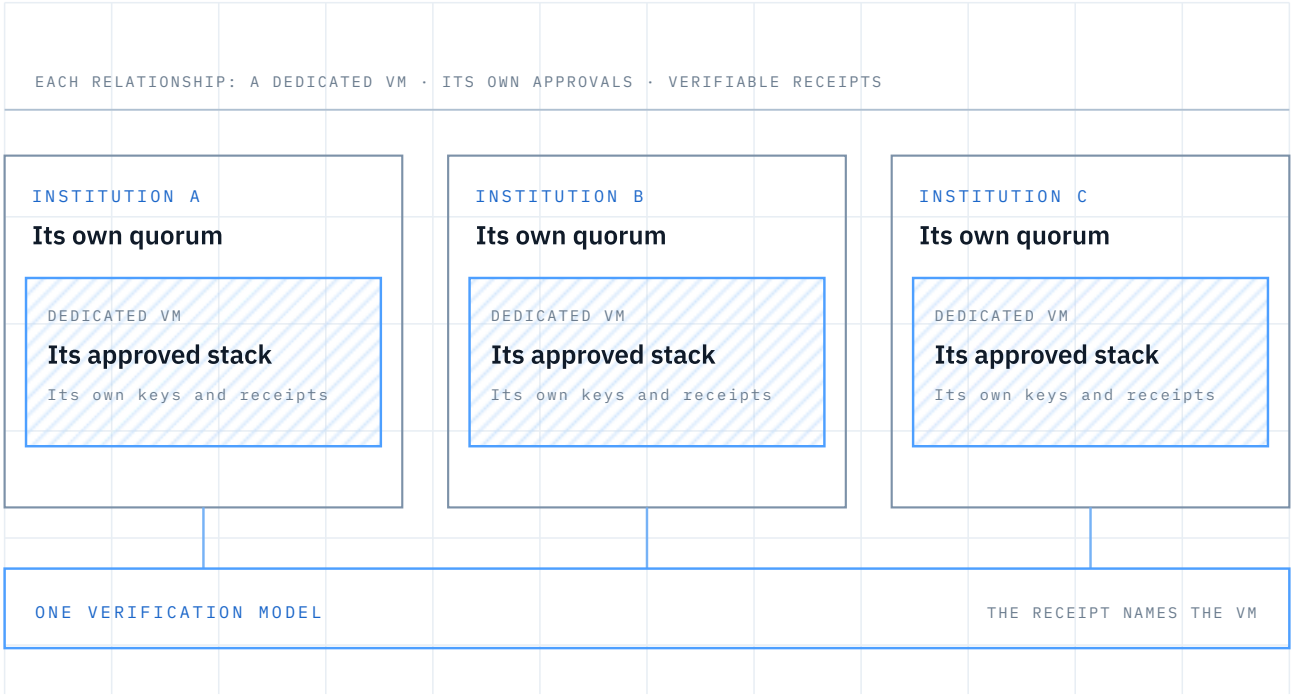
12.3 What a receipt proves, and what it does not

A receipt proves that approved code, in an attested VM, executed a declared operation to an approved destination under a named credential, at a time the institution's copy fixes. It does not prove the application's outputs were correct, that an approved recipient honored its contractual terms, or that no approved operation was misused by a subverted workload (Section 15). The discipline throughout the platform is that receipts certify exactly what their session used: sales language does not exceed the receipt.

Each institution governs its own protected deployment.

A dedicated virtual machine is provisioned for each vendor–institution relationship. The same verified controls apply everywhere, while governance and evidence belong to each institution alone.

ONE VERIFIED PLATFORM · DEDICATED VMS



13. Governance and upgrades

The most dangerous operation in any attested system is changing the code, because a malicious upgrade (adversary A4) inherits every key and every promise made about the old build. The platform therefore treats upgrades as governed, auditable events rather than deployments, and puts the governing quorum in the institution's hands.

13.1 The institution's root: a passkey-controlled account

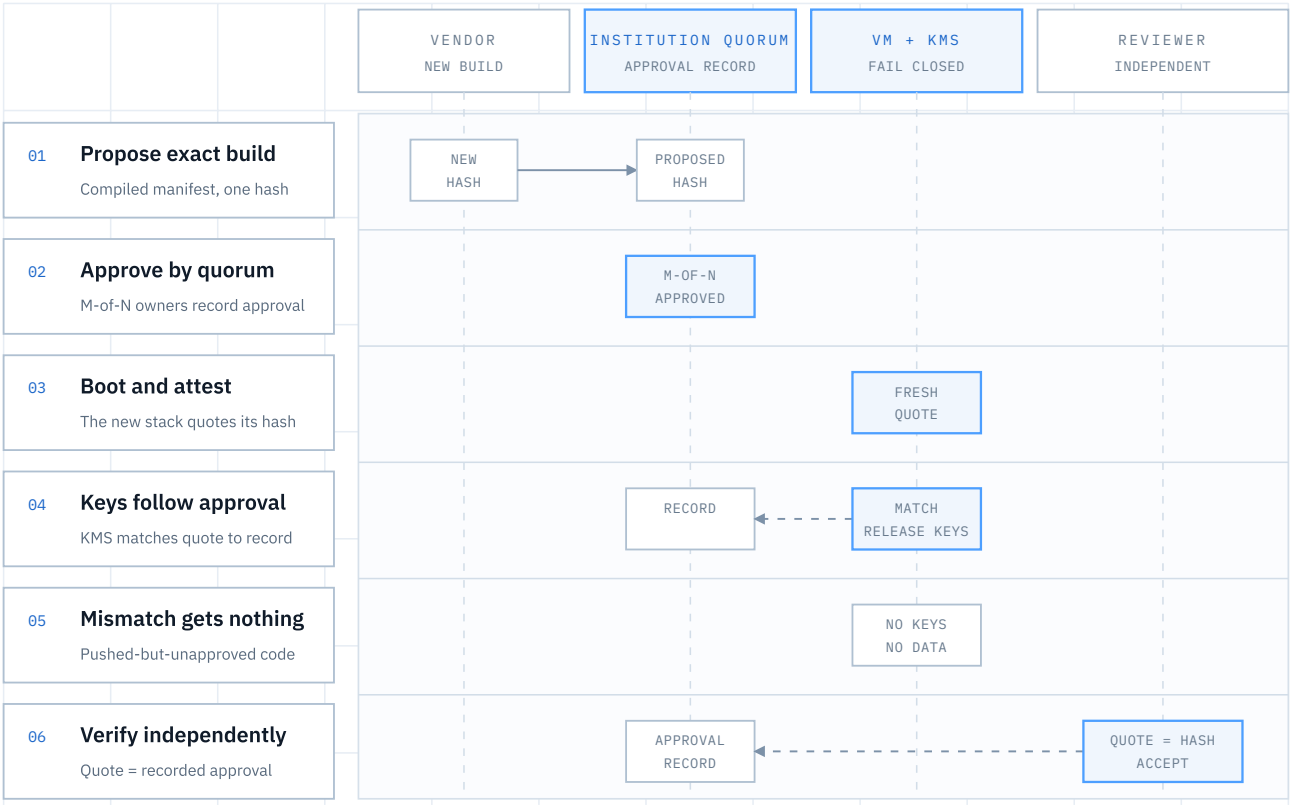
Each VM's governance contract on the ledger is owned by an institutional account created through a WebAuthn passkey ceremony,[10] the same public-key credential model institutions already deploy through platform authenticators and security keys. No seed phrases, wallet software, or tokens are involved. The ledger is used as an append-only, operator-proof record, nothing more. A solo team's 1-of-1 account is a complete organization, and a larger institution expands the owner set to M-of-N passkeys and security keys, with a quorum required for every governing action. The account's allowlist is what the KMS of Section 6 consults: governance *is* the key path.

Concretely, the governed object is the application's governance contract (Section 6.1), and the governing account is an on-chain smart account[13] whose signers are the institution's passkeys. The contract's one governing verb is extending the compose-hash allowlist, and every use of it is a signed operation from the owner account, timestamped on the ledger.

13.2 The upgrade ceremony

A new build enters governance by its measured-manifest hash. The vendor proposes, the institution's quorum approves on the ledger, and only then will the KMS release the VM's keys to the new measurement. Delivery of bytes to a machine is transport, and transport is never authorization: whoever pushes code, the allowlist decides whether that code can decrypt anything. After an approved upgrade, the VM's fresh attestation quotes the new hash, and any reviewer confirms the quote matches the recorded approval.

FIG. 7 The upgrade ceremony. New code receives an institution's keys only through its quorum's recorded approval. Pushing bytes is transport, never authorization, and a reviewer checks every fresh attestation against the approval record.



13.3 The ceremony, live

This transcript is not a mock: it is the third code revision of a live deployment.

```
# The application's governance contract on the ledger (Base); address shortened.
# These reads run against any RPC endpoint. No account with anyone is required.
APP=0x4697…428e
# 1 · Who governs? The owner is the institution's smart account, not the operator.
$ cast call $APP "owner()(address)"
0xb278…DB90

# 2 · The proposed build's compose hash, before approval: not allowlisted, no keys.
$ cast call $APP "allowedComposeHashes(bytes32)(bool)" \
    0x6a1aa4ef3b4772f201b4e8585de86b571e7e5a101d67caa8603101d03a24bb19
false

# 3 · The owner approves: addComposeHash(0x6a1a…bb19), signed by the account's quorum.
#     tx 0x0b78…d0f2
# 4 · The same read, after. The upgraded VM attests this hash and receives its keys.
$ cast call $APP "allowedComposeHashes(bytes32)(bool)" \
    0x6a1aa4ef3b4772f201b4e8585de86b571e7e5a101d67caa8603101d03a24bb19
true
```

The failure case is the same read. A build pushed by anyone (the vendor's pipeline, the operator, an attacker who has compromised either) attests its own compose hash. If the allowlist read returns `false`, the KMS derives nothing (Section 6.2), and the prior application's keys stay with the prior application. Approval is not a deployment step that can be skipped under pressure. It is the only path to keys. This deployment's owner account runs at the 1-of-1 bootstrap floor of Section 13.5. Expanding it to an institutional M-of-N changes the signers and nothing else in the transcript.

13.4 Quarantine is stop-only

Governance deliberately lacks levers that would weaken the locks. There is no "support access" role, no plaintext export, and no emergency code path that bypasses the quorum. The strongest incident response any party holds against a running VM is to stop it: the institution by governance, the operator by its stop power. A gracefully requested stop is receipted before shutdown. An abrupt halt cannot be signed by the VM after the fact, and is evidenced instead by the break in the institution's receipt stream and the endpoint going unreachable (L4). A quarantined VM keeps its secrets. The response to suspicion is halting execution, never opening the box.

13.5 Bootstrap and the transfer of governance

A vendor may provision a VM before its institution takes governance: a pilot, a proof, a sales motion. In that bootstrap state the vendor's own account holds the allowlist and the honest claim is smaller: the vendor administers these controls, provably, and the operator still cannot read the VM. The institutional-authority claim of Section 1.3 begins when the institution's owner set takes the quorum, a recorded governance transfer on the same account, not a migration. The VM's state is always disclosed in its receipts and verification surface, so the two states cannot be confused.

14. Deployment models compared

An institution choosing an AI deployment is choosing an operating model and a trust model at once. Vendor SaaS centralizes both in the vendor. On-premises and BYOC deployments move operations into the institution's perimeter but leave broad trust in application code, and the data remains readable inside that perimeter. The sovereign VM is a third structure: the workload remains operated as a service, while custody, code identity, and evidence are enforced by hardware and verified independently.[1]

DIMENSION	VENDOR SAAS	ON-PREM / BYOC	SOVEREIGN VM
VENDOR'S PRODUCT	**As shipped.**	**Lagged.** Internal deployment trails the release train.	**Institution-approved release.** Exact software versions and configuration approved by the institution.
WHO OPERATES	**The vendor.**	**The institution.** Its platform team owns tenancy, upgrades, capacity.	**Infrastructure operators.** They run machines but cannot authorize software or read plaintext.
WHO CAN READ THE DATA	**The vendor** and its infrastructure, structurally.	**The institution's perimeter** and the vendor's code inside it.	**Approved code and recipients.** Host and operator personnel hold ciphertext.
CODE GOVERNANCE	**Vendor release pipeline.**	**Vendor release pipeline,** re-audited by the buyer per release.	**Institution's quorum.** Keys follow only allowlisted measurements.
DATA LEAVING	**Contractual.** Egress is the vendor's configuration.	**Perimeter firewall,** maintained by the buyer against the whole stack.	**Declared operations only,** default-deny, receipted per event.
AUDIT ARTIFACT	**Operator logs** and attestations of process.	**The buyer's own logs.**	**Hardware attestation + receipts,** verifiable by third parties.
INSTITUTIONAL BURDEN	**Low.**	**High.** A production platform per vendor.	**Low operational, active governance.** Approve code and manifests, verify evidence.

14.1 The inference boundary

One boundary is independent of deployment model: the model call itself. A prompt routed to an external inference provider is plaintext to that provider, under the terms of the credential that authenticated it, admitted before any dispatch against a versioned provider-policy profile (Section 9.3), with every use receipted under the profile's name. The platform makes the routing, credential, and admitted terms provable. It does not make an external provider blind, and the receipt proves which admitted relationship was used, not the provider's continuing compliance.

15. Honest limits

These limits are stated up front, in receipts, in sales language, and here, rather than discovered in review. Each names what is not claimed and where the boundary sits.

L1	**No storage rollback protection.** A host can restore an earlier valid encrypted state on either tier. This breaks freshness and guaranteed deletion, not confidentiality. Outward-delivered receipts remain outside that rollback domain; applications needing freshness must anchor critical state elsewhere.
L2	**Tier 1 storage has the lifetime of its instance.** A local volume's key is bound to one VM; a replacement cannot decrypt it. Durable data belongs behind Tier 2. Copied disk images are useless elsewhere.
L3	**Key management and hardware remain roots of trust.** VM keys depend on the attested KMS root, the silicon vendor's attestation root, and the institution's on-chain allowlist. The platform does not claim that every hardware or infrastructure compromise chain is eliminated.
L4	**Stop power is real.** Ember and underlying infrastructure can halt a VM. A graceful stop is receipted; an abrupt one breaks the receipt stream. Availability depends on service commitments and Tier 2 portability, not cryptographic custody.
L5	**An allowed operation can be misused.** Containment restricts compromised software to declared operations; it does not make them safe. A prompt-injected application can send data through an approved call. Narrow manifests and receipt review reduce, but do not remove, this risk.
L6	**Approved recipients read what they receive.** A receipt proves the destination, credential, and admitted policy profile, not what an outside recipient retains or whether it continues to honor those terms.
L7	**Side channels and hardware flaws remain possible.** Confidential computing has a continuing history of timing, cache, speculative-execution, and firmware issues. Attestation lets vulnerable versions be rejected as fixes arrive; silicon risk never reaches zero.
L8	**The host observes metadata.** The host sees timing, sizes, permitted destinations, resource use, and denied-traffic counts. This reveals that work occurred and roughly how much.
L9	**Attestation proves identity, not correctness.** A quote proves which code ran, not that it is correct. Source review, endpoint security, institutional approval, and audit quality remain trusted.
L10	**The measured platform is trusted code.** The kernel, container runtime, broker, storage gateway, and KMS are trusted. If compromised, containment and receipt claims can fail. Open source, reproducible builds, and measured pinning make this base auditable, not infallible.
L11	**The domain administrator is trusted in v1.** A DNS-zone administrator can repoint a name or obtain a competing public certificate. Transparency logs and independent verification expose the impostor but do not prevent it. Ordinary browser checks inherit this trust.

16. Conclusion

The sovereign VM replaces trust in vendor and operator access controls with evidence about code and custody. The vendor maintains one product and proposes releases. Infrastructure operators run the deployment but cannot authorize it or read its plaintext. The institution decides which releases receive its keys.

Hardware confines plaintext; on-chain governance binds keys to approved software; default-deny network controls limit where data can go; receipts record what happened. These controls do not eliminate hardware roots, rollback, metadata, approved-recipient access, or application risk. They make those boundaries explicit and independently reviewable. That is the contribution: a managed deployment with a custody claim that can be tested.

Appendix A: Verifying a VM, step by step

The procedure a security reviewer runs before approving the first payload, expanded from Section 5.3. Nothing in it requires an account with the vendor or Ember.

> **Implementation status.** The live substrate supports the attestation, ledger, and key-release checks below; checks involving the manifest compiler, platform receipts, and their verifier become executable as those components ship.

1. **Fetch the quote.** The VM's verification endpoint returns the attestation document: hardware evidence, the measurement chain (firmware, runtime OS image, application-manifest hash), relevant platform state, and a freshness value. Section 5.2 names the fields on the shipping profile: the TDX quote and TCB status, the runtime measurement registers, and the application event log carrying the compose hash.
2. **Verify hardware authenticity.** Validate the quote's certificate chain against the silicon vendor's published root of trust, exactly as for any confidential-computing deployment. A passing check proves a genuine confidential VM with an acceptable TCB status produced this quote. An emulator or modified host cannot.
3. **Verify the platform stack.** Rebuild (or obtain a reproducible build of) the open-source runtime and platform components[11] and confirm the quoted OS and component measurements match. This step replaces trust in the operator with trust in an audit of public source.
4. **Verify the application.** Compare the compose hash recovered from the event log to the compiled manifest the institution's quorum approved. Then run the allowlist read shown in Section 13.3 to confirm the approval exists and predates the boot. Equal hashes prove the VM is running exactly the reviewed stack, with exactly the reviewed egress manifest.
5. **Verify the ingress identity.** Confirm the TLS certificate presented to users matches the attested certificate chain the verification endpoint returns (Section 10.2), that CAA restricts issuance, and that CT logs show no competing issuance for the VM's names.
6. **Verify receipts continuously.** Run the open-source verifier over the receipt stream: signature chain to the ledger-registered root, code hash against the allowlist, chain continuity, and per-event fields. Steps 1–5 are performed per build. This step is continuous and cheap, and its output attaches directly to a security-review record.

Appendix B: An onboarding example

A representative internal-assistant application: a web front end, an API server, a worker, and a vector database, described in ordinary Compose form.[5] The vendor's submission is two files.

```
# compose.yaml: the stack the vendor already runs, images pinned by digest
services:
  web:
    image: vendor/assistant-web@sha256:9f2c…e1a7
    depends_on: [api]
  api:
    image: vendor/assistant-api@sha256:41d8…03bc
    environment:
      OPENAI_API_BASE: http://broker.internal/openai   # index-card change 1
      S3_ENDPOINT: http://storage.internal              # index-card change 3
      TELEMETRY_DISABLED: "true"                        # index-card change 4
  worker:
    image: vendor/assistant-worker@sha256:77aa…9d02
  vectordb:
    image: qdrant/qdrant@sha256:c3f1…88e4
    volumes: [qdrant-data:/qdrant/storage]              # lands on the encrypted volume
volumes:
  qdrant-data:
```

```
# egress.yaml, index-card change 2: every way out, declared (the manifest IS the policy)
operations:
  - type: openai.chat-completions
    host: api.openai.com
    credential: institution/openai-zdr        # admitted against provider-policy profile openai-
  - type: anthropic.messages
    host: api.anthropic.com
    credential: institution/anthropic-zdr     # admitted against provider-policy profile anthrop
  - type: http.read-only
    host: ticker.internal-feeds.example.com   # pinned read-only connector
# no embeddings operation is declared, so embeddings cannot leave this VM
```

The compiler measures the pair into one production manifest and rejects anything the containment model forbids (Section 2.3). The institution's quorum approves the single resulting hash. From then on, the attestation any reviewer pulls from the running VM quotes that hash verbatim. Note what is absent: no attestation code, no key handling, no receipt logic, no network policy in the vendor's files. The platform supplies all of it, identically for every tenant.

Appendix C: Glossary

Application identity key. The KMS-derived key, bound to the application identifier recorded in attestation, that signs a VM's receipts. Verifiers walk its chain to the ledger-registered root and check the code hash independently.

Attestation. A hardware-signed statement of exactly what code and configuration is running inside a confidential VM, verifiable against the silicon vendor's published root of trust.

Broker. The measured in-VM component that is the workload's sole network exit. It executes only declared operations, injects credentials admitted through the pre-dispatch gate, and receipts every event.

Compose hash. The runtime's name for the measured manifest's digest: the value recorded in the attestation event log, allowlisted by governance, and consulted by the KMS before any key release.

CVM. Confidential virtual machine: a full VM whose memory is encrypted by the CPU against the host, with remote attestation. The form of TEE the platform uses, chosen so existing software runs unmodified.

Default-deny. The VM's network posture on both the container and VM stacks: traffic matching no declared operation has no route.

dstack. The open-source confidential-container runtime the platform builds on: a measured OS image for Intel TDX CVMs, a KMS whose key release is gated by on-ledger governance, and a zero-trust TLS gateway.

DstackApp contract. The per-application governance contract on the ledger. Its state is the application identity and the compose-hash allowlist, and its owner is the institution's governance account.

Egress manifest. The vendor-declared, institution-approved list of every outbound operation the application may perform: the policy object itself, allowlisted together with the code hash.

Institution. The governed organization whose data, policy, and audit obligations are at issue: an enterprise, a law firm, an agency. Often the vendor's buyer.

KMS. The platform's key-management service, the dstack KMS in its on-chain mode: attested workloads that derive VM keys only for measurements on the institution's ledger allowlist. It has measurements, not requester roles.

Ledger. The append-only record carrying code approvals, governance accounts, and key roots: Base, a public Ethereum layer 2, in the current deployment. Anyone can read it. Each of those is a hash, a public key, or a pseudonymous address — the ledger never carries content, receipt bodies, or the institution's name. Pseudonymous, not anonymous: anyone who learns the address can follow it. There is no token and nothing to trade.

Manifest compiler. The platform component that transforms a vendor's compose file and egress manifest into the measured production manifest, rejecting privileged mode, host mounts, custom DNS, mutable tags, and embedded telemetry.

Measured manifest. The compiled, content-hashed description of the full application stack, and the unit that approvals, attestation, and receipts name. Its digest is the compose hash.

Operator. Whoever runs the infrastructure beneath the VMs: Ember and the datacenter under it. Excluded from custody by hardware. Retains stop power.

Quorum. The M-of-N owner set of the institution's passkey-controlled governance account, and the only authority that can allowlist code for a VM.

Receipt. A signed, chained, content-free record of a session or egress event, delivered outward to the institution at session time.

Sovereign VM. A dedicated attested CVM, provisioned per vendor–institution pair, running the approved stack under the institution's governance. "The VM" unqualified means this.

TEE. Trusted execution environment: hardware-isolated, memory-encrypted execution with remote attestation.

Tier 1 / Tier 2 storage. The transparently encrypted local volume (instance lifetime) and the S3-compatible encrypting gateway (durable, portable), respectively.

Vendor. The organization that builds the AI application brought onto the platform.

Appendix D: Selected references

[1] Palantir Technologies, *Institutional Sovereignty in the Age of AI*, Section IX, "Verify compute you don't own," 2026. Source paper.

[2] Intel Corporation, *Intel Trust Domain Extensions (Intel TDX)*, architecture documentation. Intel documentation.

[3] IETF RFC 9334, *Remote ATtestation procedureS (RATS) Architecture*. RFC Editor.

[4] Confidential Computing Consortium, *A Technical Analysis of Confidential Computing*, v1.3, 2022. Source paper.

[5] Docker Inc. et al., *The Compose Specification*. compose-spec.io.

[6] IETF RFC 8555, *Automatic Certificate Management Environment (ACME)*. RFC Editor.

[7] IETF RFC 8659, *DNS Certification Authority Authorization (CAA) Resource Record*. RFC Editor.

[8] IETF RFC 6962, *Certificate Transparency*. RFC Editor.

[9] IETF RFC 8785, *JSON Canonicalization Scheme (JCS)*. RFC Editor.

[10] World Wide Web Consortium, *Web Authentication: An API for Accessing Public Key Credentials*, Level 3. W3C specification.

[11] Dstack-TEE, *dstack*: source repository, documentation, verification tooling, and reproducible OS-image builds. github.com/Dstack-TEE/dstack.

[12] S. Zhou et al., *Dstack: A Zero Trust Framework for Confidential Containers*, arXiv:2509.11555, 2025. arXiv.

[13] ERC-4337, *Account Abstraction Using Alt Mempool*. eips.ethereum.org.