Architecture

Source: docs/architecture.md · last edited 2026-07-13

The core idea

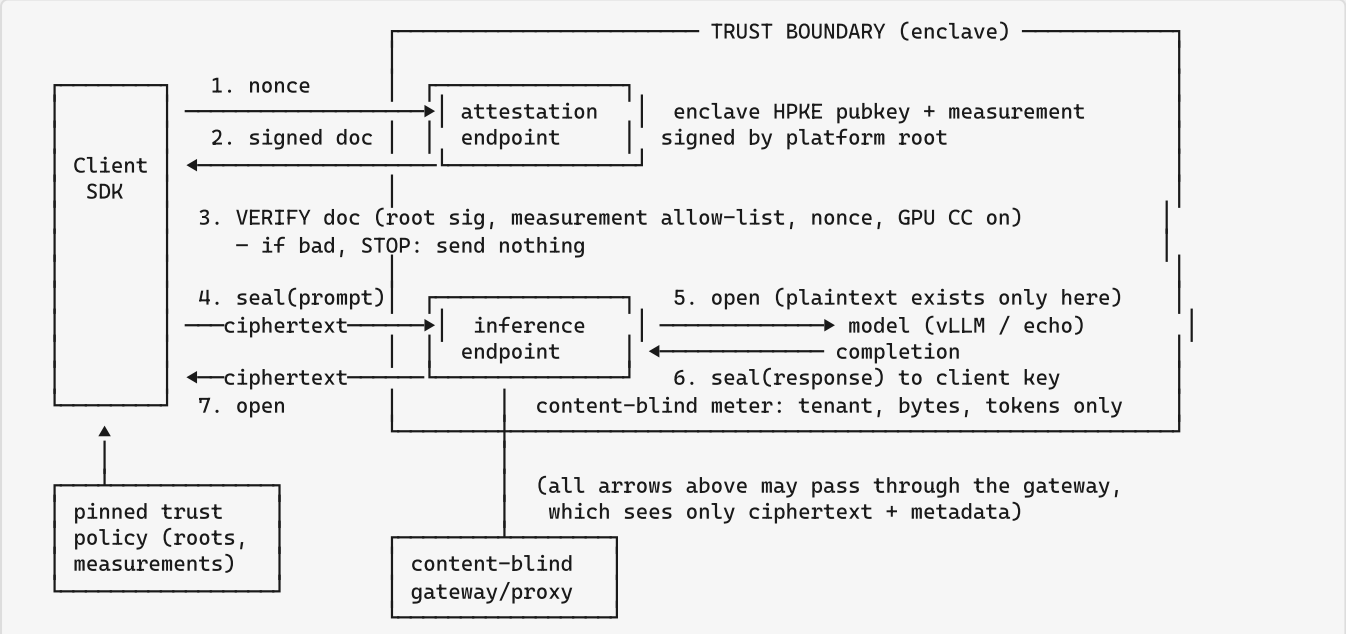
To run inference the model must operate on real tokens — you cannot compute on ciphertext at LLM scale (FHE/MPC are orders of magnitude too slow today). So "end-to-end encryption" here does **not** mean the data is never decrypted. It means:

> Plaintext exists **only ephemerally, inside a hardware-protected enclave** that even the operator cannot read into — and the client **cryptographically verifies** that before sending anything.

Three states, all protected:

- **In transit** — TLS 1.3, but terminating *inside* the enclave (attested TLS / RA-TLS) so the gateway only ever forwards ciphertext. (Phase 0 uses plain HTTP + the application-layer sealed box below; RA-TLS lands in Phase 1.)
- **At rest** — anything persisted (KV cache, logs, RAG stores, fine-tunes) encrypted with per-tenant keys, ideally customer-held (BYOK) so we cannot decrypt it.
- **In use** — the enclave: GPU confidential-computing mode encrypts HBM and the CPU↔GPU PCIe link, and the CPU runs in a confidential VM (SEV-SNP / TDX). Plaintext tokens and activations live only in encrypted memory the host cannot inspect.

Request flow



1. **Nonce** — client mints a fresh random value for freshness / anti-replay.
2. **Attestation** — enclave returns a signed document containing its fresh HPKE public key, the code measurement, GPU CC state, and the nonce.

3. **Verify** — client checks the signature chains to a **pinned** root, the measurement is on its **allow-list**, the nonce matches, and GPU CC is on. If any check fails it sends nothing. *This is the whole ballgame — it's what makes the guarantee checkable.*
4. **Seal** — client HPKE-seals the prompt to the attested enclave key. The associated data binds the ciphertext to the verified measurement, so it can't be replayed against a different or tampered enclave.
5. **Open** — only the enclave (holder of the private key) can decrypt. Plaintext exists solely here, in enclave memory.
6. **Respond** — enclave seals the completion back to a fresh client key.
7. **Open** — only the client can read the response.

Throughout, the **gateway** (and any operator infra) sees only ciphertext and metadata.

Component map

Component	Phase 0 (this repo)	Production (Phase 1–3)
Transport security	plain HTTP + app-layer sealed box	RA-TLS terminating inside the enclave
E2E payload crypto	compact HPKE-style `crypto.py`	RFC 9180 HPKE (`pyhpke`)
Attestation	real `tdx` + `sev-snp` verifiers, driven by byte-exact simulated quotes (`LIQUIFY_HWSIM=1`); `mock` / `simulated_tee` for quick dev	same verifiers, fed by real silicon + live Intel PCS / AMD KDS collateral
Measurement	sha256 of enclave source (mapped to an MRTD/launch-measurement stand-in per vendor)	MRTD (Intel) and launch measurement (AMD) from a reproducible image build — two values, one release
Model backend	echo stub / OpenAI-compatible upstream	vLLM on H200/B200 in CC mode, in-CVM localhost
Trust distribution	**signed release manifest**, verified against a pinned release key	same manifest published to a transparency log; pubkey baked into the SDK
Metering	stdout, metadata only	content-blind metering pipeline + billing

Attestation backends (Phase 1)

Attestation is pluggable behind two interfaces so real silicon drops in without touching the client SDK or wire format:

- `Producer` runs inside the enclave and emits *evidence*. Backends: `mock` (single Ed25519 key, dev only), `simulated_tee` (real X.509 chain, default for dev), `tdx` (`attest_tdx.py`) and `sev-snp` (`attest_snp.py`) — the real vendor formats.
- `Verifier` runs in the client and turns evidence into normalized `RawAttested` claims via pure cryptography. The client's single `attestation.verify()` call then applies the `TrustPolicy`.

Two verification stages, deliberately separated (mirroring real attestation stacks):

1. **Crypto** (backend `Verifier`): signature / certificate-chain validation up to a *pinned* root, plus `report_data` binding.
2. **Policy** (`attestation.verify`): measurement on the allow-list, expected TEE type, TCB floor (**downgrade rejection**, scalar *and* componentwise), vendor safety bits (debug, migration agent), GPU-CC required, freshness.

Two bindings do the heavy lifting and are identical across every backend:

- **`report_data = sha512(nonce ‖ enclave_public_key)`** — the value the enclave writes into the quote's fixed 64-byte report-data field. It simultaneously proves freshness (the client's nonce) and binds the quote to the exact key the client will encrypt to, so a captured quote can't be replayed or reused to vouch for an attacker-controlled key.
- **TCB (security-version number)** — lets the client reject an enclave running an outdated/vulnerable TCB even if everything else checks out.

Two silicon vendors, one client

Liquify's fleet is mixed: some hosts are Intel (TDX), some are AMD (SEV-SNP). A request may be routed to either, and the client must verify either — without ever blurring the two.

Almost nothing about the evidence is shared between them:

	Intel TDX	AMD SEV-SNP
Evidence	TD Quote v4 (~5 KB, nested)	attestation report (exactly 1184 bytes, flat)
Measurement	MRTD	launch measurement — **a different value for the same image**
TCB	`TEE_TCB_SVN` vector	4 SVNs packed in a u64 (bootloader/TEE/SNP/microcode)
Chain	PCK → PCK CA → Intel root (ECDSA P-256)	VCEK → ASK → ARK (RSA-4096 PSS certs, EC P-384 leaf)
Signature	r‖s **big**-endian	r‖s **little**-endian, 72-byte fields
"Not confidential" bit	TD attributes DEBUG	guest policy DEBUG / MIGRATE_MA
Extra binding	QE report vouches for the ephemeral attestation key	VCEK's AMD extensions bind it to this chip + this TCB

The one thing the two vendors agree on is that each will sign **64 bytes of the guest's choosing** — which is exactly a SHA-512 digest. That is where `report_data` goes, and it is the entire reason one client can verify both.

So the trust policy is a **list of platforms**, one per vendor, each with its own roots, measurements and TCB floor:

```
TrustPolicy
├── PlatformPolicy(backend="tdx",     roots=[Intel root], allowed_measurements={MRTD…},   min_tcb_components={tdx_module: 3})
├── PlatformPolicy(backend="sev-snp", roots=[AMD ARK],    allowed_measurements={launch…}, min_tcb_components={microcode: 213, …})
└── shared: GPU-CC requirement, GPU roots, freshness
```

Evidence names its backend; that selects **exactly one** entry, and everything downstream — roots, allow-list, TCB floor — is read from that entry. Two properties follow, both tested:

- AMD evidence is never judged against Intel's allow-list (**no cross-vendor confusion**).
- Evidence from a platform the policy doesn't list is refused *before any crypto runs*.

On the enclave side the same image boots on either vendor: `LIQUIFY_ATTESTATION_BACKEND=auto` detects the platform through the kernel's configfs-TSM interface and picks the matching Producer. Asking for hardware that isn't present is a **refusal to start**, never a silent downgrade to a weaker backend.

One thing the CPU quote cannot tell you

Neither a TD Quote nor an SEV-SNP report has anywhere to put a claim about the GPU. So on real silicon the CPU attestation says *nothing* about confidential computing on the GPU — `RawAttested.gpu` comes back empty, and a policy that requires GPU CC will correctly refuse unless a **separately-signed NVIDIA GPU report** (Phase 2, `attest_gpu.py`) is present and verifies against its own pinned root. This is the fail-closed property that stops "confidential GPU" quietly degrading into "trust us".

Testing the real verifiers without a CVM

`LIQUIFY_HWSIM=1` swaps the *silicon* for byte-exact quote generators: a real v4 TD Quote layout and a real 1184-byte SEV-SNP report, signed by stand-in Intel/AMD PKIs. The evidence flows through the **production** `TdxVerifier` / `SevSnpVerifier`, so the parsing, chain walking, endianness, and binding logic are exercised in CI today. What remains untested off-hardware is the ioctl that fetches the quote and the vendors' live collateral.

This is safe by construction: a client that pins Intel's or AMD's *real* roots can never accept a simulated quote, because the chain terminates at the wrong CA. The only way to be fooled is for an operator to publish a manifest that pins the simulated roots — so those roots carry `SIMULATED` in their subject, the manifest records `simulated: true`, and the SDK prints a warning on every run that loads one.

Establishing trust: the signed release manifest

A subtle but critical point: **where does the client get the "known-good" measurement and roots it checks attestation against?** If it fetched them from the server it's verifying, that would be circular — a rogue server could hand over whatever policy it liked.

Instead:

1. Liquify's build/release process computes the measurement (independently, from a reproducible build) and writes a **release manifest** listing the allowed measurement(s), attestation roots, TEE type, and minimum TCB.
2. The manifest is **signed with a Liquify release-signing key**. A live server never holds this key, so it cannot forge a manifest.
3. The client **pins the release-signing public key** (baked into the SDK in production; read from `provisioning/` here) and verifies the manifest with it before deriving its `TrustPolicy`. It trusts nothing the server says about what to trust.

So the chain of trust is: pinned release key → signed manifest → trusted measurement + roots → attestation verification → sealed session. Only the first link is "pinned"; every other link is cryptographically verified.

(`release.py`, `tools/build_release.py`.)

Why the layers are split this way

- `liquify_confidential/` **is shared** by client and enclave so the two can never drift on wire format or context strings.
- **Attestation verification lives in the client SDK** (not the server) on purpose: the guarantee is only meaningful if the *customer's* code checks it. It's the part they audit.
- **The measurement is computed from code**, so any change to the enclave changes the measurement — and a pinning client rejects an enclave running unexpected code. In production this is anchored by a reproducible build an auditor can reproduce independently.
- **Session binding via AEAD associated data** ties every ciphertext to a specific attested enclave, closing replay/relay attacks between enclaves.

Honest limitations (carry these into the sales narrative)

- Plaintext **does** exist in-use inside the enclave. The guarantee is "no software or operator in the path can read it," resting on TEE hardware soundness + attestation — **not** "mathematically impossible to decrypt." Be precise with technical buyers.
- Confidential mode adds low-double-digit-percent overhead (encrypted memory + PCIe); measure it and price the confidential tier accordingly.
- Requires Hopper/Blackwell CC mode (H200/B200 qualify; the RTX 3080 test rig cannot exercise the real CC path).
- The control plane must stay **content-blind**. The moment any component needs to read the prompt (a content-based router, a payload-logging system) the E2E property is punctured. Route and meter on metadata only.