AGENT COMPUTE INFRASTRUCTURE

Compute You Can Prove

A control plane for agent compute: an OpenAI- and Anthropic-compatible API plus metered code sandboxes, where every unit of work is sealed on arrival, metered at the gateway, bounded by a budget the caller sets, paid in prepaid stablecoin, and signed on the way out as a receipt anyone can verify without trusting us.

RECEIPTS	ATTESTATION	SETTLEMENT	ON-CHAIN
secp256k1 · EIP-191 public verify	Intel TDX · fail-closed base-image pin	USDT · x402 enforced in prod	Robinhood Chain 4663 contracts written, not deployed

VERSION
3.1 — September 2026

SUPERSEDES
3.0 — September 2026

STATUS
Shipping · pre-audit

WEB
buildsable.com

ABSTRACT

Autonomous agents consume compute the way processes consume CPU: continuously, unattended, and without a human in the loop to notice when something goes wrong. Two failures follow. The first is disclosure — an agent's prompts and the code it executes are its entire working memory, streamed to counterparties who log it. The second is unaccountability — nobody can say afterwards what actually ran, on what, for how much. **Sable Network** is the control plane that closes both: a drop-in OpenAI- and Anthropic-compatible endpoint plus metered code-execution sandboxes, where inbound payloads are sealed on arrival, nothing content-bearing is persisted, every unit of work is metered and pre-authorized against a hard budget, settlement is prepaid stablecoin or inline x402, and every response carries a secp256k1-signed receipt a third party can verify without trusting us. Around that core sits a proof layer — named and versioned model configurations, notarized fingerprints for work Sable never ran, replayable capsules, receipted public benchmarks, and a machine-checkable credential describing an agent's real spending bounds — and an on-chain half that makes those signatures checkable by a contract rather than by an endpoint Sable controls. This paper states what is built, what is switched on, what is neither, and why the difference is printed rather than smoothed over.

CONTENTS

WHAT CHANGED SINCE V2.0

Four things, and three of them are corrections rather than additions.

Added. A proof layer of nine subsystems (§10) and a set of on-chain verification contracts targeting an EVM L2 (§11). All of the gateway half is deployed; nearly all of it requires an operator to switch it on, and the table in §12 says which.

Corrected. v2.0 said a confidential receipt's `response_bound` flag becomes true once a per-response signature verifies. It does not, and does not today: the check needs the response bytes and the handlers do not pass them, so the flag is **false on every receipt this deployment issues**. §08 explains what was wrong with the old check and why the weaker true state ships instead.

Corrected. The confidential tier's measurement pin is on the enclave's *base image*, not its workload build. That is a narrower claim than v2.0 implied, and §08 states the narrower one.

Corrected, and this one was harmful. v3.0 said SABL was "migrating" to Robinhood Chain. It is not, and no migration was ever planned. **SABL** (Solana) and **SABLE** (Robinhood Chain) are two separate tokens with different symbols, both live, with no bridge or conversion between them — §06 now prints both addresses. The error mattered because Sable's own FAQ warns holders that anyone telling them to "migrate" is running a scam, so this document was supplying the exact language a scammer would want. v3.0 also said "the only official mint address" while a second official Sable token existed on another chain, which branded Sable's own token an impersonation.

01 The delegation problem

You cannot delegate authority to a system you can neither keep private nor hold to account.

When a person uses a model, they disclose something: a question, a draft, a fragment of a plan. When an *autonomous agent* uses one, it discloses everything — its objectives, its tools, its private state, the contents of the documents it is reasoning over, and increasingly the code it wants executed and the secrets that code needs. It does this not once per session but continuously, as a condition of functioning.

Disclosure is only the first half of the problem, and it is the half the industry has started to name. The second half is quieter and, operationally, bites sooner. An agent left running is a process with a wallet. It can loop. It can spawn sub-agents. It can be prompt-injected into doing something expensive. And when the bill arrives, the questions that matter — what ran, on what hardware, under whose budget, and can you prove it — have no answer, because the systems in the middle were built for a human clicking a button, not for a machine spending money unsupervised.

> **Two properties make delegation safe: the delegate cannot leak what it was given, and the principal can bound and later verify what it did. Most infrastructure offers neither. Both are engineering problems, not policy problems.**

Sable Network addresses both at the same layer, because they are the same layer. The wedge is deliberately narrow: a developer adopts it by changing one line.

```
# Before
client = OpenAI()

# After — sealed on ingress, metered, budget-bounded, receipted
client = Sable(base_url="https://api.buildsable.com/v1")
```

Everything else — chat completions, embeddings, streaming, tool use — keeps working unchanged, and the Anthropic Messages API is served from the same gateway, so an Anthropic-shaped client works by pointing `ANTHROPIC_BASE_URL` at it. Wire compatibility is not merely convenience; it is **portability insurance**. If Sable disappeared tomorrow, the same code runs against any other compatible endpoint with one URL change. Trust that requires lock-in is not trust — it is capture.

02 The control plane

Sable does not sell you a machine. It sells the metering, the budget, and the proof that sit over one.

The product is a gateway, and the gateway is the whole thesis. It accepts two kinds of work through one credential, one balance, and one policy surface:

- **Inference** — chat completions and embeddings, metered in tokens.
- **Sandboxes** — code execution, one-shot or in a persistent warm session, metered in vCPU-seconds and gigabyte-seconds.

Both are billed against the same prepaid balance, bounded by the same per-key spend caps, subject to the same resource allowlists and expiry, recorded in the same metadata-only event table, and signed with the same receipt key. An agent that can run inference and execute code needs one credential and one budget, not two vendors and two invoices.

The catalog is published under **stable Sable IDs**, decoupled from upstream naming churn. A few of them are load-bearing: the flagship family — `sable`, `sable-fast`, `sable-max` — where each id is kept pointed at a strong frontier engine (currently Claude Opus 5 behind `sable`), disclosed rather than hidden: each flagship id carries a chain of engines it can fail over across, and the engine that actually served is recorded in every signed receipt. Around it sits a slate of frontier models served through the gateway, where the upstream provider sees Sable as the caller rather than the buyer. That anonymization is worth having and worth being precise about: **anonymized access is not the confidential tier**. Only TEE-attested models are confidential; on every other model the upstream host sees the plaintext, exactly as the privacy contract states. Between the two sits a **double-blind private lane**: ids pinned to a second, privacy-focused intermediary, so the request reaches the model through a provider that sees Sable rather than the caller. A pinned route never fails over — a failover that changed the privacy path would be worse than an error — and the intermediary's default system prompt is disabled at egress.

What makes this a *control plane* rather than a reseller is where authority sits. Metering is computed at the gateway from what the gateway observed, not from what a backend reported it did. Budgets are enforced before an upstream call is made, not reconciled afterwards. Attestation is verified against a pinned measurement by the gateway, and if verification fails the request is refused rather than quietly downgraded. The backend is, deliberately, the least trusted component in the system — which is precisely what makes it replaceable.

> THE DESIGN CONSTRAINT
>
> Every feature must compose with the OpenAI-compatible surface rather than replace it. A caller written against the gateway on its first day keeps working as the compute underneath changes — different upstream providers, attested enclaves, and eventually machines Sable does not own. The interface is the contract; the substrate is an implementation detail. This is what lets the supply side change without breaking the demand side, and it is why every subsystem in §10 was built by composing the existing chat, billing and receipt paths in-process rather than by adding a second one beside them.

03 The privacy contract

A short list of invariants, enforced in code and testable from outside.

Sable's guarantees are expressed as a small set of blunt invariants that the gateway is built around, rather than as a policy document that describes intentions.

- **Prompts, completions, and submitted code are never written to disk or logs.** Plaintext exists only inside named, in-frame code paths: the egress shim immediately before an upstream model call, the request-scoped sandbox execution path, and the translation frames that adapt Anthropic- and Responses-shaped requests onto the chat handler. Adding a new plaintext surface requires amending that inventory, and the inventory is checked into the repository rather than described from memory.
- **Inbound payloads are sealed on ingress.** The request body is serialized and encrypted with AES-256-GCM before any routing decision is taken and before any network egress.
- **Persistence is metadata only.** One event row per unit of work: kind, unit, quantity, resource, model, provider, tier, node, token counts, cost, latency, status, error class, timestamp. The sandbox table has *no column* that could hold code, stdout, stderr, or produced files. The only content-derived value stored there is a truncated SHA-256 fingerprint.
- **Credentials are stored as Argon2id hashes.** The plaintext of an API key is returned exactly once at creation and is never re-derivable; a short prefix is kept in the clear only for labelling.
- **Telemetry respects the contract.** Request IDs, key IDs, model strings, token counts, latencies, and content *fingerprints* are loggable. Prompt text, message arrays, completion content, and upstream response bodies are not. Upstream 4xx bodies are reduced to content-free error classes at the provider boundary, because a moderation or validation rejection can echo the prompt back at you.
- **Secrets live in the environment.** The master key is required at boot; the service refuses to start without one.
- **The portal never displays content.** History shows metadata. It cannot show a prompt, because the prompt was never kept.

The exceptions, named rather than implied

Several features are worthless if the gateway forgets what it was given: a scheduler cannot re-run code it does not hold, a share link cannot serve a file it discarded, a mailbox cannot deliver a letter it deleted on arrival, and a replay capsule cannot replay a request it kept only a hash of. Each such feature is a **deliberate, enumerated amendment** to the contract rather than a quiet exception, and each takes the same posture: the content is **AES-GCM-sealed at rest under the master key**, opened in-frame only to answer its own owner, never logged, and destroyed on deletion or at a hard TTL. That posture is *ciphertext at rest under Sable's key*, which is meaningfully weaker than end-to-end encryption; every surface that offers one says so in those words rather than letting "sealed" do work it has not earned.

Two honest carve-outs inside those exceptions are worth printing, because omitting them would be the same kind of silence the list exists to prevent. A stored knowledge-base chunk's *embedding vector* is kept unsealed beside its sealed text — an embedding is a lossy but real encoding, and because this gateway also serves an embeddings endpoint, a reader of a database dump holding any key could embed a guess and confirm it by cosine.

And a caller-chosen *label* — on a share, a sealed call, a replay capsule — is public and readable at rest by construction, exactly as intended, which means it is not a place to put anything private.

The discipline that makes all of this real is a single question asked of every change: *could a malicious operator, or a curious engineer with production access, reconstruct the payload from what I am about to add?* If yes, it does not ship. Two consequences are worth stating plainly, because they are costs and not features. Abuse response is necessarily **account-level rather than content-level** — we cannot search content we never stored, so reports are actioned against accounts and against fingerprints a reporter supplies, never by scanning traffic. And a sandbox run retried with an idempotency key **replays its receipt, not its output**: the metadata was recorded, the output never was.

04 Architecture

Seal on ingress, authorize against a budget, dispatch once, meter what happened, sign the result.

The control plane is a single deployed component: an HTTP gateway speaking the OpenAI wire protocol, with an Anthropic-shaped façade, an OpenAI Responses façade, and an MCP server on the same port. Sandbox work is dispatched from it to an enrolled runner node — the gateway claims a per-job concurrency lease transactionally before it dials anything. Its internal flow is the product.

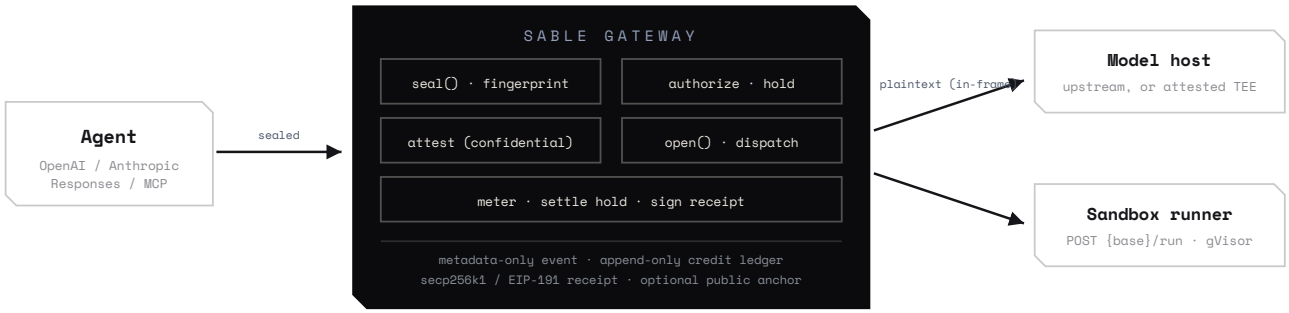

FIG. 1 — ONE PIPELINE, TWO KINDS OF WORK. PLAINTEXT EXISTS ONLY BETWEEN OPEN() AND THE DISPATCH IT WAS OPENED FOR.

Envelope encryption

On ingress the request body is serialized and sealed with **AES-256-GCM** under a fresh 96-bit nonce. The sealed envelope, not the plaintext, is what the routing layer carries. The same primitive gives integrity: any tampering with the ciphertext fails authentication on open. A short SHA-256 fingerprint is computed pre-seal for safe logging and correlation — it identifies a request without revealing it, and it is the value that later appears on the receipt.

The egress shim

Decryption happens exactly once, immediately before the dispatch it exists for, inside a deliberately small code path. This is the entire privacy surface. The plaintext does not leave that frame: it is not logged, not persisted, not echoed into errors, and not retained once the response has streamed back. Backend error text is reduced to fixed classes rather than forwarded, because a backend's own error message is one of the easiest ways to leak a payload — or, as we learned by shipping it, the gateway host's filesystem layout. An opt-in scrub (`sable_scrub`) can additionally redact secret- and PII-shaped spans — emails, wallet addresses, API-key shapes, long hex — from the outbound payload inside this same frame, before it reaches any vendor; nothing scrubbed or unscrubbed is persisted, and only the redaction count is ever traced.

Metadata-only accounting

What Sable retains per unit of work is one row:

```
usage_events (
  kind · unit · quantity · resource
  model · provider · privacy_tier · node_id
  prompt_tokens · completion_tokens · total_tokens · cost_micro_usd
  latency_ms · status · error_class · clamped · created_at
)
// absent by construction: prompts, completions, embeddings, submitted
// code, stdout, stderr, or anything they could be reconstructed from.
```

One table describes tokens and vCPU-seconds alike — the `kind` / `unit` / `quantity` triple is what makes a single billing, budgeting and reporting path work across both, and it is why every subsystem added since has been able to meter itself without a new enforcement path. The portal, the usage stream, spend caps and the invoice all read this row. None of them can surface content, because content was never stored.

05 Metering, budgets & delegation

A budget that is checked after the money is spent is not a budget.

Credit is an **append-only ledger**. A balance is the sum of its rows; nothing is updated in place, so the audit trail cannot be rewritten by the code that spends against it. A cached balance column is maintained inside the same database transaction as every ledger append, purely as an index — a reconciliation job recomputes the sum on a schedule and *alerts on drift without silently repairing it*, because a self-healing cache destroys the evidence of the bug that caused the drift.

Spending is **pre-authorized, not post-paid**. Before any upstream call the gateway places a *hold* for the request's worst-case cost — for a sandbox, every second of its declared timeout at its declared vCPU and memory; for inference, an estimate floored at a configured minimum. Spendable credit is the balance minus outstanding holds. The hold is released at settlement, on every error path, and by a sweeper if the process dies while holding one. This closes a race that a simple balance check does not: without holds, a balance of one micro-dollar authorizes an unbounded number of concurrent jobs, which is exactly the failure mode a runaway agent produces.

Debits are **idempotent at the database**, enforced by a partial unique index on the usage reference rather than by application memory, so a retried metering write cannot double-bill. Side-effecting calls get a second layer: `POST /v1/sandboxes` accepts an `Idempotency-Key` that is claimed by a row insert *before* execution, so a client retry cannot execute the code twice.

Per-key controls

Agents fail in ways human users do not, so every key carries optional, enforced limits: a **spend cap** over a refreshing window (day, week, month, or lifetime — over it, `402` before any upstream call), a **model and resource allowlist** (outside it, `403`), an **expiry** (past it, `401`), a per-key **rate limit** (over it, `429` with `Retry-After`), and an optional **circuit breaker** — a rolling spend-velocity limit that *freezes* the key rather than merely refusing the next call, and requires an explicit unfreeze. A dashboard cannot intervene in-path; this can.

Delegated sub-keys

The interesting case is an agent that spawns sub-agents mid-run, with no human available to mint credentials. `POST /v1/keys/delegate` is authenticated by the *parent API key* rather than a wallet session, and issues a child that is **at most its parent on every axis.** Two rules are deliberately the opposite of the obvious implementation, because the obvious implementation silently defeats the feature:

- **Omitting a field inherits the parent's limit** — it does not mean "unlimited".
- **An allowlist intersects.** A child requesting models its parent cannot use is refused, not issued unrestricted.

Two tree properties make it safe rather than decorative, and both are recursive queries in the database rather than bookkeeping in the application: spend is summed over a key *and all its descendants* (otherwise minting a child is a trivial way around your own budget), and revocation *cascades over the subtree* (otherwise revoking a key does nothing to an attacker who delegated first). Depth is capped, and tier is inherited rather than settable — because tier selects routing, and on the confidential tier it selects which hardware attests the call.

The intended pattern is one short-lived, tightly-scoped key per task: a runaway sub-agent is then bounded by construction rather than by somebody noticing. §10 adds the readable form of the same fact — a signed credential that publishes the *tightest* cap in a key's whole delegation chain, so a counterparty can check the real bound rather than the one the key nominally carries.

06 Payment: prepaid USDT and x402

Machines should be able to pay for compute without a card, a signup, or a human.

Accounts are funded by transferring **USDT** to a treasury address on a supported EVM chain and submitting the transaction hash. Verification is done by the gateway against its own JSON-RPC node, and it is deliberately strict: the receipt must show success, the log must be an ERC-20 `Transfer` emitted *by the configured token contract*, the recipient must be the treasury, the sender must be a wallet already linked to the account, and the transfer must have the required confirmations. Replay is prevented by a uniqueness constraint on the chain, transaction hash, and log index together, so a given transfer credits exactly once, globally.

Two of those checks exist because their absence is exploitable rather than merely untidy. Binding the sender to a linked wallet is what stops anyone from claiming a stranger's deposit by submitting its hash first. Requiring the log to come from the configured contract is what stops a lookalike token, deployed at an address a careless operator pasted, from crediting real balance. For the same reason token addresses are **operator-configured and deliberately not hardcoded**: a wrong address either fails to credit legitimate deposits or credits an unrelated token, and that decision should be made deliberately by someone checking the official contract list.

Deposits are likewise accepted in USDT on **Solana**, where a transfer is attributed to an account by a transaction memo rather than a linked sender wallet, and is verified only at finalized commitment. In production, deposits are live on Ethereum, Arbitrum One, and Solana, and **billing enforcement is on**: an out-of-credit request is refused rather than merely recorded. Metering and the ledger run on every deployment regardless; whether an unpaid request is refused remains a deployment flag, so a new deployment can observe billing against real traffic before it starts turning anyone away.

For agents that would rather not pre-fund, the gateway speaks **x402**. An out-of-credit request returns `402` with an `accepts` array describing exactly how to pay; the agent settles and retries with an `X-PAYMENT` header, which is verified and credited before the handler runs. The scheme is named `sable-usdt-onchain` rather than the canonical gasless `exact` flow, because USDT implements neither EIP-3009 nor EIP-2612 and a pull-based settlement is simply unavailable for it. Naming the scheme honestly is cheaper than explaining later why the standard one did not work.

The same rail runs in the other direction without Sable touching the money. A seller can publish a service whose price is quoted in an x402 challenge naming the *seller's own* wallet; the buyer pays that wallet directly on chain, and Sable verifies the transfer, runs the work, and signs a receipt. Nothing is custodied, which is deliberate: holding the funds would reopen a money-transmission question that the prepaid-credit model was designed to avoid.

ON TOKENS

The networks that paid for compute in their own token did not discover a new economic primitive; they discovered that paying for existence rather than for work attracts fabricated supply, and that a payout denominated in something the recipient cannot spend is not a payout. Compute is bought and sold here in dollars. If supply is ever paid, it is paid in dollars too. None of that has changed, and none of it will.

What changed after v1.0 of this document, which said Sable would not issue a token at all, is that in August 2026 the team launched **SABL**, a community token on Solana (mint `DaPayqzdCXcrmvgz9Wx7MySipXxcSofGPtkMgVdqpump`; fixed supply, mint and freeze authority revoked). That version stated, as plainly as everything else here, that SABL had no role in the platform and was not accepted as payment.

As of 2026-09-02 that stance is reversed on exactly one point. SABL gains a single utility: it can be used to pay for Sable compute and Sable Pro subscriptions, and paying in SABL burns it and earns a discount. This rail is **rolling out, not live** — the settlement code ships behind a flag that is off, the deployment reports its status as such, and going live is gated on counsel rather than on engineering. The guardrails are firm and are not softening. Paying in SABL is optional and never required, because Sable is always priced in dollars and payable in USDT. It does not gate access: nothing on Sable requires holding SABL. It confers no yield, no revenue share, no governance, and no claim on Sable Network. It is a consumptive payment utility, not an investment. Supply-side operators, when they exist, are paid in dollars, never token emissions.

There are **two** Sable tokens, on two chains, and they are distinguished by their symbol. **SABL** is the Solana token above. **SABLE** is a separate token on **Robinhood Chain** (§11), contract `0xf7894d31D569e6330592D346ecfeFdf4257F3EC1`, which reports `name()` "Sable Network" and `symbol()` "SABLE" on chain id 4663.

They are separate tokens. There is no migration, bridge or conversion between them, and none is planned. Anyone instructing you to migrate, bridge or convert SABL into SABLE — or the reverse — is running a scam. Earlier revisions of this document described SABL as "migrating" to Robinhood Chain; that was wrong, it is corrected here, and the correction is recorded in §13.

The guarantees in the paragraph above are not symbol-specific and do not weaken on an EVM: **neither token** confers yield, revenue share, governance or any claim on Sable Network, and **neither** gates access to Sable. Nothing described in §11 is payable in either token, emits a token, pays yield, stakes, bonds or votes. The pay-in-SABL rail described above is specific to SABL and is still rolling out; **SABLE carries no platform utility at all.**

Each chain has exactly one official address, and they are the two printed in this section: the Solana mint above for SABL, and the Robinhood Chain contract above for SABLE. Anything else using either name, on either chain, is an impersonation. Distribution for **SABL on Solana**, as disclosed by the team and checkable on that chain: **8% of supply is held by the project treasury; the remaining 92% circulates on the open market.** That figure describes SABL only and says nothing about SABLE supply.

07 Verifiable receipts

A guarantee you can check beats a guarantee you have to believe.

Every billable response carries a **signed receipt**: a compact, metadata-only statement of what was processed and under what terms, signed with a **secp256k1** key over an **EIP-191** digest. The signing key is derived deterministically from the deployment's master key, and its address is published. Inference and compute use distinct shapes — `v:1` describes a token-metered call, `v:2` a unit-metered execution, and `v:3` adds the serving node's own counter-signature — so a verifier is never asked to reinterpret one as the other.

```
{
  "v": 3,
  "request_id": "3f9c…",
  "content_fingerprint": "a1b2c3d4…",   // sha256 prefix, NOT content
  "kind": "sandbox", "resource": "python:3.12-slim",
  "unit": "vcpu_seconds", "quantity": 12,
  "cost_micro_usd": 178,
  "status": "ok", "exit_code": 0, "duration_ms": 6042,
  "operator": { "node_id": "node-4c91…", "verified": true },
  "logging": "metadata-only",
  "issued_at": "2026-09-13T12:00:00Z"
}
```

Because the scheme is standard secp256k1 / EIP-191, a receipt verifies with tooling the wider ecosystem already trusts — no Sable-specific library required:

```
// viem — verify against the published signer address
await verifyMessage({ address, message: atob(receipt), signature })  // -> true
```

The gateway also exposes `GET /v1/receipts/pubkey` and `POST /v1/receipts/verify` for parties who prefer a hosted check, and §11 describes a contract that answers the same question without asking Sable at all. Non-streaming responses carry the receipt in headers; streaming responses emit it as a trailing `sable.receipt` event, since the headers are long gone by then.

What rides inside the signature

A receipt is not only a bill. Several optional blocks are stamped into the same signed payload, each present only when it applies, so a receipt without them stays byte-identical to one issued before the block existed:

CONTEXT The fingerprints and order-sensitive root of the documents the call declared as its retrieved context.	ACTIONS Tool calls the model emitted — name, argument digest, order-sensitive root. What was requested, not what was executed.	POLICY The id and rules-hash of the policy that governed the call, so a later edit is visible.
ENGINE The upstream engine that actually served, whenever it differs from the public model id.	ATTESTATION The verified TDX measurement, TCB status and binding flag on a confidential call (§08).	OPERATOR The serving node's ed25519 counter-signature over the input and output fingerprints (§09).

Every one of those is a fingerprint, an identifier, or a flag. None can carry content, which is why the receipt is safe to hand to a third party — and why a receipt can be marked public by its owner and served from an unauthenticated endpoint without any filtering step to get wrong.

Chains and proofs

A billable request may carry a `sable_run_id`, which joins its receipt to a per-account hash chain: `chain(n) = sha256(chain(n-1) || sha256(receipt))`, appended under a row lock so the chain cannot fork. A whole agent session then has one head hash that a third party can recompute from the individually-signed receipts beneath it — and a signed run proof, or a compliance audit pack, states the head, the count, the total cost and the span in one verifiable artifact. That head is also what gets batched into a Merkle root and published to a public chain (§11), so the question "did this record exist at that time" stops depending on Sable being reachable or honest later.

It is worth being precise about what a receipt does and does not establish, because overstating it would undo the point. A gateway-signed receipt proves that *this deployment attests* to having processed a payload with that fingerprint, under that tier, at that metered cost — and that no field has been altered since. It is an accountable statement by a named party, not a proof of physics. It does not prove the model ran, that the token counts are honest, that a sandbox executed the code its fingerprint names, or that the output was any good. What upgrades it is the attestation block described next, and what makes it undeniable is the anchor described in §11. The shape does not change in either case, so an integration written today inherits the stronger guarantee without a code change.

08 Confidential execution & attestation

Verify the hardware before you send it anything — refuse rather than downgrade — and publish the weaker state when the stronger one is not true.

On the confidential tier, a request is routed only to a backend running inside an **Intel TDX** trusted execution environment, and only after the gateway has cryptographically verified that backend's attestation. Verification is real, not a label: the DCAP quote is verified in-process with a pure-Rust verifier against Intel collateral, the report's measurement must appear in an operator-pinned **allowlist**, and the TCB status must be up to date. Anything less **fails closed** — the request is refused with an error, never served in plaintext by a different path. A confidential request for a model that is not mapped to an attested backend is likewise refused rather than silently downgraded, which is an invariant enforced by a table-tested routing function after an early version of this code stamped one enclave's attestation onto a different model's response.

What the pinned measurement covers — and what it does not

A TDX quote carries two measurements that matter here: **MRTD**, the enclave's base image, and **RTMR3**, the specific workload build running inside it. The allowlist accepts either, and which one an operator pins is a real choice with a disclosure consequence rather than a configuration detail.

This deployment pins MRTD. The reason is empirical: a census of the production backend found *three distinct RTMR3 values against one unchanging MRTD* — it serves from a pool of enclaves behind a balancer. Pinning a single workload measurement therefore verifies only the fraction of requests the balancer happens to route to the pinned instance, which is exactly why every past workload re-pin appeared to fix the tier instantly and then decayed silently over the following days. Five outages were recorded and mis-diagnosed as backend rotations before that was measured.

So the honest statement of what a passing check proves today is narrower than the natural reading, and narrower than the previous version of this document implied:

PROVEN	NOT PROVEN
A genuine Intel TDX enclave, running the expected base image, with an up-to-date TCB, reachable at the backend's attestation endpoint.	The specific workload build inside it. And because the GPU confidential-computing configuration was only ever covered *by the workload measurement*, gpu_verified is **recorded, not proven** — the tier must not be read as a GPU-CC guarantee.

Response binding — a correction

An attestation proves that a correctly-measured enclave exists. It does not, by itself, prove that *your* response came from it. That second, stricter property is what the receipt's `response_bound` flag is for, and the flag deserves a plain statement rather than a footnote.

> CORRECTION TO V2.0
>
> `response_bound` is **false on every receipt this deployment issues today.** Binding requires re-hashing the bytes actually received and comparing them to what the enclave signed; the chat handlers call an entry point that has no response bytes, so the gateway declines to make the claim. Until 2026-09-11 the flag was set after verifying an enclave signature over text that was never compared against this request or its response — which asserted "this specific response came from that enclave" while proving only that the enclave had signed *something*. A signed claim about hardware cannot rest on that, so the check was reduced to one that can only return the weaker answer, and the weaker answer is what ships. `verification: "tee-attested"` is unaffected: the enclave, its measurement and its TCB are still verified per request.

Two further gaps remain even once the handlers pass the bytes, and both are worth knowing before relying on this tier. The **trust root** is an external enclave: Sable verifies *its* attestation rather than measuring a binary Sable builds. And the **transport** terminates at the backend host, not at the enclave — RA-TLS is unbuilt, which is precisely why re-hashing the received response is the load-bearing step rather than a formality.

Attest-then-call, enforced by types

Attesting one enclave and then sending the payload to another would produce a receipt describing hardware that never saw the request — the worst failure this system could have, and an easy one to introduce during a refactor. So attestation returns a *handle*, and the call and signature-verification functions take that handle rather than a name or an index. Routing to an enclave you did not attest is not a rule to remember; it does not compile.

The measurement pin is an allowlist rather than a single value for an operational reason learned the hard way, described above. A background loop re-verifies attestation continuously, a fail-closed window raises an operator alert within roughly two minutes rather than being discovered days later, and the whole DCAP path is exercised in CI against a captured quote and its collateral, so the code that both historical outages lived in is no longer only testable against a live backend.

09 Sandbox compute

Agents write code. Something has to run it, and the same rules should apply.

`POST /v1/sandboxes` executes a job — an image, an argv, code on stdin, a timeout, a vCPU and memory request — and returns its stdout, stderr, and exit code exactly once, or streams them as they are produced. It is billed in **vCPU-seconds and gigabyte-seconds**, on the same key and the same balance as inference. In production the work runs on a **Sable-operated house node** under gVisor isolation, enrolled through the same registry, heartbeat, and lease machinery a future fleet would use — one machine today, and the public node listing says exactly that.

A job may also hold a **persistent warm session**: a container an agent execs into repeatedly, with a workspace it can read and write and snapshot. Warm time is billed at a fraction of the active rate, scaling with both vCPU and memory and rounded up, so a second in which the container is alive is never free — the alternative, billing only active execution, prices idle capacity at zero and invites holding it forever.

Pricing memory as well as CPU is not a rounding detail. Memory is what actually bounds how many sandboxes a host can run concurrently, so charging for CPU alone means the cheapest possible request can consume the capacity you had left to sell. Any metering scheme that ignores the binding resource is a scheme that rewards exhausting it.

The privacy contract applies without exception. The submitted code, its environment variables, and both output streams cross the module in-frame, are returned once, and are never persisted or logged — the table has no column that could hold them. A session snapshot archives the workspace *on the node*, under the runner's own state directory with its own expiry; the gateway stores an id, a byte count and a timestamp, and no workspace bytes cross into the database. This has a consequence worth stating rather than burying: a run replayed via its idempotency key returns the recorded receipt with a `replayed` marker and **empty output**. The metadata was kept; the output never was.

Who ran it, proven by the machine that ran it

The serving node holds an ed25519 key generated on the box, and counter-signs every run over `input_fp` and `output_fp` — fingerprints only, never content. The gateway verifies that signature against the node's enrolled key (trusted on first observation, then pinned, so a silent key change is refused rather than attributed) and checks that the node's claimed input fingerprint equals the gateway's own hash of the submitted code, binding the attestation to this exact request. On success the receipt carries an `operator` block and bumps to `v:3`.

The label on that block is deliberately modest: an operator counter-signature proves **which machine served the request** — who to credit and who to blame — and does *not* prove the work was performed correctly. Correctness is the confidential tier's rung of the ladder, not this one. A node that has not been approved by an operator is never scheduled at all, which is a vetting gate rather than a courtesy.

Defaults are chosen so that the safe posture is the one you get by not thinking about it: **network egress is denied unless requested**, and where a session declares an allowlist it is enforced by hostname without intercepting TLS — stated in those words, because "secrets proxy" would claim more than is built. The filesystem is read-only with scratch space in memory, capabilities are dropped, privilege escalation is disabled, and code is fed over stdin

so it never appears in a process listing. A local Docker backend exists for development and single-tenant self-hosting and logs a loud warning at boot, because a container is not an isolation boundary for hostile code and pretending otherwise in production would be indefensible.

Failure classification is part of the billing contract. A non-zero exit is the caller's own program failing: it ran, and it is billed. A timeout held the slot and is billed. But a backend that was never reachable, or a runner that failed to start, is recorded and **costs nothing** — an early version billed for runs that never happened because a daemon-level failure exit code went down the success path, which is the kind of bug only ever found by running the thing for real.

10 The proof layer

Nine subsystems, one idea: take something an agent or a vendor currently just asserts, and make it checkable by someone with no reason to trust the asserter.

A signed receipt answers one question — what did this gateway process, and at what cost. The subsystems below extend the same signature, the same verification endpoint and the same anchor streams to adjacent questions that today have no answer at all: which configuration served me, did this number come from a real run, has the behaviour changed since last month, what did this agent declare it was reading, and what is this counterparty actually allowed to spend.

All of them are built and deployed on the production gateway. **Nearly all of them require an operator to switch something on before they do anything**, and one is deliberately not switched on at all. §12 states which is which, per subsystem; this section describes what each one proves and — the part that matters more — what it does not.

Intelligence Engine — a configuration with a name and a hash

An Intelligence Engine build compiles a configuration into an **immutable, versioned artifact** with a stable callable id, `engine/<slug>`, which can be passed anywhere a model id is accepted. A build names a base model, a system prompt, sampling parameters, a guardrail rule set and a key policy — all of them gateway-side settings the receipt can already attest — and it is **emphatically not fine-tuning**: no weights are trained, adapted, merged, quantized, or hosted.

What is being sold is therefore identity, not capability. One stable name, version numbers that are never reused, and a digest over the whole configuration that is stamped on every receipt the engine serves. Two receipts carrying the same digest were produced by the same setup; two carrying different digests were not, and the day it changed is visible. The system prompt itself is sealed at rest and never appears in the canonical spec — the spec carries its SHA-256 instead, which makes the spec content-free by construction rather than by filtering.

Sable Notary — a receipt for work Sable did not run

Every other receipt here describes work that crossed the gateway, which makes the proof layer available only to people willing to move their compute. Most will not: a training run sits on a cluster that took a year to fund, and a nightly ETL job is load-bearing and nobody is rewriting it for a receipt.

The Notary detaches the proof from the compute. An external process hashes what went in and what came out and registers the two digests; Sable signs that registration with the *same* receipt key, stores it in the *same* receipts table — so the existing public verifier serves it with no new machinery — and puts the hash of the signed payload on an anchor stream. The honest scope is printed inside the signature itself, so a verifier cannot read the claim without reading its limits:

> **This account registered these exact fingerprints at this time. It does *not* prove the computation happened, that it was performed correctly, or that the fingerprints describe what the caller says they describe. Sable did not run this work, did not observe it, and cannot check any of it.**

Time Machine — replay capsules that cannot become a prompt archive

A capsule records a request so it can be re-run later against the same configuration, which is how you answer "has this regressed?" without keeping a spreadsheet of anecdotes. Recording is never implicit: a capsule exists only because its owner asked for one and confirmed, per capsule, that Sable may hold that request. The request is sealed at rest, nulled the instant the owner deletes the capsule, and swept at a hard TTL.

Two design choices keep this from quietly becoming the thing §03 forbids. The recorded request is **never returned by any endpoint, including to its owner** — a regression store that hands prompts back is a prompt archive with an API in front of it. And the recorded output is **not stored at all**: only a SHA-256 over it. That is exactly sufficient for the question being asked — same or not — and insufficient for reconstructing a completion. The consequence is printed rather than hidden: a diff can prove the output changed and can never show what changed.

Source sets — fingerprints, never bytes

A source set registers the documents a call was given, by fingerprint. Sable takes hashes and never the document bytes, and the set's root is byte-identical to the `context` root a chat receipt already carries when the same documents are declared in the same order — so a reader can tie a citation manifest to the specific inference that used it. Sets are immutable; re-registering a slug mints a new version.

It proves that a document with this exact SHA-256 was declared as source *n* of this named set at this time. It does not prove the model read those documents, used them, or reasoned correctly from them, and it says nothing about whether they are true, authentic, or lawfully held. Labels, names and URIs are the caller's unverified assertions and are **public** to anyone holding the set id.

The Verifiable Arena — a leaderboard made of receipts

Every published benchmark number is structurally an assertion: someone says they ran something and got a figure, and nothing in the artifact distinguishes a measured score from a typed one. The Arena removes the assertion. Each task of each run is a real metered inference through the ordinary chat path, so it is held, capped, guardrailed, metered and individually receipted; the run stores those receipt ids and marks them public, so a skeptic can pull every one and verify each signature. A score that was typed has no receipts to show.

The other half is the declaration: a suite's tasks are published verbatim and hashed canonically, the hash rides on every leaderboard row, and a run records the hash it ran against — so a suite quietly edited under a stable name appears as a hash that no longer matches the runs beneath it. A run scores only if *every* task completed; a model that could not be reached is recorded as a failed run and never scored, because scoring an unreachable model zero would be a false claim about the model rather than a true one about the gateway. A row proves that these exact declared tasks were run against this model at this time and scored this way. It is not a capability claim and not a trust score. **No suite has been declared yet** — the endpoint is live and the board is empty, which is the honest state of a benchmark nobody has run.

Know Your Agent — the bound, not the boast

The Agent Passport is something a person reads. KYA is its machine-checkable sibling, for a reader that is not a person at all: a contract, a desk, or another agent deciding in one call whether to let an autonomous counterparty touch anything. It publishes the caps that

actually bind — the **tightest** cap in the whole delegation chain rather than the key's own nominal one, with the chain attached as evidence — plus the window it resets over, whether a breaker is armed, whether the key is frozen or expired right now, the attached policy and guardrail hashes, whether that agent's *own recent receipts* carried a verified attestation, and run counts obtained by re-folding the receipt hash chains rather than by counting rows.

It is signed with the ordinary receipt key, so it verifies through the existing public endpoint. It carries an expiry, because a credential asserting live configuration has to go stale: configuration changes, and a signature does not. It is **not a trust score, a rating, or a ranking**, and says nothing about whether the agent is competent, honest, or acting in good faith.

Autopilot — a proposal with evidence, never an action

Autopilot reads an account's real metered usage, proposes a cheaper or faster engine configuration, and proves the swap is safe by running the owner's own eval suite against both the current and the proposed spec. It is asked, never ambient: an owner calls it, and it produces nothing at all for an account below its volume and saving thresholds, because a recommendation drawn from fifty requests is noise wearing a percentage. **It never changes anything on its own.** That is the design constraint rather than a caveat: an optimizer that silently rerouted traffic would destroy the proposition the whole gateway rests on, which is that a signed receipt is worth something because nobody moved the thing it describes. Adoption is an explicit owner action that goes through the ordinary engine publish path, so every immutability invariant still holds; rollback needs no machinery, because the previous version was never mutated and can simply be published again.

The analysis may use only figures Sable actually has: real usage events, real catalogue prices with the deployment's own margin applied, and real observed latency *for models this account has itself run in the window*. It never quotes a latency for a model it has not served. There is **no quality model** — Autopilot does not know or imply that one model is as good as another; that question is answered by the owner's evals and by nothing else, and a proposal without evidence is labelled a priced hypothesis.

The Terminal — a public demo that is real, and is currently off

The Terminal is keyless public code execution: type into a box, press run, watch it execute in a real sealed sandbox, get back a real signed receipt verifiable at the public endpoint. It runs the same execution path as the paid route and writes the same metered event; the only difference is that a configured house account pays, behind a global daily budget and a per-IP rate limit. It exists because this repository deleted its previous synthetic demo console on the grounds that a fabricated demo is worse than none on a product whose entire pitch is provability. **It is deliberately not enabled on the production deployment** — the route is not mounted, and a request to it returns 404.

Escrow claims — the gateway half of pay-on-proof

Described with its contract in §11. The gateway side prepares the exact bytes a payee needs to claim on chain, and tells them *before* they spend gas whether their receipt will satisfy the contract's two preconditions. It touches no chain and signs nothing new: the receipt was signed when the work was metered.

11 On-chain verification

Everything Sable signs is already verifiable — but only against a key published at a URL Sable serves. Written, tested, and not deployed.

There is a gap in the receipt story that a hosted verifier cannot close. A receipt verifies against a signer address fetched from `GET /v1/receipts/pubkey`. If the gateway is down, nobody can fetch it. If the gateway is lying, nobody can tell when it started. Putting the signer and the batch roots on a public chain closes both: the statement becomes undeniable and timestamped by a party that is not Sable.

Sable's signatures have always been **secp256k1 / EIP-191** — natively checkable by an EVM, and by very little else Sable had access to. Sable's presence on **Robinhood Chain** — an Arbitrum Orbit (Nitro) EVM L2, **chain id 4663**, settling to Ethereum, and the chain the SABLE token launched on (§06) — is what made an EVM the obvious place to put them. Three contracts are written and tested against that target. To be explicit, because the two facts sit next to each other and a reader could reasonably join them: the SABLE token contract is live on that chain, and **Sable's own three contracts below are not**. Neither depends on the other.

> STATUS, STATED BEFORE THE DESCRIPTION
>
> **Nothing in this section is deployed.** That statement is about *Sable's three Solidity contracts* and nothing else — the SABLE token contract of §06 is live on Robinhood Chain and is not one of these. None of `SableReceiptVerifier`, `SableAnchorRegistry` or `SableEscrow` has an address on Robinhood Chain or on any other network, no key is funded, no transaction has been broadcast, and no batch root has been published to an EVM chain. The deploy script exists so the sequence is written down, not so it gets run by accident, and deploying is a founder decision rather than an engineering one. On the gateway side the EVM anchor backend requires four settings together and is **off**; with none set it is simply absent, and with some but not all set the process refuses to boot rather than degrading into a silent non-anchor. The anchor that actually runs today is the Solana memo worker, and it is the only one that has ever published a root.

The three contracts

CONTRACT	WHAT IT DOES	TRUST SURFACE
`SableReceiptVerifier`	Recovers the signer of a receipt on chain, exactly as the off-chain verifier does. High-`s` signatures are rejected per EIP-2, so one receipt has exactly one valid encoding.	Its owner can rotate the signer, which changes what it answers for future queries.
`SableAnchorRegistry`	Stores the batch Merkle roots the anchor worker publishes, with the block number and timestamp the chain assigns, and verifies membership proofs. First write wins, so a recorded timestamp is always the earliest publication and cannot be walked forward.	Its owner controls the publisher set; a publisher can anchor any root, which is the entire trust model of an anchor and is why the record names the publisher.
`SableEscrow`	Pay-on-proof settlement in three states. A payer locks funds against a job id; the payee releases them by presenting a receipt that recovers to the signer *snapshotted at open time* and literally contains the job id. The two exits — claim before the deadline, refund after — are time-disjoint, so they never race.	**No owner, no pause, no sweep.** Nobody, including the deployer, can move a funded escrow anywhere but to its payee with proof, or back to its payer after the deadline.

Solidity 0.8.28, pinned; EVM target *shanghai* rather than *cancun*, because transient storage depends on the chain's ArbOS version and that is not ours to control, so the reentrancy guard uses plain storage. **Sixty tests pass.**

Cross-language agreement was executed, not asserted

The interesting failure mode here is not a bug inside either implementation; it is the two implementations quietly disagreeing, in which case an on-chain verifier confidently verifies nothing. So the agreement is a test rather than a claim. A receipt signed by the real gateway signer is recovered to the same address in Solidity and releases a real escrow. The escrow's job-id matching rule was run over fifteen vectors in both languages *with a negative control*, so the harness cannot pass vacuously.

> WARNING — THE MERKLE TREE IS SHA-256 OVER HEX TEXT, AND AN ODD NODE IS PROMOTED
>
> This is the most surprising thing in the contracts directory and it is deliberate. The Solidity matches the Rust, not EVM idiom:
>
> ```
> leaf(c)   = sha256( "sable-merkle-leaf-v1" || utf8(c) )   // c = the 64-char lowercase hex commitment
> node(l,r) = sha256( "sable-merkle-node-v1" || l || r )     // l, r = 32 raw bytes
> ```
>
> Three differences from the proof you have probably read: SHA-256 rather than keccak256, because the gateway, the SDKs and the public docs all compute SHA-256 and an idiomatic port would silently verify nothing; leaves are the ASCII hex string hashed as text, not the 32 bytes it encodes; and pairs are ordered with an odd trailing node *promoted unchanged*, never duplicated and never sorted — sorted pairs lose position, and duplicating the tail is the classic construction that lets two different leaf multisets share a root. A test asserts the two hashes differ, so a well-meaning cleanup toward idiom cannot pass.

What an on-chain check proves

Carrying the same discipline the receipts carry, stated rather than left to be inferred.

VERIFIED RECEIPT	ANCHORED ROOT	ESCROW RELEASE
Sable's receipt key signed exactly these bytes, and — combined with the chain — cannot later be denied or backdated. Not that the model ran, that the counts are honest, that a sandbox executed the code its fingerprint claims, or that the output was correct.	This publisher put this root on this chain in this block. Not that the leaves are true, that the batch is complete, or that Sable did not also keep a second private set. Absence of a root is evidence; presence is attribution plus a timestamp.	Sable attested the job was metered, and the payment was conditioned on that attestation. It is worth exactly what the attestation is worth. A buyer who needs correctness wants §08, not this.

Two consequences of the snapshot rule are worth naming because they cut both ways. A signer rotation cannot retroactively change who may spend money already locked, which is the property a payer wants at funding time — and the cost is that escrows opened under a rotated key can no longer be claimed and must wait for their deadline and refund. Two escrows naming the same job id are both releasable by the same receipt; that is the payer's own choice, since they funded both.

And to repeat the rule from §06, because an EVM is exactly where people expect it to be relaxed: **none of this is payable in SABL, emits a token, pays yield, stakes, bonds, or votes.** The escrow escrows native currency or a plain ERC-20. Moving to an EVM makes Sable's signatures machine-checkable; it does not make the network token-secured, and it will not.

12 What is built, and what is not

The most load-bearing section in this document.

Infrastructure whitepapers routinely describe a planned system in the present tense. We think that is the single most corrosive habit in this category, so the tables below separate what the code does today from what it does not — and, for the parts that are built, whether an operator has actually switched them on. A capability that exists in the binary and is switched off is not the same as one a buyer can use, and collapsing the two is how a roadmap gets printed as a product.

BUILT	BUILT · OFF	NOT BUILT	NEVER
Deployed and working on the production gateway with no further configuration.	Deployed, but inert until an operator sets something. Named per row.	Does not exist in the code. Direction at best.	A decision, not a backlog item.

THE CORE — THE GATEWAY A BUYER USES TODAY

CAPABILITY	STATUS	NOTE
OpenAI chat & embeddings, streaming	BUILT	Multi-provider failover; 4xx short-circuits; optional latency-ranked ordering
Anthropic Messages & OpenAI Responses	BUILT	Both translate and delegate to the chat path — not second implementations
Sealed ingress, metadata-only persistence	BUILT	The privacy contract, §03, with its exceptions enumerated there
Signed receipts + public verification	BUILT	secp256k1 / EIP-191; run chains, run proofs, audit packs
Metered sandboxes, streaming output, warm sessions	BUILT	One Sable-operated house node; gVisor; egress denied by default
Node counter-signed receipts (`v:3`)	BUILT	Proves which machine served — not that the work was correct
Credit ledger, pre-auth holds, spend caps, breaker	BUILT	Enforcement is **on** in production: an out-of-credit request is refused
USDT deposits, x402 settlement, seller-side x402	BUILT	Live on Ethereum, Arbitrum One and Solana; sanctions screening is a configurable hook
Delegated sub-keys	BUILT	Monotone clamping, subtree spend, cascading revoke
Remote MCP server, OAuth 2.1 + DCR, MCP gateway	BUILT	The access token *is* a real key, so revoking the key ends the grant
Batches, files, hosted agents, memory, agent runtime	BUILT	Batch lines run through the same handlers, so nothing drifts
TDX attestation, fail-closed routing	BUILT	Third-party enclave; **base-image pin**, so the workload build is not proven — §08
Per-response binding (`response_bound`)	NOT BUILT	**False on every receipt issued today.** Correction in §08 — the handlers do not pass the response bytes the check needs
Double-blind private lane	BUILT	Inference only; pinned routing never fails over

CAPABILITY	STATUS	NOTE
Node registry, scheduler, leases, node agent, vetting gate	BUILT	Serving exactly one Sable-operated house node
Sable Vault — private RWA registry	BUILT	Sealed metadata, hash-chained per-asset ledgers, disclosure proofs, Solana anchoring. Registry infrastructure: issuer-declared entries, no custody, no offers, no legal enforcement of ownership; attested and anchored, *not* zero-knowledge

THE PROOF LAYER (§10) — DEPLOYED, MOSTLY NOT SWITCHED ON

CAPABILITY	STATUS	NOTE
Intelligence Engine — `engine/<slug>`	BUILT	Configuration, versioning and identity. Not fine-tuning; no weights are trained or hosted
Sable Notary	BUILT default $0.001/receipt	Proves registration of fingerprints, never that the work happened. No idempotency key yet: a blind retry mints and bills a second receipt
Time Machine replay capsules	BUILT	Opt-in per capsule; the request is never returned to anyone, the output is kept only as a hash
Source sets	BUILT	Fingerprints, never bytes; labels and URIs are public and unverified
The Verifiable Arena	BUILT · OFF no house account configured	Endpoints live and readable; **zero suites declared, board empty.** Running one needs an operator-configured account and budget
Know Your Agent credential	BUILT	Publishes the tightest cap in the delegation chain. Not a trust score
Autopilot	BUILT caller-initiated	Proposes with evidence, never acts. Produces nothing for an account below its volume thresholds. No quality model — the owner's evals are the evidence
The keyless Terminal	BUILT · OFF deliberately; route returns 404	Real execution, real receipt, house-funded behind a daily budget. Not enabled in production
Escrow claim preparation	BUILT	Prepares and pre-checks claim bytes. Useless until a contract is deployed

ON-CHAIN (§11) — WRITTEN AND TESTED, NOTHING DEPLOYED

CAPABILITY	STATUS	NOTE
`SableReceiptVerifier`, **`SableAnchorRegistry`**, **`SableEscrow`**	WRITTEN · NOT DEPLOYED 60 tests pass	Solidity 0.8.28, Robinhood Chain (Arbitrum Orbit, chain id 4663) as the target. **These three have no address on any network; no key funded; nothing broadcast.** Distinct from the SABLE token contract (§06), which is live on that chain
EVM anchoring from the gateway	BUILT · OFF 4 settings, none set	Needs a deployed registry to anchor to. Roots stay honestly unanchored on this backend
Solana memo anchoring	BUILT	The only anchor backend that has ever published a root
EigenDA blob anchoring	BUILT · OFF disperser URL unset	HTTP disperser-proxy implementation; no confirmation polling
ERC-8004 identity registry	NOT BUILT	The document is ERC-8004-*shaped*; the contract is not deployed and the API says so permanently

NOT BUILT, AND NEVER

CAPABILITY	STATUS	NOTE
Sable-owned enclaves	NOT BUILT	Direction — §13
Confidential *sandboxes* (host cannot read your code)	NOT BUILT	The next major build; needs a TDX host. Tinfoil and Phala already ship host-blind execution on owned hardware — the differentiator would be the control plane around it, not the enclave
Sealed environment delivery to the runner	NOT BUILT	Today `env` reaches the runner as plaintext; on a third-party node that would defeat the point, which is one reason there are none
Third-party operators, earnings, payouts	NOT BUILT	**Zero third-party machines serve Sable traffic today**
Multi-region routing	NOT BUILT	A region pin is honoured only if it matches this deployment's declared region, and refused otherwise
On-chain node registry, staking, slashing	NOT BUILT	Contingent on operators existing at all
Fine-tuning, LoRA hosting, GPU-attached sandboxes	NEVER	An Intelligence Engine build is a configuration, not a trained weight (§10)
Pay for Sable in SABL — optional, burns the token, earns a discount	ROLLING OUT flag off · counsel-gated	Never required; Sable is always priced in dollars and payable in USDT; no access gating, no yield, no revenue share, no governance, no claim — §06
Token-incentivized supply, token-settled compute, token-gated access	NEVER	Unchanged by anything on any chain. Neither SABL nor SABLE is paid to supply, settles compute, or gates access

Region pinning deserves its own sentence, because an earlier version of this document described it incorrectly. A pinned request used to be mapped onto one of four labelled nodes that did not exist, and the caller's chosen region was then stamped onto a signed receipt — a jurisdiction claim backed by nothing. Today a pin is honoured only when it matches the region the deployment actually declares, and refused otherwise. Fewer requests succeed. The ones that do mean something.

The same instinct produced the `response_bound` correction in §08 and the narrower attestation claim beside it. Both make this deployment look weaker than the previous document implied. That is the intended direction of travel for a document like this: the claims that survive contact with measurement are the ones worth printing.

13 Direction: compute for rent

The machines are the commodity. The trust layer over them is not.

The compute behind the API today is rented from providers, and the intent is to keep going in that direction rather than to buy hardware: first machines Sable rents and enrols itself, then vetted third-party operators, and open supply last if ever. The first step is no longer hypothetical: the control machinery a fleet requires — a node registry with hardware descriptors and per-node credentials, heartbeats, an approval gate that refuses to schedule an unvetted machine, and a scheduler that claims a per-job lease transactionally before dialing — is built and serving production sandbox traffic, on exactly one node, which Sable operates. Third-party supply remains gated on demand evidence and a compliance programme, not on code. What Sable would sell is never the machine — it is gateway-authoritative metering, hard budgets, signed receipts, fail-closed attestation, and agent-native payment *over hardware nobody has to trust*.

This ordering is a conclusion, not a preference. The record of the last several years is unambiguous about which parts work. Vetted supply with dollar rails works. Selling *outcomes* rather than raw machines works, and is the only model under which heterogeneous supply has ever held together. What fails, repeatedly and in the same way, is paying for existence rather than for work — which reliably produces fabricated supply — along with points and emissions treadmills, volunteer swarms, verification-first marketplaces that never found buyers, and undifferentiated machine listing, where the market is already won by incumbents.

The gap nobody fills is the one Sable already occupies: signed per-request receipts, fail-closed attestation, prepaid stablecoin and x402, budget-bounded delegated credentials, and now a proof layer that extends the same signature to configuration, provenance, benchmarks and counterparty limits. That is a trust-and-payments layer, and it is what makes commodity compute sellable at a premium — the buyer is paying for the proof, not the silicon.

The sequencing follows from the same evidence. Demand comes before supply, because building the operator half first reproduces the failure mode of every exchange with no sellers. The wedge is *provable* compute rather than *cheap* compute, because at commodity prices nobody buys code execution on a stranger's readable machine — the buyer for whom this is better rather than merely cheaper is the one who needs their code kept from the host, or a portable proof of what ran. Which is also why the operator programme cannot open before sealed environment delivery and confidential sandboxes exist: today a runner host can read the code it is handed, and offering that on a stranger's machine would be selling the thing this paper argues against.

> A RULE WE HOLD OURSELVES TO
>
> Sable will not describe itself as a marketplace, a network of operators, or decentralized infrastructure in the present tense until independent operators are actually serving a meaningful share of paid work. Until then this section is labelled as direction, the node listing reports only things whose existence can be checked, and any page inviting operators says outright that zero third-party machines serve Sable traffic. An exchange with no sellers is the most common artefact in this industry. We would rather ship late than describe one.

14 Threat model & trust boundaries

What is protected now, what is not, and which of those we intend to change.

Defended today

- **In-transit interception.** Payloads are sealed on ingress; plaintext never crosses the network between caller and dispatch.
- **At-rest retention.** No prompt, completion, submitted code, or output stream is written down on the ordinary paths. A subpoena, a breach, or a curious engineer finds metadata. Where a feature must retain content to function at all, §03 enumerates it and the content is sealed under the master key — which is protection against a database dump, not against Sable.
- **Silent substitution and tampering.** A signed receipt lets a client detect any mismatch between what was claimed and what was delivered; a run chain extends that across a whole session, and an anchor extends it across time.
- **Unattested confidential execution.** A confidential request whose backend fails verification is refused, not downgraded — including when the failure is our own misconfiguration, which is how five outages were paid for in availability rather than in false claims.
- **Budget exhaustion by a runaway agent.** Pre-auth holds, subtree-aware spend caps, a spend-velocity breaker that freezes rather than refuses, and cascading revocation bound the blast radius of a sub-agent you no longer control.
- **Double execution and double billing.** Idempotency is enforced by database constraints rather than application memory, at both the billing and the execution layer.
- **Unvetted supply.** A node that has not been approved by an operator is never scheduled, so enrolling a machine does not by itself put it in the path of anyone's code.

Not solved — and how it changes

- **The gateway operator at the egress frame.** On the standard tier, plaintext is momentarily visible to the gateway process. Confidential inference removes this for the model call; moving the shim itself into an enclave Sable owns removes it generally.
- **The sandbox host.** A runner host can read the code it executes and the environment handed to it. This is the honest gap, it is the reason confidential sandboxes are the next major build, and it is why a sandbox on a machine we do not control is not offered at all rather than offered quietly.
- **The upstream model host on the standard tier.** Decryption is required because a third-party provider necessarily sees its input. The confidential tier is the answer; the standard tier does not pretend otherwise, in the product or in the docs.
- **The confidential tier's own residual gaps.** The enclave is a third party's, the pin is on the base image rather than the workload, `gpu_verified` is recorded rather than proven, and per-response binding is not yet true. Each is stated in §08 rather than averaged into a single reassuring word.
- **Sealed content under Sable's key.** Hosted agent code, shared files, mailboxes, replay capsules and Vault metadata are ciphertext at rest, not end-to-end encrypted. An operator with the master key and database access can open them. Making that not-true requires client-held keys and is not built.

- **Availability.** A single gateway remains a concentration of risk, and several subsystems still hold per-process state, so horizontal scale is an engineering prerequisite rather than a configuration change.

We state the current boundary plainly because a confidentiality product that overstates its guarantees is worse than one that makes none: it manufactures confidence that is then acted on. The roadmap is, in effect, the systematic shrinking of this boundary — and every step of it should be checkable from outside, by a buyer who has no reason to take our word for anything.

15 Conclusion

The agent economy will route an enormous volume of machine intent through a small number of compute endpoints. Whether that layer becomes an unaccountable one — where nobody can say what ran, on what, for whom, or how much — is being decided now, by the defaults the infrastructure ships with.

Sable's bet is that the durable position is not the hardware and not the model, but the layer that makes a unit of compute *provable*: metered where the metering can be trusted, bounded before the money moves, attested where the hardware allows it, and signed into a receipt that outlives the request and does not require trusting the party that issued it. Build that, and the machine underneath stops needing to be trusted — which is exactly what makes it replaceable, and exactly why the trust layer is the part worth owning.

Two corrections in this edition make the product look smaller than the last one described. That is the mechanism working rather than failing. A proof layer is only worth anything if its author is willing to publish the measurement that contradicts them — and a document that never gets worse in any respect is not a record of a system, it is marketing with page numbers.

> **We earn trust through architecture, not assurances. Promises don't scale. Verification does.**

A Appendix — API surface

A representative slice of roughly 340 operations. Full reference at buildsable.com/docs, and the machine-readable document at `/openapi.json`.

CORE — INFERENCE AND COMPUTE

METHOD	PATH	PURPOSE
POST	/v1/chat/completions	OpenAI-shape chat; `stream` supported; accepts `sable_privacy_tier`, `sable_region`, `sable_scrub`, `sable_context`, `sable_run_id`
POST	/v1/embeddings	OpenAI-shape embeddings, same sealed path
POST	/v1/messages	Anthropic Messages API; translates and delegates to the chat handler
POST	/v1/responses	OpenAI Responses API; delegates the same way
POST	/v1/sandboxes	Metered code execution; accepts `Idempotency-Key`; optional SSE streaming
POST	/v1/sandboxes/sessions	Persistent warm session: exec, workspace files, snapshots
POST	/v1/mcp	Remote MCP server (JSON-RPC); bearer key or OAuth 2.1 access token

BUDGET, IDENTITY AND SETTLEMENT

POST	/v1/keys/delegate	Mint a bounded child of the calling key — parent-key authenticated
GET	/v1/credit	Key-authenticated balance including holds — an agent's remaining runway
GET	/v1/billing/methods	Treasury address, chains, token contracts, confirmations, enforcement state
GET	/v1/models	Catalog under stable Sable IDs, with per-token pricing at the billed rate

PROOF

GET	/v1/receipts/pubkey	Receipt-signer address & scheme
POST	/v1/receipts/verify	Verify a `{receipt, signature}` — public, no key required
GET	/v1/attestation	Live verified TEE attestation — pre-flight, before you send anything
GET	/v1/runs/:id	An agent run's chained receipts; `/proof` and `/audit` mint verifiable artifacts
POST	/v1/engine/builds	Compile a configuration into a versioned `engine/<slug>` artifact
POST	/v1/notary/receipts	Sign and anchor fingerprints for a computation Sable did not run

POST	`/v1/replay/capsules`	Record a replayable capsule; `/replay` re-runs and compares fingerprints
POST	`/v1/sources/sets`	Register documents by fingerprint; the root matches a receipt's `context` root
GET	`/v1/arena/leaderboard`	Receipted public benchmark board — ordering and trust model ride in the response
GET	`/v1/kya/public/:handle`	Machine-checkable agent credential: tightest caps, policy hashes, verified history
POST	`/v1/autopilot/analyze`	Propose a cheaper configuration with receipted evidence — never acts
GET	`/v1/escrow/claims/:request_id`	Prepare and pre-check the bytes `SableEscrow.claim` takes
GET	`/v1/anchors/:root`	Every public anchor for a root, per backend, with verification instructions

Response headers on every metered call: `x-sable-node`, `x-sable-region`, `x-sable-privacy-tier`, and the receipt triplet `x-sable-receipt` / `-sig` / `-signer`. Streaming responses emit the receipt as a trailing `sable.receipt` event instead, because the headers have already been flushed.

> HOW TO CHECK THIS DOCUMENT AGAINST THE SYSTEM
>
> Nothing here asks to be taken on trust. `GET /openapi.json` lists every mounted route, and a path that is absent returns 404 with an empty body while a mounted one returns a JSON error envelope — which is how the difference between "built" and "switched on" in §12 was established rather than recalled. `GET /v1/status` reports the confidential tier's live posture. `GET /v1/nodes` reports what is actually serving, marking anything that is not a real enrolled machine as synthetic. `GET /v1/billing/methods` reports whether billing enforcement is on and what the SABL rail's status is. If any claim in this paper disagrees with those endpoints, the endpoints are right.