



Sealroom whitepaper

A Solana registry and payment rail where model owners sell per-call inference on vetted confidential-GPU operators, and the weight key is released only to attested, bonded enclaves.

Edition 1 • October 2026

1. Introduction	
2. System overview	
3. Confidential computing foundations	
4. Operator admission and platform pinning	
5. The attestation pipeline	
6. Key management and release	
7. The sealroom program	
8. Payments over x402	
9. Settlement and distribution	
10. Bonds, disputes and arbitration	
11. Services and infrastructure	
12. Security design	
13. Parameters and governance	
14. Economics	
Appendix A. Instruction reference	
Appendix B. Glossary	

1. Introduction

1.1 What Sealroom is

Sealroom is a Solana registry and payment rail where model owners sell per-call inference on vetted confidential-GPU operators, and the weight key is released only to attested, bonded enclaves.

A model owner encrypts its weights once, under a random 256-bit data key, and publishes the ciphertext wherever it likes. The data key stays with the owner. It leaves the owner's key broker only as an HPKE ciphertext addressed to a public key that was generated inside a confidential virtual machine (CVM) and whose attestation is recorded on Solana. That CVM is a Sealroom node: one Intel TDX trust domain with one to eight NVIDIA H100 or H200 GPUs in confidential computing mode, running a measured Sealroom runtime, on a physical server that a vetted operator has registered and bonded in USDC.

Callers reach a node over TLS that terminates inside the enclave and pay for each call over x402 in USDC. Each payment lands in a lane vault, a token account the program owns for one (model, node) pair. Anyone can then call `distribute` on the lane, and one transaction pays the protocol fee into the buyback vault, the serving operator's share and every recipient in the model's revenue split. The buyback vault is a USDC account held by a program address that has no private key; its only outflow is `buyback_swap_and_burn`, one instruction that swaps the fee for \$SEALROOM, the protocol's fixed-supply, fairly launched token, and burns every token it receives (14.7).

Solana holds the registry and the money: operators, platforms, nodes, attestations, runtime allowlists, models with their policies and splits, lanes, bonds and disputes. Weights, data keys, prompts, completions, raw attestation evidence and the GPUs themselves stay off chain. The chain records hashes of everything a verifier or a broker needs to compare against: the operator agreement, the platform identity, each GPU's identity, the runtime measurement, the evidence, and the enclave's public keys.

1.2 The problem it answers

A team with a valuable specialist model has three established ways to sell inference, and each leaves one party exposed.

Option	Owner's position	Buyer's position
Run its own API	Pays for idle GPUs; confidential-computing GPUs are mostly sold reserved or by quote	Must send prompts to the owner
Host on an inference cloud	The host can read the weights unless confidential computing is used; protection is contractual	Must trust both the host and the owner
License the weights for on-premises use	Gives up control of the weights permanently	Must run its own GPUs

Sealroom serves the case of two-sided distrust. The owner will not ship weights to the buyer's infrastructure, and the buyer will not send data to the owner's. Both can accept a third machine when its hardware proves which software it runs and its operator is named and bonded, with access to the weights decided by a public policy the owner wrote.

Between many owners, many operators and many callers, that requires four things, and Sealroom provides each of them:

1. A public, machine-checkable record of which physical machines are attested and which operator is liable for them.
2. Key release gated on that record, by the owner's own broker.
3. Per-call payment without an account, so that software agents can pay as easily as enterprises.
4. Revenue splits to upstream contributors, enforced by the program at distribution time.

1.3 Who uses it

Model owners list models, each with its own key broker, runtime pins, operator requirements, price and split. The primary segment is model vendors selling to regulated buyers, such as banks, hospitals and government agencies, that refuse vendor-hosted APIs. Enterprises licensing internal models to partners and subsidiaries, and specialist fine-tuners in legal, medical and financial domains seeking agent distribution, use the same path.

Operators are vetted datacenter companies running confidential-mode GPUs. Each signs the Operator Agreement, posts a USDC bond, and registers its servers' platform identities and the device identities of their GPUs. An operator earns the Operator-role share of every lane its nodes serve.

Callers are applications, enterprises and AI agents. A caller checks a node's attestation on chain, opens TLS to the enclave, and pays per call over x402, either a fixed price (`exact`) or a metered amount under a ceiling (`upto`).

Contributors are the base-model owners, fine-tuners and data providers named in a model's split. A base model can require a derivative royalty, which every derivative model must carry as a locked split entry paid to the base owner's canonical USDC account.

Verifiers form a committee of five independent parties. Each verifier checks node evidence against Intel and NVIDIA collateral and against the registry, and signs a fixed 200-byte message; three signatures are required for an attestation to land. Sealroom runs at most two of the five verifiers.

The arbitration panel is a named multisig, distinct from the admin multisig, that rules on disputes under the Operator Agreement. It can freeze an operator, revoke every attestation the operator holds, and slash its bond to the claimant.

1.4 Design commitments

The rest of this paper elaborates a small number of fixed decisions.

- **Supply is permissioned.** Operators are vetted off chain, then admitted on chain by the admin multisig with an agreement hash and a tier. Only admitted, bonded operators can register platforms and nodes.
- **Platforms are pinned.** A quote counts only if its platform identity and the identities of the GPUs that produced its companion evidence were registered by that operator for that node.
- **Owners hold their keys.** The default key broker is run by the owner, reads Solana directly, re-verifies the evidence itself, and publishes an audit record for every release and every denial.
- **Policy is public.** Runtime allowlists, tier and bond minimums, operator allowlists, attestation expiry and revocation state are on-chain accounts that any party can read.
- **Payment needs no account.** A node answers HTTP 402 with x402 payment terms; the caller signs a USDC transfer to the lane vault and retries.
- **Splits are settled at distribution.** Each call is one transfer to one vault; `distribute` later allocates the vault across the protocol fee and the split in one atomic transaction.
- **Money is USDC.** Prices, bonds, fees, deposits and payouts are USDC base units moved with checked integer arithmetic.

1.5 What this paper covers

Chapter 2 gives the system overview: actors, the lifecycle, trust boundaries, the division between chain and off-chain services, and the reasons for Solana. Chapter 3 covers Intel TDX, NVIDIA Hopper confidential computing, the device identities Sealroom pins, and the construction that binds a node's fresh keys and a recent slot hash into both pieces of hardware evidence.

Chapter 4 describes operator admission and platform pinning, Chapter 5 the attestation pipeline, and Chapter 6 key management and release. Chapter 7 documents the `sealroom` program: accounts, PDAs, instructions, errors, timelocks and upgrade authority. Chapters 8 to 10 cover payments over x402, settlement and distribution, and bonds, disputes and arbitration. Chapters 11 to 14 cover services and infrastructure, security design, parameters and governance, and economics. Appendix A is an instruction reference and Appendix B a glossary.

1.6 Conventions

Amounts are USDC base units (1 USDC = 1,000,000 base units). Hashes are SHA-256 and 32 bytes unless stated otherwise; runtime measurements are 48 bytes. On chain, integers are little-endian, public keys are 32 raw bytes, and every account begins with an 8-byte Anchor discriminator followed by a `version: u8` and a `bump: u8`. A program-derived address is written with its seeds, for example `["att", node]`. The symbol `||` (written `||` in formulas) denotes byte concatenation. In

the API, keys and signatures are base58, hashes are lowercase hex, and amounts and slots are decimal strings.

2. System overview

2.1 Actors and roles

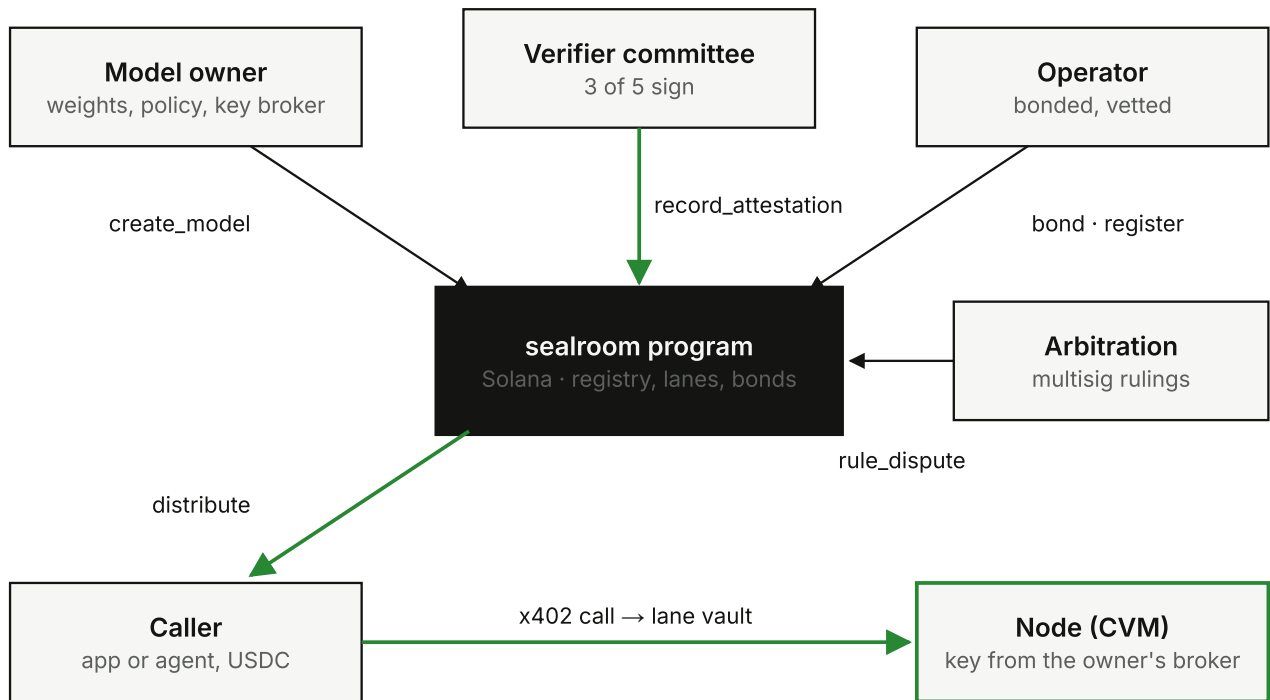


Figure: The seven actors of Sealroom and their relationships to the `sealroom` program.

Actor	Role	On-chain identity	Signs
Model owner	Encrypts weights, publishes the ciphertext, registers a <code>Model</code> with policy, price and split; runs or delegates a key broker	<code>Model ["model", owner, id]</code>	<code>create_model</code> , <code>update_price</code> , <code>update_policy</code> , <code>propose_split</code> , <code>cancel_split</code> , <code>open_dispute</code> , <code>claim_dispute</code>
Operator	Vetted datacenter company running CC-mode GPUs; signs the Operator Agreement, posts a USDC bond, registers servers and GPUs	<code>Operator ["op", authority]</code>	<code>deposit_bond</code> , <code>request_unbond</code> , <code>withdraw_bond</code> , <code>register_platform</code> (with the admin), <code>register_node</code>
Node	One CVM: an Intel TDX trust domain with 1 to 8 GPUs, running the measured Sealroom runtime	<code>Node ["node", platform, index]</code> , <code>Attestation ["att", node]</code>	Enclave keys only: TLS sessions and usage vouchers

Actor	Role	On-chain identity	Signs
Verifier committee	Five independent parties; three must check the evidence and sign	<code>VerifierSet</code> <code>["verifiers", epoch]</code>	Ed25519 signatures over the 200-byte attestation message; <code>revoke_attestation</code>
Caller	Application, enterprise or agent paying per call over x402 in USDC	Its wallet	Its own USDC transfer to the lane vault
Contributors	Base-model owners, fine-tuners and data providers listed in a model's split	Split entries in <code>Model</code> and <code>Lane</code>	Consent to lower or remove a locked entry; claims of deferred payouts
Arbitration panel	Named multisig that rules on disputes and slashes bonds under the Operator Agreement	<code>Config.arbitration</code>	<code>rule_dispute</code> , <code>arbitrate_operator</code> , <code>clear_attestation_revocation</code>

Alongside the actors, a set of services operates the system. Each runs as its own process with its own credentials.

Service	Function
Admin multisig	Squads multisig in <code>Config.admin</code> : admits, tiers and freezes operators, co-signs platforms, pauses, allows and revokes runtimes, proposes fee changes, names the buyback authority and selects the buyback mint once
Key broker	Owner-run: reads the chain, re-verifies evidence, wraps the data key, audits every decision
Verifiers	One process per committee member, each with one Ed25519 key and its own collateral cache
Coordinator	Collects committee signatures and submits <code>record_attestation</code> ; holds only a fee-payer key outside the committee
Facilitator	Verifies and settles x402 payments; co-signs only as fee payer
Indexer and API	Mirror accounts and events into PostgreSQL and serve the public registry API under <code>/v1</code>
Keeper	Applies matured splits, activates lanes, reconciles expired <code>upto</code> sessions, delivers deferred protocol fees, and, as the buyback authority, starts <code>buyback_swap_and_burn</code>

2.2 The lifecycle



Figure: List, Onboard, Attest, Release, Serve and pay, Distribute and Dispute, with the instruction or service call at each step.

A model is served by a node through seven steps. The first two are registrations that happen once. Attestation repeats at least every 24 hours for each node. Release happens when a node loads a model. Serving, payment and distribution run continuously. Dispute is invoked when an owner has evidence against an operator.

2.2.1 List

1. The owner generates a random 256-bit data key, encrypts the weights with AES-256-GCM in chunks, and uploads the ciphertext to any storage, public storage included.
2. The owner signs `create_model`, which creates `Model ["model", owner, id]` (2,348 bytes) with the 32-byte manifest hash, the key broker's public key and URL (at most 128 bytes), up to 8 allowed runtime measurements of 48 bytes, `min_tier` (1 to 3), `min_bond`, an operator allowlist of up to 16 operators (empty admits any operator meeting tier and bond), a price (`Exact { per_call }` or `Upto { input_per_million_tokens, output_per_million_tokens, max_per_call }`), `derivative_royalty_bps`, an optional base model, and a split of 1 to 8 entries summing to exactly 10,000 bps. Each entry names a recipient token account, a beneficiary wallet, `bps`, `locked` and a role: `Owner`, `Operator`, `BaseModel` or `Contributor`.
3. The program checks every split recipient is a canonical USDC account of its declared owner and, if the base model sets a royalty, requires a locked `BaseModel` entry of at least that many bps paid to the base owner.

Later split changes go through `propose_split` and become applicable after 7 days (604,800 seconds); a locked entry can be lowered or removed only with its beneficiary's signature. The price and the policy change at the owner's signature alone: `update_price` replaces the price, and `update_policy` replaces the allowed runtimes, `min_tier`, `min_bond` and the operator allowlist. Both take effect in the transaction that carries them and leave the split and every lane balance unchanged.

2.2.2 Onboard

1. Sealroom vets the operator off chain: legal entity, datacenter certifications, and the signed Operator Agreement.
2. The admin multisig signs `register_operator(agreement_hash, tier)`, creating `Operator ["op", authority]` with its bond vault and payout account.

3. The operator signs `deposit_bond`. An operator is healthy while it is unfrozen and its bond is at least its tier's minimum in `Config.tier_bond_minimums`.
4. For each server, the operator and the admin multisig both sign `register_platform(platform_id_hash, gpu_ueid_hashes)`, creating `Platform ["plat", operator, platform_id_hash]` with up to 8 GPU UEID hashes, a global `PlatformPin`, and one global `GpuPin` per GPU.
5. For each CVM, the operator signs `register_node` with an index (`u16`), a node key, an HTTPS endpoint of at most 128 bytes, the GPU model (`H100` or `H200`) and the UEID hashes of the GPUs it claims; each claimed GPU must be on the platform and unclaimed.

Runtime images are admitted separately: `allow_runtime`, signed by the admin multisig, creates `Runtime ["rt", SHA-256(measurement)]`, which becomes effective after a 48-hour timelock (172,800 seconds). `revoke_runtime` takes effect immediately.

2.2.3 Attest

1. The node boots the measured runtime in a TDX trust domain with its GPUs in confidential computing mode.
2. Inside the CVM it generates three key pairs: an X-Wing HPKE key pair (ML-KEM-768 + X25519), an Ed25519 TLS key and an Ed25519 voucher key.
3. It reads a recent `(slot, hash)` pair from Solana's SlotHashes sysvar.
4. It computes $H = \text{SHA-256}(\text{public keys} \parallel \text{recent slot hash})$, places H in TDX REPORTDATA, and uses H as the attestation nonce for every GPU (Chapter 3 gives the exact construction).
5. It assembles a `sealroom-evidence-v1` object with the node address, the slot pair, the three public keys, the TD quote and one GPU report per GPU, and submits it to the coordinator.
6. The coordinator proposes the next revision (`sequence + 1`, the active verifier epoch, the operator's attestation epoch, `expires_at = \text{chain time} + 86,400 - 1 \text{ seconds}`) to the five verifiers. Each independently verifies the TD quote against Intel collateral and the GPU evidence against NVIDIA RIMs and OCSP, checks the registry, archives the evidence by its hash, and signs the 200-byte canonical message.
7. The coordinator checks each signature against its member key and submits one v0 transaction: an Ed25519 precompile instruction with three signatures, then `record_attestation`, which writes `Attestation ["att", node]`. The transaction is 1,056 bytes, under the 1,232-byte packet limit.

2.2.4 Release

1. The node posts a key request to `Model.key_broker_url`: a request UUID, the model address, and the evidence object that was attested.
2. The broker reads the model, node, platform, operator, attestation, runtime, verifier set and pin accounts directly from RPC in one context, and runs the eligibility rules: attestation current, unexpired and unrevoked; runtime allowed and pinned by the model; platform registered; tier and bond at the required minimums; operator unfrozen and allowlisted; TCB `UpToDate`; node enabled; protocol running.
3. It checks that the evidence hash, enclave key hash and slot pair equal those in the `Attestation` account, re-verifies the vendor evidence itself, and repeats the registry checks.

4. It wraps the 32-byte data key with HPKE (X-Wing, HKDF-SHA256, AES-256-GCM) to the enclave's HPKE public key, with associated data that binds the chain, program, model, node, attestation revision, evidence hash and request.
5. It reads the chain a final time, persists the audit record, and returns the wrapped key.

Inside the CVM the key client unwraps the data key, decrypts the weights in CVM memory and loads them into GPU protected memory. Only the CVM holding the HPKE private key can unwrap it.

Chapter 6 gives the full protocol.

2.2.5 Serve and pay

1. The caller's SDK reads the node's `Attestation` account and the model's eligibility, recomputes the enclave key hash from the node's public keys, and checks that the node's TLS key is the committed one, so prompts terminate inside the enclave.
2. Without a payment, the node answers HTTP 402 with x402 version 2 `PAYMENT-REQUIRED` terms: scheme, USDC mint, the lane vault as `payTo`, amount, a 120-second timeout, fee payer, invoice memo and session identifier.
3. The caller signs its own transaction (compute-budget instructions, one `TransferChecked` of USDC to the payee, the memo) and retries with the `PAYMENT-SIGNATURE` header.
4. The facilitator verifies the authorization without moving funds; the node runs inference and checks eligibility again; the facilitator co-signs as fee payer and submits; the node returns the completion.

With `exact`, the amount is `per_call`. With `upto`, the caller escrows `max_per_call` in the payment-channels program, and the node settles actual usage with a voucher signed by the enclave's voucher key, binding the domain `sealroom:usage:v1`, the chain, lane, channel, request, nonce, cumulative amount, expiry and attestation revision. Settlement and refund of the unused ceiling form one atomic transaction. The metered charge is:

```
charge = ceil((input_tokens * input_per_million_tokens + output_tokens *
output_per_million_tokens) / 1000000)
charge <= max_per_call
```

Because the voucher key exists only inside the attested enclave, the operator cannot raise a charge without breaking the enclave. The runtime stops serving when its own attestation lapses.

2.2.6 Distribute

Anyone can call `distribute(lane)`. With `G` the vault's unreserved balance:

```
G = vault.amount - lane.reserved_amount
fee = floor(G * protocol_fee_bps / 10000)
share_i = floor((G - fee) * bps_i / 10000)
share_last = share_last + ((G - fee) - sum(share_i))
```

The fee goes to the buyback vault, the canonical USDC account of the program PDA `["protocol"]`. The `operator`-role weight is paid to the payout account of the operator whose node owns the lane, so one model can serve on several operators and each receives its own share. Every other entry is paid to its declared account; a share whose account cannot receive it, the fee included, is reserved in the lane as a claimable entitlement. With the 500 bps launch fee, each 1,000,000 base units of revenue yields 50,000 to the buyback vault and 950,000 to the split. The keeper then spends the vault's USDC through `buyback_swap_and_burn`, which swaps it for \$SEALROOM through Jupiter and burns the whole output in the same instruction.

2.2.7 Dispute

1. A model owner with evidence of leaked weights or attestation fraud signs `open_dispute` against an operator, with the evidence hash and the configured deposit. The owner must hold a lane on that operator whose unconsumed protocol fees, actually delivered into the buyback vault, cover the deposit.
2. The arbitration multisig signs `rule_dispute`. A ruling for the claimant can slash up to the tracked bond principal, freeze the operator, and advance `Operator.attestation_epoch`, which invalidates every attestation of that operator at once. A dismissal forfeits the deposit.
3. `settle_slash` funds the reserved slash, and the claimant signs `claim_dispute` to receive it.

An operator with an open dispute cannot withdraw its bond. `arbitrate_operator` lets the panel freeze or revoke an operator without a claimant.

2.3 Trust boundaries

Sealroom's components sit in six domains, each with its own material and its own enforcement.

The enclave. A node's trust domain and its GPUs in confidential computing mode hold the plaintext weights, the data key, the three enclave private keys, and every prompt and completion. The operator's host schedules the CVM and carries its encrypted traffic without access to its memory. The data key enters HPKE-wrapped to a key generated inside the enclave, and prompts enter over TLS terminated inside it. The runtime's egress allowlist permits only the key broker, the Solana RPC and the facilitator, and the runtime runs no owner-supplied code (`trust_remote_code` is disabled).

The physical platform. A node counts only on a server whose Intel platform identity and GPU identities are pinned to a vetted, bonded operator. The Operator Agreement, recorded as `Operator.agreement_hash`, covers the facility, and the USDC bond backs the operator's liability. Evidence from genuine hardware that no operator registered fails the pin checks.

The owner. The key broker holds the data key at rest and releases it only HPKE-wrapped, after its own chain reads and its own vendor verification. Each release and denial is public in `/v1/key-releases` with the attestation account and revision, the evidence hash, a policy hash, the chain read slot and the broker's public key.

The committee. Five verifiers each hold one Ed25519 key. The program accepts an attestation only with signatures from at least three distinct members of the active `VerifierSet` over a message that

commits to the program, the genesis hash, the node, the epoch, the replay sequence, the attestation content and the recent slot. The coordinator that relays the signatures holds no verifier key.

The chain. The program enforces registration, pinning, timelocks, expiry, revocation, payment destinations and settlement arithmetic. The admin and arbitration multisigs are separate Squads multisigs that must differ. Runtime allowance, fee changes and dispute-configuration changes wait 48 hours, and an owner's own runtime pins and operator allowlist apply on top of the protocol's. Protocol fees sit in a vault for which only the program signs, and they leave it only inside the instruction that swaps them and burns the output.

The caller. The caller signs only its own transfer to the lane vault, verifies the enclave's TLS key against the chain before sending data, and, under `upto`, accepts only vouchers signed by the attested voucher key.

Two vendor roots anchor the enclave domain: Intel, through its PCK certificate hierarchy and the Provisioning Certification Service collateral, and NVIDIA, through its device certificate hierarchy, Reference Integrity Manifests and OCSP. Each verifier keeps its own collateral cache, and running nodes continue to serve on their current attestation while collateral is refreshed.

2.4 On chain and off chain

The program stores identity, policy, attestation results and money. The table lists its accounts with their seeds and maximum allocated sizes.

Account	PDA seeds	Bytes	Holds
Config	config	676	Admin and arbitration multisigs, USDC mint, genesis hash, protocol fee, tier bond minimums, durations, active verifier epoch, pause flag, buyback mint and authority, USDC spent and tokens burned by the buyback
VerifierSet	verifiers, epoch	264	Room for 7 Ed25519 keys; the program requires 5 distinct members and threshold 3
Operator	op, authority	193	Agreement hash, tier, bond vault and amount, unbond deadline, frozen flag, payout account, attestation epoch
Platform	plat, operator, platform id hash	343	Platform identity hash, up to 8 GPU UEID hashes
PlatformPin	platform_pin, platform id hash	106	The one platform and operator holding this identity
GpuPin	gpu_pin, GPU UEID hash	107	The platform and the one node holding this GPU
Node	node, platform, index	503	Node key, endpoint, GPU model, count and UEID hashes, status

Account	PDA seeds	Bytes	Holds
Attestation	att , node	384	Measurement, RIM hash, evidence hash, enclave key hash, TCB status, slot pair, expiry, method, signer bitmap, revocation
Runtime	rt , SHA-256(measurement)	256	48-byte measurement, image name and version, source commit, effective time, revocation
Model	model , owner, id	2,348	Manifest hash, key broker, policy, price, royalty, split and pending split
Lane	lane , model, node	719	Vault, totals in, allocated and distributed, reserved amount, active split and revision
Claim	claim , lane, recipient account	122	A deferred payout and its beneficiary
Dispute	dispute , operator, sequence	415	Claimant, evidence hash, deposit, status, ruling and slash
ClaimantDisputes	claimant_disputes , claimant, operator	78	Open-dispute counter for one claimant against one operator

USDC itself sits in program-controlled token accounts: one bond vault per operator, one vault per lane, deposit vaults for disputes, and the buyback vault of the protocol PDA ["protocol"] , which receives every protocol fee.

Off chain, the system holds everything that is large, secret or both:

Material	Location	On-chain anchor
Encrypted weights	Any storage the owner chooses	Model.manifest_hash
Data key	The owner's key store, read by the key broker	Model.key_broker
TD quote and GPU reports	Each verifier's private evidence archive, keyed by evidence hash	Attestation.evidence_hash
Enclave public keys	The evidence object; the registry API publishes their hashes and the voucher key once they reproduce the chain value	Attestation.enclave_key_hash
Operator Agreement and vetting records	Sealroom and the operator	Operator.agreement_hash , Platform.platform_id_hash
Prompts and completions	The enclave and the caller	None
Key-release decisions	The broker's audit log, public through /v1/key-releases	Attestation account and signed revision
Payment sessions and vouchers	The facilitator and node	The settled transfer to the lane vault

Raw quotes stay off chain. A TD quote carries the full PCK certificate chain in its certification data, and the GPU reports carry their own certificate chains; the chain stores one 32-byte hash of the canonical evidence object and the derived values every party compares against.

2.5 Why Solana

Agent payments run on Solana. On January 20, 2026, x402 payments on Solana passed those on Base, 518,400 against 505,000. By mid-August 2026, Solana carried about 70% of monthly x402 volume and had processed more than 35 million x402 transactions in total.

Metered billing exists natively. Coinbase announced the x402 `upto` scheme in April 2026. On Solana, a caller escrows a ceiling in the Solana Foundation payment-channels program (`CHNLxYvVA28MJP9PrFuDXccuoGXAx7jBacfLEkahyGsX`, audited by Cantina in July 2026), and the server settles actual usage with an Ed25519-signed voucher. This maps directly onto per-token LLM billing; in Sealroom the voucher is signed by the enclave's voucher key.

State updates are cheap. Every node re-attests at least once a day, and lanes are distributed continuously, each transaction costing a fraction of a cent. A three-signature `record_attestation` uses 49,314 of the 1.4 million compute units a transaction may spend, in 1,056 bytes; a distribution to the buyback vault and eight split recipients is a 308-byte transaction. Both fit in one 1,232-byte packet.

Committee signatures verify natively. The Ed25519 precompile verifies the verifiers' signatures in the runtime. The program reads the precompile instructions through the `instructions` sysvar, requires every offset to point inside the verified instruction, and matches the signed message byte for byte against the message it derives itself. No signature arithmetic runs in the program.

The chain supplies its own freshness beacon. The `SlotHashes` sysvar exposes the hashes of the most recent 512 slots. A node binds one of them into its hardware evidence, and the program accepts the attestation only while that pair is still in the sysvar.

USDC is native. Bonds, prices, vaults and payouts are SPL Token USDC moved with `TransferChecked`. SVM x402 facilitators validate a fixed instruction layout around one `TransferChecked` to the payee; Sealroom keeps that layout by making each call a single transfer to the lane vault and moving the split into `distribute`, which stays atomic across all recipients.

Proof-based attestation fits the compute budget. The `Attestation.method` field distinguishes `Committee` from `Zk`. Groth16 verification over the `alt_bn128` syscalls takes under 200,000 compute units in the `groth16-solana` crate and about 280,000 in `sp1-solana`, inside the 1.4 million limit, so a proof whose public inputs commit to the same attestation fields records into the same account.

3. Confidential computing foundations

3.1 The confidential node

A Sealroom node is a confidential virtual machine built from two hardware mechanisms. The CPU side is an Intel TDX trust domain, which isolates the VM's memory and CPU state from the host. The GPU side is one to eight NVIDIA Hopper GPUs (H100 or H200) in confidential computing mode, passed through to that trust domain. The runtime image is reproducibly built on Confidential Containers and Kata, and contains the attestation agent, the key client, vLLM, the x402 middleware for `exact` and `upto`, the voucher signer, TLS termination and the egress allowlist.

Each mechanism produces its own signed evidence. Intel's evidence is a TD quote, signed under a key that Intel certifies for one specific platform. NVIDIA's evidence is a GPU attestation report, signed by a key that NVIDIA certifies for one specific GPU. Sealroom ties both to one node, one boot and one set of enclave keys by writing the same 32-byte value into each: a hash over the enclave's public keys and a recent Solana slot hash. The following sections describe each mechanism, the identities Sealroom pins, and that binding.

3.2 Intel TDX

3.2.1 Trust domains

A trust domain (TD) is a virtual machine whose private memory and register state are protected by the processor and by the Intel TDX module. The TDX module is Intel-signed software that runs in the processor's Secure Arbitration Mode (SEAM) and mediates every transition between the host's virtual machine monitor and a TD. TD private memory is encrypted with a key specific to that TD, and the TDX module prevents the host from reading it or remapping it. The host still schedules the TD and carries its I/O, so everything a node exchanges with the outside is either public or encrypted end to end. The TDX module is itself measured: its measurement (MRSEAM), its signer's measurement (MRSIGNERSEAM) and its security version numbers (TEE_TCB_SVN) appear in every quote.

3.2.2 Measurement registers

A TD has one build-time measurement register and four runtime registers, each 48 bytes (the size of a SHA-384 digest).

- **MRTD** measures the initial contents of the TD: the memory pages the host adds and measures before the TD starts, which hold the virtual firmware. It is fixed once the TD is finalized.

- **RTMR0 to RTMR3** are extended by software inside the TD as it boots. By convention RTMR0 records the firmware's configuration, RTMR1 the OS loader and kernel, RTMR2 the kernel command line and initial file system, and RTMR3 is available to the application.

An extend operation folds a new digest into a register; a register can only be extended, never set:

$$\text{RTMR}[i]' = \text{SHA-384}(\text{RTMR}[i] \parallel \text{digest})$$

A reproducibly built runtime image therefore produces the same register values on every compliant platform. Sealroom records the runtime as one 48-byte measurement, the value the verifier's Intel adapter reports for the quote's measured image. That value is the key of the `Runtime` allowlist account (`["rt", SHA-256(measurement)]`), the content of `Attestation.runtime_measurement`, and an entry in each model's `allowed_runtimes`.

3.2.3 TDREPORT and REPORTDATA

Software inside a TD asks the TDX module for a TDREPORT with the `TDG.MR.REPORT` call, passing 64 bytes of its own choice: REPORTDATA. The TDX module returns a 1,024-byte TDREPORT containing the TD's measurements, attributes and REPORTDATA, authenticated with a MAC that only the same platform can check.

REPORTDATA is the channel through which a TD commits to data it generated itself. Whatever the TD writes there is carried unchanged into the quote and covered by the quote's signature. Sealroom writes H, the binding value defined in Section 3.5, into the first 32 bytes.

3.2.4 The TD quote

A TDREPORT is local to one machine. To make it verifiable anywhere, the host passes it to the TD Quoting Enclave (QE), an Intel-signed SGX enclave on the same platform. The QE verifies the TDREPORT's MAC, then signs the report contents with its ECDSA P-256 attestation key. The attestation key is in turn certified by the platform's Provisioning Certification Key (PCK): the QE's own SGX report, whose REPORTDATA holds SHA-256 of the attestation public key and the QE authentication data, is signed with the PCK. The PCK certificate chains to the Intel SGX Root CA.

A version 4 quote is laid out as follows; version 5 adds a type-and-size descriptor before the body.

TD quote v4		
Header	48 bytes	version, attestation key type (ECDSA P-256), TEE type (TDX), QE vendor ID, user data
TD quote body	584 bytes	see below
Signature data length	4 bytes	
Signature data	variable	
ECDSA signature	64 bytes	over header body, by the attestation key
ECDSA attestation key	64 bytes	P-256 public key
Certification data	variable	
QE report	384 bytes	SGX report body of the Quoting Enclave
QE report signature	64 bytes	by the PCK
QE authentication data	variable	
PCK certificate chain	variable	PEM: PCK leaf, intermediate CA, root CA

The 584-byte body carries every value a verifier compares:

Offset	Bytes	Field
0	16	TEE_TCB_SVN
16	48	MRSEAM
64	48	MRSIGNERSEAM
112	8	SEAMATTRIBUTES
120	8	TDATTRIBUTES
128	8	XFAM
136	48	MRTD
184	48	MRCONFIGID
232	48	MROWNER
280	48	MROWNERCONFIG
328	48	RTMR0
376	48	RTMR1
424	48	RTMR2
472	48	RTMR3
520	64	REPORTDATA

Because the PCK certificate chain travels inside the quote, the quote identifies its platform by itself: the PCK leaf certificate carries Intel's platform extensions, including the Platform Provisioning ID (PPID), the FMSPC that identifies the platform model, and the TCB levels the certificate was issued for.

3.2.5 Collateral and TCB status

A quote's signatures prove origin. Whether the platform's firmware and microcode are current is decided against collateral that Intel publishes through its Provisioning Certification Service (PCS): the TCB Info for the platform's FMSPC, the QE Identity, the PCK certificate revocation list and the root CA revocation list, each signed under the Intel SGX Root CA. Quote verification with Intel's DCAP libraries, or through Intel Trust Authority, proceeds in this order:

1. Validate the PCK certificate chain to the Intel SGX Root CA and check both revocation lists.
2. Verify the QE report signature with the PCK public key, check the QE report against the QE Identity, and check that the QE report's REPORTDATA commits to the attestation key and authentication data.
3. Verify the quote signature over the header and body with the attestation key.
4. Compare the PCK certificate's TCB components and the body's TEE_TCB_SVN against the TCB Info levels, and derive the platform's TCB status.

Sealroom carries the outcome as the `TcbStatus` enum (`UpToDate` , `OutOfDate` , `ConfigurationNeeded` , `Revoked` , `Unknown`). The verifier's claims type admits only `UpToDate` , and `record_attestation` rejects any other value with `InvalidTcb` .

3.3 NVIDIA Hopper confidential computing

3.3.1 Confidential computing mode

An H100 or H200 placed in confidential computing mode partitions its memory so that a protected region is reachable only by the GPU's own engines and by the confidential VM it is assigned to. The host cannot read protected memory or inspect the GPU's internal state through debug paths. The GPU driver inside the CVM and the GPU establish an authenticated session using SPDМ (the DMTF Security Protocol and Data Model), and all data moving between CVM memory and the GPU passes through bounce buffers in shared memory, encrypted and authenticated with AES-GCM under that session's keys. Model weights decrypted inside the CVM are copied into GPU protected memory over that channel.

3.3.2 The GPU attestation report

Each GPU holds a device identity key pair provisioned by NVIDIA, with a certificate chain that leads to NVIDIA's device identity root. The attestation report is an SPDМ measurement exchange:

- **Request.** The requester, here the attestation agent in the CVM, sends a 32-byte nonce.
- **Response.** The GPU returns its measurement record (indexed measurement blocks covering firmware, VBIOS and security configuration), opaque data including the driver and VBIOS versions, and a signature by the device's attestation key over the whole exchange, the requester's nonce included.

The report is accompanied by the GPU's certificate chain. The device certificate carries the GPU's unique entity identifier (UEID), and the verification result reports it.

3.3.3 RIMs and OCSP

NVIDIA publishes Reference Integrity Manifests (RIMs) for each driver and VBIOS release: signed documents listing the expected value of every measurement index. A verifier using NVIDIA's local verifier proceeds as follows:

1. Validate the GPU certificate chain to NVIDIA's root and check the status of every certificate with NVIDIA's OCSP service.
2. Verify the report signature with the device certificate's key.
3. Check that the nonce in the report equals the expected nonce.
4. Fetch the driver RIM and the VBIOS RIM matching the versions in the report from NVIDIA's RIM service, verify their signatures, and check their signing certificates by OCSP.
5. Compare every measurement in the report with the corresponding RIM value.

The NVIDIA Remote Attestation Service (NRAS) performs the same evaluation as a hosted service, and a verifier may use it as an optional cross-check. Sealroom's verifier reduces the result to a 32-byte `gpu_rim_hash` identifying the reference set the GPUs were checked against; all GPUs of one node must report under the same RIM hash and with distinct UEIDs.

3.4 Device identities Sealroom pins

Genuine evidence proves that a genuine Intel platform and genuine NVIDIA GPUs produced it. Sealroom additionally requires that they are the specific machines a vetted operator registered. Two identities make that possible, both readable from the evidence itself.

Intel platform identity. The PPID is the identifier Intel uses to provision a platform's PCK, and the PCK certificate that carries it is embedded in every quote that platform produces. Sealroom stores a 32-byte hash of this identity (`Platform.platform_id_hash` , identity kind `intel_ppid_pck` in the API). During onboarding the operator supplies it from its vetting record; at attestation the verifier derives it from the quote.

GPU identity. Each GPU's UEID comes from its NVIDIA device certificate and appears in its verified evidence. Sealroom stores a 32-byte hash per GPU.

The registry holds these identities in four accounts:

Account	Content	Rule
Platform ["plat", operator, platform_id_hash]	platform_id_hash , gpu_ueid_hashes (1 to 8, distinct), registered	Created by <code>register_platform</code> with both the admin and the operator signing, while the operator is healthy
PlatformPin ["platform_pin", platform_id_hash]	platform , operator , platform_id_hash	Global: one platform identity maps to one operator and one Platform account
GpuPin ["gpu_pin", gpu_ueid_hash]	platform , gpu_ueid_hash , node	Global: one GPU belongs to one platform, and <code>register_node</code> sets <code>node</code> once

Account	Content	Rule
Node ["node", platform, index]	gpu_ueid_hashes , gpu_count , gpu_model	Every assigned GPU must be registered on the platform and unclaimed by any other node

Because the pins are global accounts derived from the identity hashes alone, a platform identity cannot be registered under two operators and a GPU cannot be assigned to two nodes. Evidence from an unregistered machine, from a registered machine attributed to the wrong operator, or from GPUs borrowed from another server fails these comparisons before any signature is produced.

3.5 Binding enclave keys and a recent slot hash

3.5.1 The enclave keys

On every boot, the runtime generates three key pairs inside the CVM. Their private halves never leave it.

Key	Algorithm	Public size	Use
HPKE key	X-Wing (ML-KEM-768 + X25519), suite x-wing-draft06-HKDF-SHA256-AES256GCM	1,216 bytes	Receives the data key from the owner's broker
TLS key	Ed25519	32 bytes	Terminates callers' TLS sessions inside the enclave
Voucher key	Ed25519	32 bytes	Signs upto usage vouchers

The X-Wing public key is the ML-KEM-768 public key (1,184 bytes) followed by the X25519 public key (32 bytes).

3.5.2 The public-key transcript

The three public keys are concatenated in a fixed order. Fixed order and fixed sizes make the encoding unambiguous: no two different key sets produce the same byte string.

```
public-key transcript (1,280 bytes)
  offset    0    1,184 bytes  ML-KEM-768 public key  } X-Wing HPKE public key
  offset 1,184     32 bytes  X25519 public key      } (1,216 bytes)
  offset 1,216     32 bytes  TLS Ed25519 public key
  offset 1,248     32 bytes  voucher Ed25519 public key
```

Its hash is the enclave key hash recorded on chain:

```
enclave_key_hash = SHA-256(hpke_public_key || tls_public_key ||
  voucher_public_key)
```

3.5.3 The recent slot hash

Solana's SlotHashes sysvar holds the most recent slot hashes as a list: an 8-byte little-endian count, followed by up to 512 entries of 40 bytes each, an 8-byte slot number and its 32-byte hash. The node reads one `(recent_slot, recent_slot_hash)` pair before it requests any evidence. A slot hash cannot be predicted before that slot exists, so evidence that commits to it was produced after it.

3.5.4 H in REPORTDATA and the GPU nonce

The binding value appends the slot hash to the transcript:

```
H = SHA-256(hpke_public_key || tls_public_key || voucher_public_key ||
recent_slot_hash)
```

The input is 1,312 bytes and H is 32 bytes. The node writes H into both pieces of evidence:

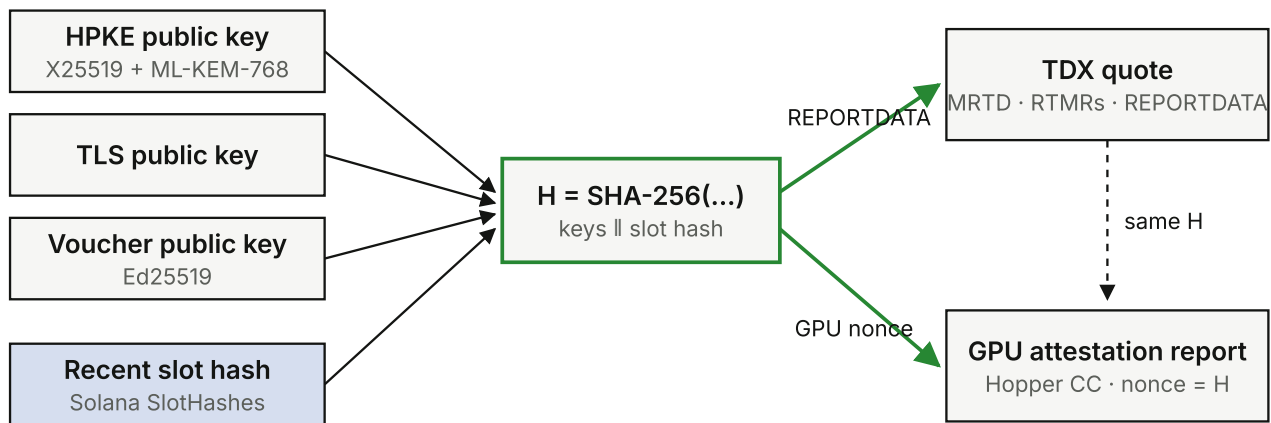
```
TDX REPORTDATA (64 bytes)
  bytes 0..31  H
  bytes 32..63  reserved; a verifier rejects unexpected bytes here

GPU attestation nonce (32 bytes), for every GPU in the node
  bytes 0..31  H
```

The complete procedure inside the CVM is:

1. Generate the HPKE, TLS and voucher key pairs.
2. Build the 1,280-byte transcript and compute `enclave_key_hash`.
3. Read `(recent_slot, recent_slot_hash)` from SlotHashes.
4. Compute H over the transcript and the slot hash.
5. Call `TDG.MR.REPORT` with `REPORTDATA = H` in bytes 0 to 31, and have the Quoting Enclave convert the TDREPORT into a TD quote.
6. For each assigned GPU, request an attestation report with nonce = H.
7. Assemble the evidence object.

H links four facts. The TD quote proves that a measured runtime on a specific Intel platform produced H. Each GPU report proves that a specific genuine GPU, in a measured state, was asked to attest with H. H commits to exactly one set of enclave public keys, so the HPKE key the broker wraps to, the TLS key the caller connects to and the voucher key that signs charges all belong to that runtime. And H commits to a slot hash, so all of it happened after that slot.



Verifiers recompute H from the enclave key hash and the slot hash named in the record; both reports must carry it.

Figure: The three enclave public keys and a recent slot hash are hashed into H, which is written into TDX REPORTDATA and used as every GPU's attestation nonce.

3.5.5 The evidence object

The node submits its evidence as a strict JSON object, format `sealroom-evidence-v1`:

Field	Content	Bound
<code>version</code>	"sealroom-evidence-v1"	Literal
<code>nodeId</code>	Node account address	32-byte base58 key
<code>recentSlot</code>	Slot number	u64 decimal string
<code>recentSlotHash</code>	Slot hash	64 hex characters
<code>keys.hpkePublicKey</code>	X-Wing public key	Base64, exactly 1,216 bytes
<code>keys.tlsPublicKey</code>	TLS Ed25519 public key	Base64, exactly 32 bytes
<code>keys.voucherPublicKey</code>	Voucher Ed25519 public key	32-byte base58 key
<code>tdxQuote</code>	TD quote	Canonical base64, at most 700,000 characters
<code>gpuEvidence</code>	One report per GPU	1 to 8 canonical base64 blobs

The whole object is bounded to 1 MiB. Its hash is taken over a serialization with a fixed field order (`version`, `nodeId`, `recentSlot`, `recentSlotHash`, `keys` with `hpkePublicKey`, `tlsPublicKey`, `voucherPublicKey`, then `tdxQuote`, `gpuEvidence`):

```
evidence_hash = SHA-256(serialize(evidence))
```

`evidence_hash` is what the committee signs into the attestation and what verifiers use as the key of their private evidence archives. Any party holding the archived object can recompute it, recompute `enclave_key_hash` and `H`, and compare them with the chain.

3.6 What each verifier checks

Four parties examine a node's evidence or its on-chain result: each committee verifier (vendor verification, then registry checks), the program, the owner's key broker and the caller. Checks are listed with the error codes they produce.

3.6.1 Vendor verification

Each committee verifier runs Intel quote verification (Section 3.2.5) and NVIDIA GPU verification (Section 3.3.3) and reduces the result to a fixed set of claims:

Claim	Size	Source
<code>platformIdHash</code>	32	Platform identity from the PCK certificate in the quote
<code>runtimeMeasurement</code>	48	The quote's measured runtime image
<code>reportDataHash</code>	32	Bytes 0 to 31 of REPORTDATA
<code>tcbStatus</code>	enum	Intel TCB evaluation; only <code>UpToDate</code> is admitted
<code>gpuUeidHashes</code>	32 each	One per GPU report, from the device certificate
<code>gpuNonceHashes</code>	32 each	The nonce in each GPU report
<code>gpuRimHash</code>	32	The RIM set every GPU was checked against

3.6.2 Registry checks by the committee verifier

The verifier reads all policy accounts from RPC in one context, retrying if any changes between reads, and checks:

1. The chain's genesis hash is the expected one and the evidence names the node read (`CHAIN_IDENTITY_MISMATCH` , `NODE_MISMATCH`).
2. Protocol running, operator unfrozen, node enabled (`PROTOCOL_PAUSED` , `OPERATOR_FROZEN` , `NODE_DISABLED`).
3. The platform is registered to this operator, its `platform_id_hash` equals the claim, and the global `PlatformPin` agrees (`PLATFORM_NOT_REGISTERED` , `PLATFORM_ID_MISMATCH` , `GLOBAL_PLATFORM_PIN_MISMATCH`).
4. The UEID claims are distinct, match the node's assigned GPUs one for one, are registered on the platform, and every `GpuPin` names this platform and node (`GPU_UEID_MISMATCH` , `EXCLUSIVE_GPU_PIN_MISMATCH`).
5. The runtime is allowed, unrevoked and effective (`RUNTIME_NOT_ALLOWED`).
6. The slot pair is in SlotHashes (`RECENT_SLOT_HASH_MISMATCH`).

7. `reportDataHash` and every GPU nonce equal H recomputed from the evidence (`ENCLAVE_KEY_BINDING_MISMATCH`).
8. The bond meets the tier minimum (`OPERATOR_BOND_BELOW_TIER_MINIMUM`).
9. No sticky revocation; next sequence; current operator epoch; active verifier epoch with this verifier a member; expiry within the TTL (`ATTESTATION_REVOKED_ADMIN_CLEAR_REQUIRED` , `ATTESTATION_SEQUENCE_MISMATCH` , `VERIFIER_NOT_ACTIVE` , `ATTESTATION_EXPIRY_INVALID`).

Only then does the verifier archive the evidence and sign a 200-byte message: the 16-byte domain `sealroom.att.v0\0` , the program id, the genesis hash, the node, the verifier epoch, the replay sequence, the SHA-256 of a 194-byte commitment, the recent slot and the recent slot hash. The commitment carries the platform, the runtime measurement, the GPU RIM hash, the evidence hash, the enclave key hash, the TCB status, the expiry, the method and the operator epoch. Chapter 5 gives the byte offsets.

3.6.3 The program

`record_attestation` is permissionless to relay and checks the result rather than the raw evidence:

Check	Error
Protocol not paused; operator not frozen and bond at its tier minimum	<code>Paused</code> , <code>OperatorFrozen</code> , <code>InsufficientBond</code>
Platform registered; node <code>Registered</code> ; every node GPU on the platform	<code>PlatformNotRegistered</code> , <code>NodeDisabled</code> , <code>InvalidGpuSet</code>
Runtime not revoked and effective; commitment's measurement and platform match the accounts	<code>RuntimeNotAllowed</code> , <code>AttestationMismatch</code>
TCB <code>UpToDate</code> and method <code>Committee</code>	<code>InvalidTcb</code>
Expiry after now, within 86,400 seconds, and not after the operator's unbond deadline	<code>AttestationExpired</code> , <code>AttestationBeyondUnbond</code>
Epoch is the active, unretired verifier epoch; operator epoch is current	<code>InvalidVerifierSet</code> , <code>AttestationReplay</code>
Not revoked; sequence is exactly the next one	<code>AttestationRevoked</code> , <code>AttestationReplay</code>
Slot pair present in <code>SlotHashes</code>	<code>InvalidRecentSlot</code>
Ed25519 precompile instructions earlier in the transaction carry signatures over exactly the derived message by at least 3 distinct members of a 5-member set	<code>WrongAttestationMessage</code> , <code>UnknownVerifier</code> , <code>DuplicateVerifier</code> , <code>InsufficientSignatures</code>

The resulting `Attestation` account (384 bytes) stores the committed values together with the slot pair, `recorded_at` , the verifier epoch, a signer bitmap and the SHA-256 of the signed message. Chapter 7 gives its full layout.

3.6.4 The key broker

The owner's broker repeats the decisive checks with its own reads and its own vendor verification. It requires the model to name this broker (`BROKER_AUTHORITY_MISMATCH`), the evidence hash, enclave key hash and slot pair to equal the attestation (`EVIDENCE_CHAIN_MISMATCH`), the eligibility rules and the registry checks of Section 3.6.2 to pass on its fresh vendor claims, and the runtime measurement and RIM hash to equal those recorded (`VENDOR_CLAIMS_CHAIN_MISMATCH`). The slot pair need not still be in SlotHashes at release time, because freshness was proved when the attestation was recorded and the record carries its own 24-hour expiry. A final chain read must match the first (`REGISTRY_CHANGED_DURING_RELEASE`).

3.6.5 The caller

The caller checks that the node it is about to send data to is the node the chain attested:

1. Read `Attestation ["att", node]` and the model's eligibility for this node: current, unexpired, unrevoked, operator epoch current, runtime pinned by the model.
2. Obtain the node's three public keys, recompute `enclave_key_hash` and compare it with the account. The registry API also publishes `tlsPublicKeyHash`, `hpkePublicKeyHash` and `voucherPublicKey` for each attestation whose archived evidence reproduces the on-chain value.
3. In the TLS handshake, require the server's Ed25519 public key to equal the 32-byte TLS key in the transcript.
4. Under `upto`, accept a usage voucher only if the transcript's voucher key signed it and it names the current attestation and revision.

These checks use only public data, so a caller performs them without an account with any party.

4. Operator admission and platform pinning

Sealroom admits operators by decision of its admin multisig after off-chain vetting, and it accepts a node only on hardware that an admitted operator registered. The threat model behind this design is an attacker with physical control of a confidential-computing server, who can extract the platform's attestation keys and produce quotes that verify as `UpToDate`. If key release accepted any genuine Intel platform, one purchased server would be enough to impersonate a node. Sealroom therefore accepts evidence only when its platform identity and every GPU UEID were registered by a vetted, bonded operator bound by the Operator Agreement.

This chapter follows the admission path in the order an operator walks it.

4.1 Off-chain vetting

4.1.1 Intake

An operator applies through the Sealroom intake at `/apply` with a strict, typed record:

Field	Content	Bound
<code>organisation</code> , <code>contactName</code> , <code>contactEmail</code> , <code>website</code>	Applicant and contact	Names 2 to 120 characters; optional HTTPS website
<code>authority</code>	Wallet that will control the <code>Operator</code> account	32-byte base58 key, optional
<code>datacenterRegions</code>	Regions of the facilities offered	1 to 16 entries
<code>gpuFleet</code>	GPU model and count	1 to 8 entries; <code>H100</code> or <code>H200</code> ; count 1 to 100,000
<code>certifications</code>	Datacenter certifications held	At most 16 entries
<code>confidentialComputeReady</code>	Whether the fleet runs TDX with GPUs in CC mode	Boolean

Each submission carries an `Idempotency-Key` and passes a server-side Cloudflare Turnstile challenge. Its status moves from `received` to `reviewing`, then to `accepted` or `declined`.

4.1.2 What vetting establishes

Vetting is a legal and physical-security decision, so it happens off chain. Sealroom establishes:

1. **Legal entity.** The applicant is an identified company, with KYC on the entity behind the operator authority.

2. **Facilities.** The datacenters that will host registered servers, and their certifications.
3. **Hardware.** Intel TDX with NVIDIA H100 or H200 GPUs in confidential computing mode, the v0 hardware.
4. **Machine identities.** For each server, the platform identity (the Intel PPID carried by its PCK certificate) and the UEID of every GPU. The admin later co-signs these records into the registry.
5. **The Operator Agreement.** The contract under which the arbitration panel rules on disputes and slashes bonds.

Vetting continues as periodic review after onboarding, charged at cost recovery.

4.1.3 What reaches the chain

Vetting produces the operator's authority wallet, the hash of its signed agreement and its tier, which `register_operator` records, and the platform and GPU identities of each server, which `register_platform` records with their global pins. Both instructions are signed by `Config.admin`, the Squads admin multisig, which keeps the registry's identities tied to the vetting records.

4.2 `register_operator` and tiers

4.2.1 The instruction

`register_operator(agreement_hash: [u8; 32], tier: u8)` creates the operator's account, its bond vault and its link to a payout account. The operator does not sign: admission is the admin's act. The operator's first signature on chain is its first `deposit_bond`.

Account	Role and constraint
<code>admin</code>	Signer and rent payer; must equal <code>Config.admin</code>
<code>config</code>	<code>Config ["config"]</code>
<code>authority</code>	The admitted operator key; read only, does not sign
<code>operator</code>	Created: <code>Operator ["op", authority]</code> , 193 bytes
<code>usdc_mint</code>	Must equal <code>Config.usdc_mint</code>
<code>bond_vault</code>	USDC associated token account whose authority is the <code>operator</code> PDA; created if absent
<code>payout_ata</code>	The canonical USDC associated token account of <code>authority</code> ; must already exist
<code>token_program</code> , <code>associated_token_program</code> , <code>system_program</code>	Programs used to create the vault

Checks, in order:

1. The signer is `Config.admin` (`Unauthorized`).

2. The protocol is not paused (`Paused`).

3. `Config.tier_bond_minimums` has an entry for `tier` (`ConfigurationUnset`).

The instruction then writes the account and emits `OperatorRegistered { operator, authority, tier, agreement_hash, timestamp }`. Because the PDA is seeded by the authority alone, one wallet controls at most one `Operator`.

4.2.2 The `Operator` account

Field	Type	Bytes	Value at registration
<code>discriminator</code>		8	Anchor account discriminator
<code>version</code> , <code>bump</code>	<code>u8</code> , <code>u8</code>	2	1, PDA bump
<code>authority</code>	<code>Pubkey</code>	32	Admitted wallet
<code>agreement_hash</code>	<code>[u8; 32]</code>	32	Hash of the signed Operator Agreement
<code>tier</code>	<code>u8</code>	1	1, 2 or 3
<code>bond_vault</code>	<code>Pubkey</code>	32	USDC vault owned by the <code>operator</code> PDA
<code>bond_amount</code>	<code>u64</code>	8	0
<code>unbond_at</code>	<code>Option<i64></code>	9	None
<code>frozen</code>	<code>bool</code>	1	false
<code>payout_ata</code>	<code>Pubkey</code>	32	Authority's canonical USDC account
<code>open_dispute_count</code>	<code>u32</code>	4	0
<code>next_dispute_sequence</code>	<code>u64</code>	8	0
<code>registered_at</code>	<code>i64</code>	8	Chain clock
<code>attestation_epoch</code>	<code>u64</code>	8	0
<code>pending_slash_amount</code>	<code>u64</code>	8	0
Total		193	

`payout_ata` is where the Operator-role share of every lane on this operator's nodes is paid (Chapter 9). `attestation_epoch` is the operator-wide revocation counter described in Section 5.9.

4.2.3 Tiers

`Config.tier_bond_minimums` is a list of exactly three `TierBondMinimum { tier: u8, amount: u64 }` entries, for tiers 1, 2 and 3 in that order. The program refuses to initialize with any other shape (`ConfigurationUnset`, `InvalidTier`), and it refuses any `register_operator` or `set_tier` whose tier has no entry.

`set_tier(tier)` is signed by the admin and changes `operator.tier` under the same check, emitting `operatorTierSet`. A tier change takes effect at once in every rule that reads it: if the

operator's bond is below the new tier's minimum, it stops being healthy (Section 4.3) and its next attestation is refused.

Models use tiers as a floor. `Model.min_tier` is between 1 and 3, and the key broker releases only to nodes whose operator's tier is at least that value (`tier_below_min` , Section 6.4).

4.3 Bonds per tier

4.3.1 The bond vault

Every operator's bond sits in its `bond_vault` , a USDC token account whose authority is the `Operator` PDA, so only the program can move it. Amounts are USDC base units (6 decimals), and every transfer is a `TransferChecked` .

`Operator.bond_amount` is the tracked principal. It rises only through `deposit_bond` and falls only through withdrawal or a slash. Tokens sent to the vault by a plain transfer do not increase it, and they do not count toward any minimum.

4.3.2 Health

An operator is healthy when it is not frozen and its tracked bond meets its tier's minimum:

```
tier_minimum = Config.tier_bond_minimums[Operator.tier].amount
healthy = (Operator.frozen == false) and (Operator.bond_amount >=
tier_minimum)
```

Health is required by `register_platform` , `register_node` and `record_attestation` . The failures are `OperatorFrozen` and `InsufficientBond` .

A model can demand more than the protocol does. At release time the key broker applies the larger of the two minimums:

```
release_minimum = max(Model.min_bond, tier_minimum)
eligible_bond = Operator.bond_amount >= release_minimum
```

4.3.3 Deposits, unbonding and withdrawal

Instruction	Signer	Preconditions	Effect
<code>deposit_bond(amount)</code>	Operator authority	Not paused; <code>amount > 0</code> ; not frozen; no unbond request	Transfers <code>amount</code> from the authority's USDC account to the vault; <code>bond_amount += amount</code>
<code>request_unbond</code>	Operator authority	<code>bond_amount > 0</code> ; not frozen; <code>open_dispute_count == 0</code> ; no request pending	<code>unbond_at = now + 2,592,000</code> (30 days)

Instruction	Signer	Preconditions	Effect
<code>cancel_unbond</code>	Operator authority	A request is pending	<code>unbond_at = None</code> ; no funds move
<code>withdraw_bond</code>	Operator authority	Not frozen; <code>open_dispute_count == 0</code> ; <code>now >= unbond_at</code> ; vault balance at least <code>bond_amount</code>	Transfers the full <code>bond_amount</code> to the authority's USDC account; <code>bond_amount = 0</code> ; <code>unbond_at = None</code>

A slash ruled by arbitration subtracts from `bond_amount` and adds to `pending_slash_amount` in the same instruction, so a reserved slash is never part of the withdrawable principal. Chapter 10 describes slashing and its settlement.

4.3.4 Attestations follow the bond

An operator that has asked to leave cannot hold attestations past its departure. `record_attestation` requires every new attestation to expire no later than `unbond_at` when one is set (`AttestationBeyondUnbond`):

```

expires_at > now
expires_at <= now + 86400
expires_at <= Operator.unbond_at           (when unbond_at is set)

```

Worked timeline, in days from the unbond request:

Time	Event	Longest attestation the program accepts
Day 0	<code>request_unbond</code> ; <code>unbond_at = day 30</code>	24 hours
Day 28	Daily renewal	24 hours, ends day 29
Day 29, 12:00	Renewal	12 hours, ends exactly at day 30
Day 30 onward	Any renewal	None: no expiry can be after now and at or before <code>unbond_at</code>
Day 30 onward	<code>withdraw_bond</code>	Bond returned; the operator is no longer healthy

When the bond leaves the vault, every node of the operator is already outside a valid attestation, and the broker has nothing to release against.

4.4 register_platform

4.4.1 What a platform is

A `Platform` is one physical server: one Intel platform identity and the GPUs installed in it. Its identity is stored as `platform_id_hash`, a 32-byte hash of the PPID/PCK identity that the server's TD quotes carry in their PCK certificate chain (identity kind `intel_ppid_pck` in the API). Each GPU is stored as `gpu_ueid_hash`, a 32-byte hash of the UEID from its NVIDIA device certificate. A platform holds 1 to 8 GPUs. Section 3.4 explains how the verifier derives both identities from evidence.

4.4.2 The instruction

`register_platform(platform_id_hash: [u8; 32], gpu_ueid_hashes: Vec<[u8; 32]>)` requires two signatures: the operator authority, which pays rent and binds the machine to its bond, and the admin multisig, which attests that the identities match the vetting record.

Account	Role and constraint
<code>authority</code>	Signer, rent payer; <code>has_one</code> on the <code>Operator</code>
<code>admin</code>	Signer; must equal <code>Config.admin</code>
<code>config</code>	<code>Config ["config"]</code>
<code>operator</code>	<code>Operator ["op", authority]</code>
<code>platform</code>	Created: <code>Platform ["plat", operator, platform_id_hash]</code> , 343 bytes
<code>platform_pin</code>	Created: <code>PlatformPin ["platform_pin", platform_id_hash]</code> , 106 bytes
<code>system_program</code>	Account creation
remaining accounts	One writable <code>GpuPin ["gpu_pin", gpu_ueid_hash]</code> per GPU, in the order of <code>gpu_ueid_hashes</code>

Steps:

1. The admin signed, the protocol is running and the operator is healthy (`Unauthorized` , `Paused` , `OperatorFrozen` , `InsufficientBond`).
2. `gpu_ueid_hashes` has 1 to 8 distinct entries and exactly that many remaining accounts are supplied (`InvalidGpuSet`).
3. Each `GpuPin` address equals its derived PDA, is writable, holds no data and is owned by the System Program (`GpuAlreadyClaimed`). The program tops up rent if the address already holds lamports, allocates 107 bytes, assigns the account to itself and writes `{ platform, gpu_ueid_hash, node: None }`.
4. The `PlatformPin` is created; creation fails if the address is in use.
5. The `Platform` is written with `registered = true`, and `PlatformRegistered` is emitted.

The steps run in one transaction, so a platform is registered with all of its pins or not at all.

4.4.3 Layouts

`Platform` (343 bytes):

Field	Type	Bytes
discriminator, <code>version</code> , <code>bump</code>		10
<code>operator</code>	Pubkey	32
<code>platform_id_hash</code>	[u8; 32]	32
<code>gpu_ueid_hashes</code>	Vec<[u8; 32]>, at most 8	4 + 256
<code>registered</code>	bool	1
<code>registered_at</code>	i64	8

`PlatformPin` (106 bytes) and `GpuPin` (107 bytes):

Account	Seeds	Fields
<code>PlatformPin</code>	<code>["platform_pin", platform_id_hash]</code>	<code>platform: Pubkey, operator: Pubkey, platform_id_hash: [u8; 32]</code>
<code>GpuPin</code>	<code>["gpu_pin", gpu_ueid_hash]</code>	<code>platform: Pubkey, gpu_ueid_hash: [u8; 32], node: Option<Pubkey></code>

4.5 Global pins: one machine, one registration

The `Platform` PDA includes the operator in its seeds, so by itself it would let two operators each create a `Platform` for the same server. The pins close that gap. Their seeds contain only the identity hash, so each pin address exists once across the whole program:

- `PlatformPin ["platform_pin", platform_id_hash]` maps one platform identity to one operator and one `Platform` account.
- `GpuPin ["gpu_pin", gpu_ueid_hash]` maps one GPU to one platform, and after `register_node` to one node.

Attempt	What stops it	Result
The same operator registers the same server twice	<code>Platform</code> and <code>PlatformPin</code> addresses are in use	Account creation fails
A second operator registers a server already pinned	<code>PlatformPin</code> address is in use	Account creation fails
A new platform lists a GPU pinned to another platform	<code>GpuPin</code> holds data and is program-owned	<code>GpuAlreadyClaimed</code>
A third party pre-funds a future <code>GpuPin</code> address to block it	The program tops up rent, then allocates and assigns	Registration proceeds

Attempt	What stops it	Result
Two nodes claim the same GPU	<code>GpuPin.node</code> is already set	<code>GpuAlreadyClaimed</code>
A node claims a GPU registered on a different platform	Hash absent from this platform's list, or <code>GpuPin.platform</code> differs	<code>InvalidGpuSet</code> or <code>GpuAlreadyClaimed</code>

Pins have no release or migration instruction. A platform identity or GPU, once registered, stays bound to the operator and platform that first registered it.

The verifier and the broker read the pins again at every attestation and every release. They require the `PlatformPin` to name this platform, the node's operator and the identity in the quote (`GLOBAL_PLATFORM_PIN_MISMATCH`); the GPU evidence to carry exactly the node's assigned UEIDs (`GPU_UEID_MISMATCH`); and every `GpuPin` to name this platform and this node (`EXCLUSIVE_GPU_PIN_MISMATCH`). Evidence borrowed from a GPU outside the node fails the UEID check. The GPU nonce, bound to H (Section 3.5.4), proves the evidence is fresh; the UEID check proves which GPUs produced it.

4.6 `register_node`

4.6.1 The instruction

A node is one CVM: a TDX trust domain with 1 to 8 of its platform's GPUs, running the measured Sealroom runtime. The operator registers it alone; the admin's co-signature was given at the platform level.

```
register_node(args: RegisterNodeArgs):
```

Argument	Type	Bound
<code>index</code>	u16	0 to 65,535; part of the PDA seed
<code>node_key</code>	Pubkey	The node's 32-byte public key
<code>endpoint</code>	String	HTTPS URL, at most 128 bytes
<code>gpu_model</code>	enum	<code>H100</code> or <code>H200</code>
<code>gpu_ueid_hashes</code>	<code>Vec<[u8; 32]></code>	1 to 8 distinct hashes, each registered on the platform
Account	Role and constraint	
<code>authority</code>	Signer, rent payer	
<code>config</code>	<code>Config ["config"]</code>	
<code>operator</code>	<code>Operator ["op", authority]</code> , <code>has_one</code> authority	
<code>platform</code>	<code>Platform ["plat", operator, platform_id_hash]</code> , <code>has_one</code> operator	

Account	Role and constraint
node	Created: Node ["node", platform, index] (index as u16 little-endian), 503 bytes
system_program	Account creation
remaining accounts	One writable GpuPin per hash in gpu_ueid_hashes, same order

Steps:

1. The protocol is running, the operator is healthy and the platform is registered (Paused, OperatorFrozen, InsufficientBond, PlatformNotRegistered).
2. The endpoint passes the URL rules below (InvalidEndpoint).
3. gpu_ueid_hashes has 1 to 8 distinct entries, all in Platform.gpu_ueid_hashes, with one GpuPin supplied per entry (InvalidGpuSet).
4. Each GpuPin is at its derived address, program-owned, with a matching bump and hash (InvalidGpuSet), names this platform and has no node (GpuAlreadyClaimed). The program sets GpuPin.node to the new node.
5. The Node is written with gpu_count equal to the number of hashes and status Registered, and NodeRegistered is emitted.

Because each GPU can be claimed by one node and every node has at least one GPU, a platform's GPUs are partitioned among its nodes: an 8-GPU server can be one 8-GPU node, eight 1-GPU nodes, or any split in between.

4.6.2 Endpoint rules

The program accepts only an unambiguous HTTPS URL: ASCII, at most 128 bytes, no whitespace, control characters, backslash or user@ component; an optional port from 1 to 65,535; host labels of 1 to 63 letters, digits or hyphens with no xn-- labels; and numeric hosts only as canonical four-part IPv4 or bracketed IPv6. The same rules apply to Model.key_broker_url.

4.6.3 The Node account

Field	Type	Bytes
discriminator, version, bump		10
platform	Pubkey	32
operator	Pubkey	32
index	u16	2
node_key	Pubkey	32
endpoint	String, at most 128 bytes	4 + 128
gpu_model	enum (H100 = 0, H200 = 1)	1
gpu_count	u8	1

Field	Type	Bytes
gpu_ueid_hashes	Vec<[u8; 32]>, at most 8	4 + 256
status	enum (Registered = 0, Disabled = 1)	1
Total		503

A Disabled node cannot be attested (NodeDisabled) and fails the broker's node_disabled check. The registry API reports node status as registered , serving , offline or disabled ; serving means the node is healthy and its current attestation matches its evidence.

4.7 Operator status and freezing

4.7.1 Status

The registry API derives one status for each operator from its account, in this order:

Condition	Status
frozen == true	frozen
unbond_at is set	unbonding
bond_amount == 0	unbonded
Otherwise	bonded

The API also publishes tier , agreementHash , bondAmount , the vault balance , attestationEpoch , openDisputeCount , pendingSlashAmount and the operator's platforms and nodes.

4.7.2 Who can freeze

Path	Signer	Effect
freeze_operator(frozen: bool)	Config.admin	Sets or clears Operator.frozen ; emits OperatorFrozen
rule_dispute(..., freeze = true, ...)	Config.arbitration	Sets frozen , as part of a ruling for the claimant
arbitrate_operator(..., freeze = true, ...)	Config.arbitration	Sets frozen as a security action, with or without a slash

None of these paths is blocked by the protocol pause, and none makes a token transfer, so a frozen USDC account cannot prevent a freeze. Clearing a freeze is an admin decision through freeze_operator(false) .

4.7.3 What freezing stops

Operation	Result for a frozen operator
<code>deposit_bond</code> , <code>request_unbond</code> , <code>withdraw_bond</code>	<code>OperatorFrozen</code> : the bond stays in the vault for arbitration
<code>register_platform</code> , <code>register_node</code>	<code>OperatorFrozen</code>
<code>record_attestation</code>	<code>OperatorFrozen</code> : no new or renewed attestation
Verifier signing	<code>OPERATOR_FROZEN</code> : no committee signature
Key broker	<code>operator_frozen</code> and <code>OPERATOR_FROZEN</code> : no key release
Node serving	The node's eligibility check before inference and again before settlement fails, so no call is served or charged

A freeze stops new key releases and service at the next read. To invalidate every attestation the operator holds at once, arbitration also advances `operator.attestation_epoch` (Section 5.9.4), which makes every attestation signed under the previous epoch revoked for every consumer.

5. The attestation pipeline

An `Attestation` account states that a node, on a registered platform with registered GPUs, booted an allowed runtime measurement and holds a specific set of enclave keys, and that at least three of the five committee verifiers checked its vendor evidence within the last 24 hours. This chapter follows that statement from evidence to transaction, then through revocation, renewal and the ZK path that follows the committee.

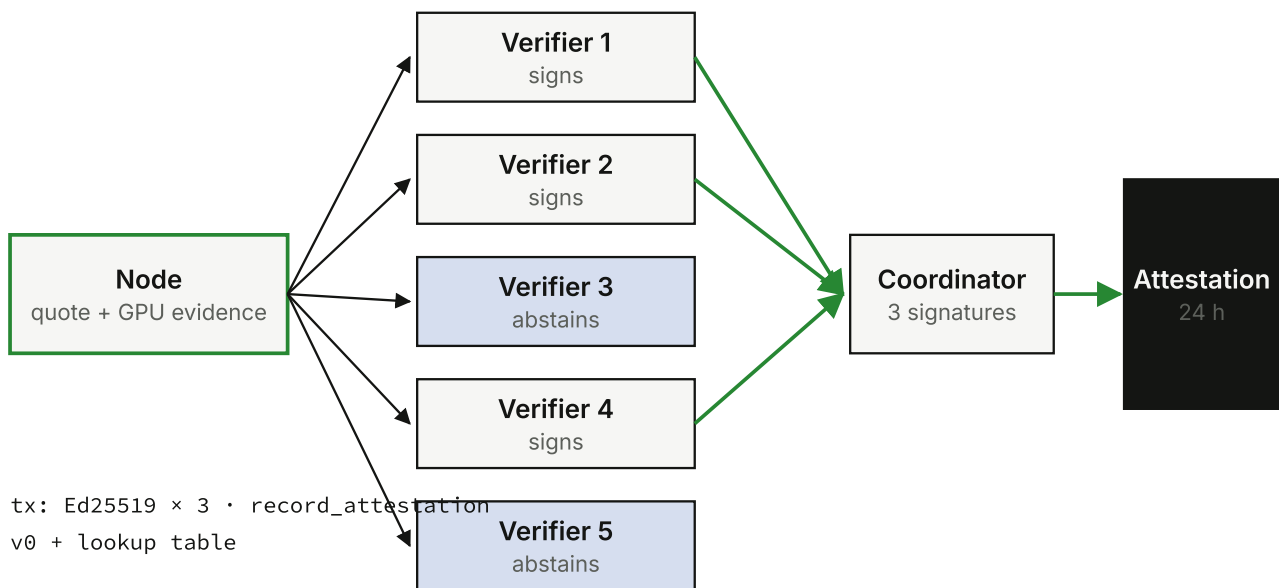


Figure: The node submits bound evidence to the coordinator, five independent verifiers check it and sign one 200-byte message, and three signatures land on chain through the `Ed25519` precompile and `record_attestation`.

5.1 The pipeline end to end

1. **Bind.** The runtime generates its enclave keys, binds them and a recent slot hash into `H`, and obtains a TD quote and one GPU report per GPU carrying `H` (Section 3.5).
2. **Submit.** The node sends its `sealroom-evidence-v1` object to the attestation coordinator.
3. **Propose.** The coordinator verifies the evidence, reads the registry and fixes the next sequence, the verifier epoch, the operator epoch and the expiry.
4. **Sign.** All five verifiers check the evidence independently; each archives it and signs the 200-byte canonical message.

5. **Record.** The coordinator keeps three valid signatures and submits an Ed25519 precompile instruction followed by `record_attestation`, which re-derives the message, checks the signatures through the instructions `sysvar` and writes `Attestation ["att", node]`.
6. **Publish.** The indexer mirrors `AttestationRecorded` as a new revision for the API, callers and key brokers.

5.2 The verifier set

5.2.1 Composition

The verifier committee has five members. They are independent parties; Sealroom operates at most two of the five. Each verifier runs as its own service with exactly one Ed25519 signing key, its own credential, its own host and secret store, and its own cache of Intel and NVIDIA collateral. A quorum is three distinct members signing the same message, so the committee produces attestations with any two members unavailable, and no two members can produce one on their own.

The coordinator is a separate service that holds no verifier key. Its only key is the fee payer that submits transactions, and it refuses to run if that key is a committee member. It cannot create a committee signature; it can only collect and relay them.

5.2.2 The `VerifierSet` account

Field	Type	Bytes
discriminator, <code>version</code> , <code>bump</code>		10
<code>epoch</code>	<code>u64</code>	8
<code>members</code>	<code>Vec<Pubkey></code> , at most 7	4 + 224
<code>threshold</code>	<code>u8</code>	1
<code>activated_at</code>	<code>i64</code>	8
<code>retired_at</code>	<code>Option<i64></code>	9
Total		264

The PDA is `["verifiers", epoch]` with the epoch as `u64` little-endian, so each committee membership lives in its own account. `Config.active_verifier_epoch` names the set whose signatures count. The account allocates room for seven members; the v0 committee is exactly five distinct, non-default keys with threshold 3. The program enforces that shape at initialization (`InvalidVerifierSet`) and again inside every `record_attestation`, before it counts a single signature. A set with `retired_at` set is never accepted.

A member's position in `members` is its bit in the attestation's `signer_bitmap`: member 0 is bit 0, member 4 is bit 4. Bitmap `0b00000111` (7) means members 0, 1 and 2 signed.

5.3 What each verifier checks

A verifier exposes one signing route that accepts only server requests carrying its own credential. The request is an attestation proposal: the `evidence` object plus the proposed `sequence`, verifier `epoch`, `operatorEpoch` and `expiresAt`. The verifier works through three groups of checks; Section 3.6 lists each with its error code.

1. **Vendor verification.** The TD quote is verified against Intel collateral (DCAP with Intel PCS, or Intel Trust Authority), admitting only TCB status `UpToDate`. Each GPU report is verified against NVIDIA's reference integrity manifests and OCSP with the NVIDIA local verifier, with NVIDIA's remote attestation service as an optional cross-check.
2. **Registry checks.** In one RPC read at `confirmed` commitment, the verifier loads `Config`, `Node`, `Platform`, `Operator`, `Attestation`, the active `VerifierSet`, the claimed `Runtime`, the `PlatformPin`, every `GpuPin` and the `Clock` and `SlotHashes` sysvars. It checks the genesis hash, pause and freeze flags, platform identity and pins, GPU set, runtime activation, slot pair, H binding and tier bond. "Now" is the chain's `Clock`.
3. **Proposal checks.** No sticky revocation is in place; `sequence` is the current sequence plus one; `operatorEpoch` equals `Operator.attestation_epoch`; `epoch` is active and includes this verifier; `expiresAt` is after now and at most 86,400 seconds ahead.

When every check passes, the verifier archives the evidence privately under `evidence_hash` with its byte length, format version and capture time; builds the 200-byte canonical message (Section 5.4) from its own registry read and vendor claims; and returns `{ publicKey, signature, messageHash, evidenceHash }`. Raw quotes run to several kilobytes and include the PCK certificate chain, so they stay off chain; the on-chain hash and the archive let an auditor reproduce the verification. A verifier signs only a message it computed itself.

5.4 The canonical attestation message

5.4.1 Message layout

Every committee member signs the same 200 bytes. The program builds the identical bytes from its own accounts and the instruction arguments, and the client SDK encodes them the same way.

Offset	Bytes	Field	Source in <code>record_attestation</code>
0	16	Domain <code>sealroom.att.v0</code> followed by one NUL byte	Constant
16	32	Program id	<code>crate::ID</code>
48	32	Genesis hash	<code>Config.genesis_hash</code>
80	32	Node PDA	<code>node</code> account address
112	8	Verifier epoch, u64 LE	<code>args.epoch</code>
120	8	Replay sequence, u64 LE	<code>args.sequence</code>

Offset	Bytes	Field	Source in <code>record_attestation</code>
128	32	SHA-256 of the Borsh-encoded commitment	<code>args.commitment</code>
160	8	Recent slot, u64 LE	<code>args.recent_slot</code>
168	32	Recent slot hash	<code>args.recent_slot_hash</code>
200			

5.4.2 Commitment layout

The commitment is the `AttestationCommitment` struct, Borsh-encoded in field order. It is always 194 bytes.

Offset	Bytes	Field	Encoding
0	32	<code>platform</code>	Platform PDA
32	48	<code>runtime_measurement</code>	Raw measurement
80	32	<code>gpu_rim_hash</code>	Raw hash
112	32	<code>evidence_hash</code>	SHA-256 of the evidence serialization
144	32	<code>enclave_key_hash</code>	SHA-256 of the 1,280-byte key transcript
176	1	<code>tcb_status</code>	Enum tag: <code>UpToDate</code> 0, <code>OutOfDate</code> 1, <code>ConfigurationNeeded</code> 2, <code>Revoked</code> 3, <code>Unknown</code> 4
177	8	<code>expires_at</code>	i64 LE, Unix seconds
185	1	<code>method</code>	Enum tag: <code>Committee</code> 0, <code>Zk</code> 1
186	8	<code>operator_epoch</code>	u64 LE
194			

```

commitment_hash = SHA-256(borsh(AttestationCommitment))      (194 bytes
in, 32 out)
message = domain || program_id || genesis_hash || node || epoch ||
sequence || commitment_hash || recent_slot || recent_slot_hash
canonical_message_hash = SHA-256(message)                    (stored
in the Attestation)

```

5.4.3 What each field binds

Field	A signature over it cannot be reused for
Domain	Any other message type signed with a verifier key
Program id	Another deployment of the program

Field	A signature over it cannot be reused for
Genesis hash	Another cluster
Node PDA	Another node, including another node on the same platform
Verifier epoch	A committee other than the active one
Replay sequence	Any earlier or later revision of this node's attestation
Commitment hash	A different platform, runtime, GPU reference set, evidence, key set, TCB status, expiry, method or operator epoch
Recent slot and hash	A recording after that slot has left SlotHashes

5.5 Collecting signatures and the Ed25519 packet

5.5.1 The coordinator

The coordinator runs the vendor verification and registry checks itself before it asks anyone to sign, then fixes the proposal:

```
sequence = Attestation.sequence + 1          (1 when the node has no
Attestation yet)
epoch = Config.active_verifier_epoch
operator_epoch = Operator.attestation_epoch
expires_at = clock_now + 86400 - 1
```

It sends the proposal to all five verifiers in parallel, each with a 30-second timeout. It accepts a response only if the public key equals the key configured for that endpoint and is a member of the active set, the `evidenceHash` matches, and the Ed25519 signature verifies over the coordinator's own copy of the message. Valid responses are deduplicated by public key; with fewer than three the coordinator records nothing (`VERIFIER_QUORUM_UNAVAILABLE`), and with more it uses exactly three.

5.5.2 Precompile instruction layout

The three signatures travel in one instruction to Solana's native Ed25519 program (`Ed25519SigVerify11111111111111111111111111111111`), placed before `record_attestation` in the same transaction. The message appears once and all three signatures reference it.

Ed25519 precompile instruction data, n signatures

offset	size	field
0	1	n (1 to 7)
1	1	padding, must be 0
2	14 x n	one offsets record per signature, seven u16 LE values: signature_offset signature_instruction_index = 0xFFFF public_key_offset public_key_instruction_index = 0xFFFF message_data_offset message_data_size = 200 message_instruction_index = 0xFFFF
2 + 14n	96 x n	per signer: public key (32), then signature (64)
2 + 110n	200	canonical message, shared by all signatures

For the three-signature packet:

$$\text{data_bytes} = 2 + 3 * 14 + 3 * 96 + 200 = 532$$

5.5.3 How the program reads the packet

The Ed25519 program verifies every signature in the packet before `record_attestation` runs; a single invalid signature fails the whole transaction. `record_attestation` then reads the instructions sysvar to learn what was verified. For every instruction before its own index whose program id is the Ed25519 program, it requires:

1. The instruction has no accounts, `n` is between 1 and 7, the padding byte is 0, and the data holds all `n` offsets records (`InvalidSignatureInstruction`).
2. Every instruction-index field is `0xFFFF`, meaning "this instruction". A packet therefore cannot point the precompile at bytes in another instruction while the program inspects different ones (`InvalidSignatureInstruction`).
3. The 64-byte signature and 32-byte public key lie inside the data (`InvalidSignatureInstruction`).
4. The message size is exactly 200 and the message bytes equal the message the program derived (`WrongAttestationMessage`).
5. The public key is a member of the active `VerifierSet` (`UnknownVerifier`).
6. That member's bit is not already set in the bitmap, across all Ed25519 instructions in the transaction (`DuplicateVerifier`).

After the scan, the number of set bits must reach the set's threshold of 3 (`InsufficientSignatures`). The resulting bitmap is stored as `Attestation.signer_bitmap`.

5.6 record_attestation

5.6.1 Signers, accounts and arguments

The only signer is the payer. Anyone can relay a committee packet; the authority of the record comes from the three committee signatures, which the program checks itself.

#	Account	Constraint
1	<code>payer</code>	Signer, writable; pays rent on first creation
2	<code>config</code>	Config ["config"]
3	<code>verifier_set</code>	VerifierSet ["verifiers", args.epoch]
4	<code>operator</code>	Operator ["op", operator.authority]
5	<code>platform</code>	Platform ["plat", operator, platform_id_hash], has_one operator
6	<code>node</code>	Node ["node", platform, index], has_one platform and operator
7	<code>runtime</code>	Runtime ["rt", SHA-256(measurement)]
8	<code>attestation</code>	Attestation ["att", node], created on first use, writable
9	<code>instructions</code>	Instructions sysvar, exact address
10	<code>slot_hashes</code>	SlotHashes sysvar, exact address
11	<code>system_program</code>	Account creation
12	<code>event_authority</code>	Self-CPI event authority
13	<code>program</code>	The <code>sealroom</code> program

RecordAttestationArgs :

Argument	Type	Bytes
<code>commitment</code>	<code>AttestationCommitment</code>	194
<code>epoch</code>	u64	8
<code>sequence</code>	u64	8
<code>recent_slot</code>	u64	8
<code>recent_slot_hash</code>	[u8; 32]	32

With the 8-byte instruction discriminator, the instruction data is 258 bytes.

5.6.2 Checks, in program order

1. The protocol is not paused (`Paused`).

2. The operator is healthy: not frozen, bond at its tier minimum (`OperatorFrozen` , `InsufficientBond`).
3. The platform is registered (`PlatformNotRegistered`).
4. The node's status is `Registered` (`NodeDisabled`).
5. Every GPU hash on the node is on the platform (`InvalidGpuSet`).
6. The runtime is not revoked and its 48-hour timelock has passed (`RuntimeNotAllowed`).
7. The commitment's measurement equals `Runtime.measurement` and its platform equals the platform account (`AttestationMismatch`).
8. The commitment's TCB status is `UpToDate` and its method is `Committee` (`InvalidTcb`).
9. `now < expires_at <= now + 86,400` , and `expires_at <= unbond_at` when an unbond is pending (`AttestationExpired` , `AttestationBeyondUnbond`).
0. `args.epoch` equals `Config.active_verifier_epoch` and the set's own epoch, and the set is not retired (`InvalidVerifierSet`).
1. The commitment's `operator_epoch` equals `Operator.attestation_epoch` (`AttestationReplay`).
2. The attestation is not revoked (`AttestationRevoked`), and `args.sequence` is 1 for a new account or the stored sequence plus 1 (`AttestationReplay`).
3. (`recent_slot` , `recent_slot_hash`) appears as a pair in `SlotHashes` , which holds at most 512 entries of 40 bytes after an 8-byte count (`InvalidRecentSlot`).
4. The program derives the 200-byte message and verifies the committee packet as in Section 5.5.3.

It then overwrites the node's single `Attestation` account and emits `AttestationRecorded` with every stored field, the signer bitmap and the canonical message hash.

5.6.3 The `Attestation` account

Offsets are from the start of the account. The three optional fields at the end occupy up to 75 bytes; the allocation reserves their maximum.

Offset	Field	Type	Bytes	Written as
0	discriminator		8	
8	<code>version</code>	u8	1	1
9	<code>bump</code>	u8	1	PDA bump
10	<code>node</code>	Pubkey	32	Node PDA
42	<code>sequence</code>	u64	8	<code>args.sequence</code>
50	<code>signed_sequence</code>	u64	8	<code>args.sequence</code>
58	<code>operator_epoch</code>	u64	8	From the commitment
66	<code>runtime_measurement</code>	[u8; 48]	48	From the commitment
114	<code>gpu_rim_hash</code>	[u8; 32]	32	From the commitment

Offset	Field	Type	Bytes	Written as
146	evidence_hash	[u8; 32]	32	From the commitment
178	enclave_key_hash	[u8; 32]	32	From the commitment
210	tcb_status	enum	1	UpToDate
211	slot	u64	8	args.recent_slot
219	recent_slot_hash	[u8; 32]	32	args.recent_slot_hash
251	expires_at	i64	8	From the commitment
259	recorded_at	i64	8	Chain clock
267	method	enum	1	Committee
268	verifier_epoch	u64	8	args.epoch
276	signer_bitmap	u8	1	Bits of the three signers
277	canonical_message_hash	[u8; 32]	32	SHA-256 of the 200-byte message
309	revoked_at	Option<i64>	up to 9	None
	revoked_by	Option<Pubkey>	up to 33	None
	revoke_reason_hash	Option<[u8; 32]>	up to 33	None
Allocation			384	

There is one `Attestation` per node. A renewal overwrites it; the history of every signed revision is kept by the indexer from the `AttestationRecorded` events, keyed by account address and sequence, together with the transaction signature, the signing verifiers and a link to the archived evidence.

5.7 Measured cost

A three-signature `record_attestation` is sent as a v0 transaction with an address lookup table:

Form	Bytes	Compute units
v0 transaction with lookup table	1,056	49,314
Equivalent legacy transaction	1,361	Over the 1,232-byte packet limit

The two instructions carry 532 bytes of Ed25519 packet and 258 bytes of `record_attestation` data. The legacy form spends 32 bytes on each of the 13 account addresses; the lookup table replaces them with one-byte indexes. The coordinator creates the table once, reuses it, and refuses to send any attestation packet over 1,232 bytes.

49,314 compute units is 3.5% of the 1.4 million units a transaction may use. With renewal at least every 24 hours, the committee path costs one such transaction per node per day.

5.8 Sequence, revision and expiry

`sequence` is the replay barrier: every committee record and every revocation consumes the next value. `signed_sequence` is the revision number of the latest committee-signed record, and it is the value that key-release logs, usage vouchers and the API cite. `operator_epoch` is fixed at signing; `expires_at` is at most 86,400 seconds after recording.

Transition	<code>sequence</code>	<code>signed_sequence</code>	<code>revoked_at</code>	<code>expires_at</code>
First record	1	1	None	At most now + 86,400
Renewal	+1	= new <code>sequence</code>	None	New expiry
<code>revoke_attestation</code>	+1	Unchanged	Now	Unchanged
<code>clear_attestation_revocation</code>	Unchanged	Unchanged	None	Now (expired)
First record after a clear	+1	= new <code>sequence</code>	None	New expiry

The registry API reports each attestation as `valid`, `expiring_soon` (3,600 seconds or less remain), `expired` or `revoked`. An attestation counts as revoked when `revoked_at` is set or when its `operator_epoch` is lower than the operator's current `attestation_epoch`. Every consumer compares `expires_at` with the chain's Clock.

5.9 Revocation

5.9.1 `revoke_attestation`

`revoke_attestation(reason_hash: [u8; 32])` takes four accounts: the signing `authority`, `config`, a `verifier_set` and the writable `attestation`. The signer must be `Config.admin` or a member of the supplied `VerifierSet` when that set is the one at `Config.active_verifier_epoch`; anyone else fails with `Unauthorized`.

If the attestation is not already revoked, the instruction increments `sequence`, sets `revoked_at` to now, `revoked_by` to the signer and `revoke_reason_hash` to the supplied hash, and emits `AttestationRevoked`. Revoking an already revoked attestation succeeds and changes nothing. The instruction is not gated by the protocol pause.

5.9.2 Immediate effect

Revocation takes effect at the next read of the account by each consumer:

Consumer	Effect
Key broker	<code>attestation_revoked</code> : no release, including for a request already in progress whose final chain read sees the revocation
Node	Its eligibility check before inference fails, so it serves no new call; its check before settlement fails, so it charges for none
Caller SDK	Sees the attestation as revoked and does not send data
Verifiers	Refuse to sign any proposal for the node (<code>ATTESTATION_REVOKED_ADMIN_CLEAR_REQUIRED</code>)
Program	<code>record_attestation</code> fails with <code>AttestationRevoked</code>

5.9.3 Held packets and the sticky latch

A revocation is sticky. The node cannot clear it by renewing, and a renewal packet that was already signed when the revocation landed can never be recorded:

1. The node's attestation is at sequence 10. The coordinator holds a fully signed renewal for sequence 11 that it has not yet submitted.
2. A verifier revokes. The revocation consumes sequence 11: `sequence = 11` , `revoked_at` set.
3. The held packet for 11 is submitted. It fails with `AttestationRevoked` .
4. The admin or arbitration multisig calls `clear_attestation_revocation` . It clears `revoked_at` , `revoked_by` and `revoke_reason_hash` and sets `expires_at` to now, so the account is valid for no time at all. `sequence` stays 11.
5. The held packet for 11 is submitted again. The program now expects sequence 12 and fails with `AttestationReplay` .
6. The node must produce fresh evidence and the committee must sign a new proposal at sequence 12. That record becomes `signed_sequence = 12` .

`clear_attestation_revocation` is signed by `Config.admin` or `Config.arbitration` and requires the attestation to be revoked (`AttestationNotRevoked`). Clearing never revalidates anything: it only permits a new committee record.

5.9.4 Operator-wide revocation through the epoch

Arbitration can revoke every attestation an operator holds with one field. `rule_dispute` and `arbitrate_operator` accept `revoke_attestations: bool` ; when true, the ruling increments `Operator.attestation_epoch` .

Every attestation carries the epoch it was signed under, inside its signed commitment. After the increment, each of the operator's attestations has an `operator_epoch` below the operator's: the broker treats it as revoked (`attestation_revoked`), the indexer marks it revoked at the ruling's time, and the node's own eligibility check fails. Any held packet signed under the old epoch fails `record_attestation` with `AttestationReplay` . New attestations need fresh evidence signed under the new epoch, which a frozen operator cannot obtain. The Attestation accounts themselves are not touched, so the revocation costs one instruction however many nodes the operator runs.

5.9.5 Runtime revocation

`revoke_runtime` marks a `Runtime` revoked immediately. From then on `record_attestation` refuses that measurement (`RuntimeNotAllowed`), verifiers refuse to sign it (`RUNTIME_NOT_ALLOWED`), and the broker's eligibility rule requires the runtime to be globally allowed as well as pinned by the model (`runtime_not_allowed`). Existing attestations under a revoked runtime stop receiving keys at once.

5.10 Renewal

An attestation lives at most 86,400 seconds; the coordinator proposes `chain now + 86,399`. A node renews by running the pipeline again with a new (`slot`, `hash`) pair. If it keeps its key pairs, `enclave_key_hash` is unchanged while `H` and `evidence_hash` change with the slot hash. The committee signs at `sequence + 1` under the operator's current epoch, and `record_attestation` overwrites `Attestation ["att", node]` with `signed_sequence` at the new revision.

The API flags an attestation as `expiring_soon` in its last hour. A renewal passes every rule a first attestation does, so the operator must still be healthy, the runtime still allowed and the pins intact each day. The node's key release is tied to the revision it was issued under (Section 6.6): after a renewal it requests a release for the new revision before serving again, and the runtime stops serving when its own attestation lapses.

5.11 The ZK verification path

The committee is the v0 verification method. The account layout already carries its successor: `AttestationMethod` has two variants, `Committee` and `Zk`, and the method tag is part of the signed commitment. `record_attestation` accepts only `Committee` (`InvalidTcb` for any other method).

The next method, `record_attestation_zk`, replaces the three committee signatures with one Groth16 proof:

Property	Committee (v0)	ZK
Evidence check	Five verifiers run Intel and NVIDIA verification off chain; three sign	A zkVM program (SP1 or RISC Zero) runs the same verification; its execution is wrapped in a Groth16 proof
On-chain input	Ed25519 packet with three signatures over the 200-byte message	One proof of about 256 bytes, with public inputs that commit to the same fields as the commitment
On-chain verification	Ed25519 precompile; 49,314 compute units for the transaction	Groth16 over the alt_bn128 syscalls: under 200,000 compute units with the groth16-solana crate, about 280,000 per sp1-solana's README, and a measured mean of about 246,906 per transaction in one study
Proving time	Seconds	Minutes; Automata reported about five minutes to prove a DCAP verification on Bonsai

Property	Committee (v0)	ZK
Trust	Three of five verifiers honest	Soundness of the zkVM and the guest program
Resulting account	Attestation with method = Committee	The same Attestation with method = Zk

The proof fits within the 1.4 million compute units of one transaction. Because the public inputs commit to the same fields, every check downstream of the account (the broker's eligibility rules, the caller's key check, revocation by epoch and sequence) reads a ZK attestation exactly as it reads a committee one. The ZK path removes trust in the committee's code; platform pinning and operator vetting remain the basis of hardware trust under both methods.

6. Key management and release

The model owner keeps its weights encrypted everywhere outside an attested enclave. One 256-bit data key per model opens them, and that key leaves the owner's key broker only as an HPKE ciphertext addressed to a specific enclave's key, after the broker has checked the chain, the evidence and the model's policy for itself. Every decision, release or denial, is logged with the attestation it relied on.

6.1 Weight packaging

The owner packages a model once:

1. Generate a random 256-bit data key for the model.
2. Encrypt the weights with AES-256-GCM in chunks. Each chunk is authenticated, so a modified chunk fails to decrypt.
3. Write a manifest describing the encrypted package and compute its 32-byte hash.
4. Upload the ciphertext and the manifest to any storage, public storage included; the ciphertext is unreadable without the data key.
5. Place the data key in the broker's key store (Section 6.2).
6. Sign `create_model` with the manifest hash, the broker's identity and the release policy.

The manifest hash pins exactly which ciphertext the model refers to. A node fetches the manifest, checks it against `Model.manifest_hash`, and decrypts chunk by chunk with the released data key.

The Model account fields that govern release:

Field	Type	Bound	Role in release
<code>owner</code>	Pubkey		PDA seed; signs model changes
<code>id</code>	u64		PDA seed: <code>Model ["model", owner, id]</code>
<code>manifest_hash</code>	[u8; 32]		Identifies the encrypted package
<code>key_broker</code>	Pubkey	Not the default key	The only broker that may release this model's key
<code>key_broker_url</code>	String	HTTPS, at most 128 bytes, same rules as node endpoints	Where nodes send key requests
<code>allowed_runtimes</code>	Vec<[u8; 48]>	1 to 8, distinct	Measurements the owner accepts
<code>min_tier</code>	u8	1 to 3	Lowest operator tier

Field	Type	Bound	Role in release
<code>min_bond</code>	u64	USDC base units	Lowest operator bond; the larger of this and the tier minimum applies
<code>operator_allowlist</code>	Vec<Pubkey>	At most 16, distinct, non-default	Empty: any admitted operator; otherwise only listed operators
<code>derivative_royalty_bps</code>	u16	0 to 10,000	Royalty this model requires from derivatives (Section 6.8)
<code>base_model</code>	Option<Pubkey>		The base this model derives from

`create_model` validates every bound (`BoundsExceeded` , `InvalidBps` , `InvalidEndpoint`). The full `Model` account, with price and split, is 2,348 bytes; Chapter 7 gives its layout.

6.2 The owner-run key broker

The key broker is an open-source service that the owner runs, or delegates to a party it chooses. Sealroom does not hold model keys.

Aspect	Behavior
Identity	An Ed25519 public key recorded as <code>Model.key_broker</code> ; the broker serves only models that name it
Endpoint	<code>POST /v1/key-requests</code> at <code>key_broker_url</code> ; authenticated server requests only
Chain reads	Directly from Solana RPC at <code>confirmed</code> commitment, with account ownership and IDL decoding checked; never from the indexer's database
Deployment pin	Expected genesis hash and program id
Data keys	Loaded per request from the owner's key store or HSM; exactly 32 bytes; zeroed after wrapping
Vendor verification	Its own Intel and NVIDIA verification, independent of the committee
Audit store	Append-only release log, written before any ciphertext is returned

A node contacts the broker after boot and after each renewal of its attestation; between those, it serves with the release it holds.

The owner-run broker is the v0 default. As an opt-in for owners who prefer not to run one, the v1 design adds a Sealroom threshold committee: the data key split into 3-of-5 Shamir shares held in HSMs at independent firms, each re-verifying the evidence before contributing its share.

6.3 The key request

The node sends a strict JSON body with three fields: `requestId`, a fresh UUID; `modelId`, the `Model` address; and `evidence`, the same `sealroom-evidence-v1` object the committee attested.

Outcome	HTTP	Body
Released	201	{ <code>auditId</code> , <code>attestationId</code> , <code>attestationRevision</code> , <code>wrappedKey</code> }
Denied	403	FORBIDDEN, not retryable, with <code>reasonCodes</code> and the <code>auditId</code> of the logged denial
Unavailable	503	SERVICE_UNAVAILABLE, retryable, with a reason code

`attestationRevision` is the attestation's `signed_sequence`. The request UUID never appears in any public record; the public audit identity is a separate random value.

6.4 Release checks

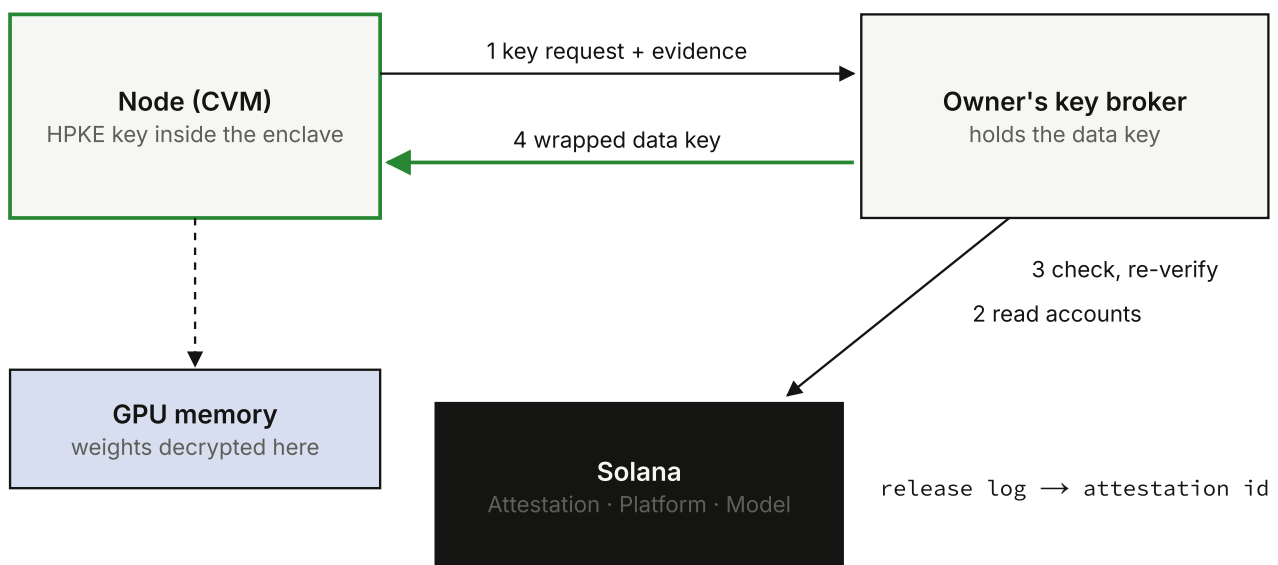


Figure: The broker reads the chain, checks the evidence against the on-chain attestation, re-verifies it, wraps the data key to the enclave's HPKE key, reads the chain again and logs the decision before returning the ciphertext.

6.4.1 The release procedure

The broker processes each request in this order:

1. **Parse.** Validate the request, compute `evidence_hash`, and draw a fresh audit id (32 random bytes, base58).

2. **First chain read.** In one RPC call, read `Config`, `Node`, `Platform`, `Operator`, `Attestation` `["att", node]`, the `Clock` and `SlotHashes` sysvars, the active `VerifierSet`, the `Model`, the attested `Runtime`, the `PlatformPin` and every `GpuPin`. If the node's links or the verifier epoch change between the preview and the full read, retry, up to three times. The genesis hash must be the configured one (`CHAIN_IDENTITY_MISMATCH`).
3. **Policy checks.** Run the twelve eligibility rules (Section 6.4.2); require `Model.key_broker` to be this broker (`BROKER_AUTHORITY_MISMATCH`); and require `evidence_hash`, the SHA-256 of the evidence's key transcript, `recentSlot` and `recentSlotHash` to equal the attestation's `evidence_hash`, `enclave_key_hash`, `slot` and `recent_slot_hash` (`EVIDENCE_CHAIN_MISMATCH`). Any failure denies with every failing reason.
4. **Fix the release identity:** genesis hash, attestation address, `sequence`, `signed_sequence`, `evidence_hash`, broker key and policy hash as read.
5. **Re-verify** the TD quote against Intel collateral and every GPU report against NVIDIA RIMs and OCSP.
6. **Registry checks on fresh claims.** Repeat the committee's registry checks (Section 5.3) against the broker's own vendor claims. The `SlotHashes` presence check is omitted: freshness was proved when the attestation was recorded, and its expiry bounds its age.
7. **Claims against chain.** The verified runtime measurement and RIM hash must equal `Attestation.runtime_measurement` and `Attestation.gpu_rim_hash` (`VENDOR_CLAIMS_CHAIN_MISMATCH`).
8. **Build the context** for the key release (Section 6.5.2).
9. **Load and wrap** the model's 32-byte data key with HPKE to `evidence.keys.hpkePublicKey`, with the context as associated data.
10. **Final chain read.** Every check of step 3 must still pass and the release identity must be unchanged. A revocation, a freeze, a slash below the minimum or a renewal between the two reads denies the release (`REGISTRY_CHANGED_DURING_RELEASE`).
1. **Log** the audit record with outcome `released`. If the write fails, nothing is returned.
2. **Return** the wrapped key with the audit id, attestation address and revision, and zero the broker's copy of the data key.

On any failure, the broker re-reads the chain once if it has not completed the final read, logs the denial with the policy reasons from its latest read plus the error that stopped the procedure, and returns 403. Every request is checked from the beginning; a repeated request never reuses an earlier decision.

6.4.2 Eligibility rules

The broker runs the same eligibility function that the public API serves at `/v1/eligibility?modelId={modelId}&nodeId={nodeId}`, so anyone can compute a broker's policy decision in advance from public data. The function evaluates twelve checks, in this order, and reports every one that fails:

#	Code	Passes when
1	<code>no_valid_attestation</code>	An attestation exists and is the node's current one

#	Code	Passes when
2	<code>attestation_expired</code>	<code>expires_at</code> is after the chain's current time
3	<code>attestation_revoked</code>	<code>revoked_at</code> is unset and <code>Attestation.operator_epoch</code> equals <code>Operator.attestation_epoch</code>
4	<code>runtime_not_allowed</code>	The <code>Runtime</code> account is allowed (not revoked, past its timelock) and the attested measurement is in <code>Model.allowed_runtimes</code>
5	<code>platform_not_registered</code>	The platform is registered, belongs to the node's operator and is the node's platform; the attestation names the same node, platform and operator
6	<code>tier_below_min</code>	<code>Operator.tier</code> $\geq$ <code>Model.min_tier</code>
7	<code>bond_below_min</code>	<code>Operator.bond_amount</code> $\geq$ <code>max(Model.min_bond, tier_minimum)</code>
8	<code>operator_frozen</code>	<code>Operator.frozen</code> is false
9	<code>operator_not_on_allowlist</code>	<code>Model.operator_allowlist</code> is empty or contains the operator
10	<code>tcb_not_up_to_date</code>	<code>tcb_status</code> is <code>UpToDate</code>
11	<code>node_disabled</code>	The node's status is not <code>Disabled</code>
12	<code>protocol_paused</code>	<code>Config.paused</code> is false

6.4.3 Broker reason codes

Beyond the twelve eligibility codes and the registry codes of Section 3.6.2, a denial can carry:

Code	Meaning
<code>MODEL_UNAVAILABLE</code>	The <code>Model</code> account could not be read
<code>BROKER_AUTHORITY_MISMATCH</code>	The model names a different broker key
<code>EVIDENCE_CHAIN_MISMATCH</code>	Evidence hash, enclave key hash, slot or slot hash differs from the attestation
<code>VENDOR_CLAIMS_CHAIN_MISMATCH</code>	Verified measurement or RIM hash differs from the attestation
<code>MODEL_KEY_NOT_CONFIGURED</code>	The key store holds no valid 32-byte key for the model
<code>REGISTRY_CHANGED_DURING_RELEASE</code>	The final read differs from the first
<code>REGISTRY_RECHECK_UNAVAILABLE</code>	The re-read after a failure could not complete

6.5 HPKE wrapping to the enclave's key

6.5.1 The suite

Parameter	Value
Mode	HPKE Base mode (RFC 9180)

Parameter	Value
KEM	X-Wing, the hybrid of ML-KEM-768 and X25519 (draft-connolly-cfrg-xwing-kem-06)
KDF	HKDF-SHA256
AEAD	AES-256-GCM
Suite identifier	X-Wing-draft06-HKDF-SHA256-AES256GCM
info	sealroom.owner-key-release.v1
Recipient public key	evidence.keys.hpkePublicKey : 1,216 bytes, the ML-KEM-768 public key (1,184) followed by the X25519 public key (32)
Plaintext	The 32-byte data key
Associated data	The key-release context
Sender context	One fresh context per wrap

The hybrid KEM derives the shared secret from both ML-KEM-768 and X25519, so recovering the data key from a recorded envelope requires breaking both.

The broker trusts the recipient key because its hash is the on-chain `enclave_key_hash` (step 3), and the same transcript produced the H carried in REPORTDATA and every GPU nonce (step 6) of a quote from an allowed runtime on a pinned platform. The private key was generated inside that runtime and never leaves the CVM.

6.5.2 The key-release context

The associated data is the UTF-8 JSON encoding of:

Field	Value
domain	sealroom.key-release.v1
programId	The sealroom program id
genesisHash	The cluster's genesis hash
modelId	Model address
nodeId	Node address
attestationId	Attestation ["att", node] address
attestationRevision	signed_sequence of the attestation used
evidenceHash	evidence_hash of the request's evidence
requestId	The node's request UUID

The envelope returned to the node has five fields: `version` (`sealroom-key-wrap-v1`), `suite` , and the base64 `enc` (KEM encapsulation), `ciphertext` (the sealed data key) and `context` (the

associated data).

6.5.3 Unwrap inside the CVM

The runtime's key client:

1. Authenticates the broker's TLS identity against the broker recorded for the model in the registry, before accepting the response.
2. Rebuilds the expected context from its own values (program, genesis hash, model, node, its evidence hash and request id) and the attestation address and revision in the response, and requires the envelope's `context` to equal it byte for byte.
3. Opens the ciphertext with its HPKE private key, the same `info` and the context as associated data.
4. Requires a 32-byte plaintext.

Any failure in steps 2 to 4 yields one generic error, `KEY_UNWRAP_FAILED`. The client then fetches the manifest, checks it against `Model.manifest_hash`, decrypts the weights chunk by chunk in CVM memory and loads them into GPU protected memory.

6.6 Holding and using the key

The node keeps one release per model, bound to the attestation revision it was issued under. Before each inference it checks that its release revision equals the attestation's current `signed_sequence` and that the model is still eligible on the node; it checks eligibility again before settling the payment. After a renewal it requests a release for the new revision. After a revocation, a freeze or an epoch advance, both checks fail and the node neither serves nor charges.

The runtime's network egress is limited to the key broker, the Solana RPC and the payment facilitator, and it serves only architectures its pinned vLLM build supports, with `trust_remote_code` off, so no owner-supplied code runs next to the weights and prompts.

6.7 The release log

6.7.1 Record schema

The broker appends one record per attempt, released or denied:

Field	Type	Content
<code>id</code>	base58, 32 bytes	Audit id, random per attempt
<code>modelId</code>	base58	<code>Model</code> address
<code>nodeId</code>	base58	<code>Node</code> address
<code>operatorId</code>	base58	<code>Operator</code> address observed

Field	Type	Content
brokerPublicKey	base58	This broker's key
attestationId	base58	Attestation address relied on
attestationRevision	u64 string	signed_sequence relied on
evidenceHash	64 hex	Hash of the request's evidence
enclaveKeyHash	64 hex	SHA-256 of the request's key transcript
policyHash	64 hex	SHA-256 of the model policy observed (below)
readSlot	u64 string	Slot of the chain read the decision used
releasedAt	ISO 8601	Decision time
outcome	enum	released or denied
reasonCodes	string array	Empty for a release; every failing code for a denial
audit.chainMatch	bool	Whether the evidence matched the on-chain attestation
audit.checkedAt	ISO 8601	Time of the check

The policy hash is taken over the JSON encoding of the model's release policy as read from chain:

```
policyHash = SHA-256(json({ allowedRuntimes, minTier, minBond,
operatorAllowlist })))
```

6.7.2 Rules

A ciphertext leaves the broker only after its `released` record is persisted. Every attempt has its own audit id, and a denial is logged as durably as a release, with the revision and policy from the latest chain read. The log holds hashes and identifiers only: data keys, private keys, raw evidence, wrapped ciphertexts and request UUIDs never enter it. The log is public at `/v1/key-releases`, filterable by outcome, and each record also appears in the registry's activity feed as `key_released` or `key_denied` with source `key_broker`.

6.7.3 Auditing against the chain

Each record names everything needed to check it against the chain without trusting the broker. For a `released` record, an auditor:

1. Finds the `AttestationRecorded` event for `attestationId` at `attestationRevision` and checks that its `evidence_hash` and `enclave_key_hash` equal the record's.
2. Checks that the attestation had not expired at `releasedAt` and had no `AttestationRevoked` event or operator-epoch advance before `readSlot`.
3. Recomputes the model's policy hash from the `Model` account and compares it with `policyHash`.
4. Re-evaluates the twelve eligibility rules at `readSlot`.

Comparing the broker log with the chain this way is how Sealroom measures that no key is released outside policy.

6.8 Base-model royalties through key gating

6.8.1 What the program enforces

A base model sets `derivative_royalty_bps`, the share it requires from every model built on it. A derivative, such as a LoRA adapter trained on an encrypted base model, names its base in `base_model`. Split entries carry a role (`Owner`, `Operator`, `BaseModel` or `Contributor`) and a `locked` flag. At `create_model` with `base_model = Some(base)`:

1. The base `Model` account must be supplied and equal `base (InvalidBaseModel)`. Without a `base_model`, no base account may be supplied.
2. If the base's `derivative_royalty_bps` is above zero, the derivative's split must contain an entry with role `BaseModel`, `locked = true`, `bps >= base.derivative_royalty_bps`, paying the canonical USDC associated token account of the base's owner, supplied and verified as a token account owned by that owner (`RoyaltyRequired`).

At `propose_split`, every locked entry of the current split must reappear with the same recipient, beneficiary and role, still locked, with at least the same bps, unless the entry's beneficiary signs the proposal (`LockedSplit`). The base owner's royalty therefore cannot be lowered, unlocked or removed without the base owner's signature. Each lane snapshots its model's split, and `distribute` pays the `BaseModel` entry from every lane of the derivative.

6.8.2 The key gate

The base model's weights are encrypted under the base owner's own data key, held by the base owner's broker. A node serving the derivative needs two releases: the derivative's key from the derivative owner's broker, and the base key from the base owner's broker. The base owner's broker releases the base key only for derivatives whose split pays the base's `derivative_royalty_bps` as a locked entry. The program already guarantees that any `Model` naming the base in `base_model` carries that entry, so the broker's royalty check reduces to reading the derivative's `base_model` and confirming the locked `BaseModel` entry is present in its split.

6.8.3 Worked example

The protocol fee is 500 bps at launch. Take a base owner that sets `derivative_royalty_bps = 1000`, a derivative split that gives the base exactly that, and a lane that receives 100 USDC (100,000,000 base units) of caller payments:

```
fee = floor(1000000000 * 500 / 10000) = 50000000
net = 1000000000 - 50000000 = 950000000
base_royalty = floor(950000000 * 1000 / 10000) = 95000000
```

The base owner receives 9.5 USDC from this distribution, in the same transaction that pays the protocol fee, the operator and the derivative's other recipients. Chapter 9 gives the complete distribution arithmetic, including the rounding remainder paid to the final entry.

7. The `sealroom` program

Sealroom keeps its registry and its money in one Solana program, `sealroom`, written with Anchor 0.31.1 and deployed through the upgradeable BPF loader. The program records operators and their bonds, the physical platforms and GPUs each operator has pinned, the nodes that run on them, the runtime measurements that are allowed, the attestations the verifier committee has signed, and the models that owners list with their policy, price and split. It holds four kinds of funds: operator bonds, the USDC that callers pay into per-(model, node) lane vaults, dispute deposits, and the protocol fee in the buyback vault. Each sits in a token account whose authority is a program-derived address, and leaves it only through the instructions described here.

The published interface (IDL `sealroom`, version 0.2.0) contains 45 instructions, 14 account types, 40 event types and 58 program error codes. This chapter documents all of them from the program source: the account layouts byte by byte, every instruction with its signers, constraints, state changes, events and failure codes, the fixed durations that govern change, the deployment and upgrade model, and the measured transaction sizes and compute costs.

7.1 Design principles

One program, one configuration. All Sealroom state lives in accounts owned by the `sealroom` program, anchored by a single `Config` account at the seed `config`. `Config` fixes the USDC mint, the genesis hash of the cluster, the admin and arbitration authorities, the protocol fee, the bond minimum for each operator tier, the dispute settings, the buyback mint and authority, and the pause flag. Every instruction that reads configuration derives `Config` from that seed and its stored bump, so a transaction cannot substitute a configuration account of its own.

Program-derived addresses for every record. All fourteen account types are PDAs, and their seeds encode their relationships: a Node is derived from its Platform and an index, a Lane from a Model and a Node, a Claim from a Lane and a recipient token account, a Dispute from an Operator and a sequence number. Instructions re-derive each address from the account's own stored fields and check parent links with Anchor `has_one` constraints, so an account from another operator, model or lane fails before the handler runs. Uniqueness comes from creation with `init`: a second PlatformPin for the same platform identity, a second Lane for the same pair or a second Dispute with the same sequence number cannot exist.

USDC through the original SPL Token program. Every USDC amount is a `u64` count of base units of a six-decimal mint; `initialize` rejects a mint with any other decimals. Every USDC transfer the program signs is an SPL Token `transfer_checked` against the `Config` mint with six decimals, so a token account of another mint fails inside the token program. The one USDC movement it does not issue itself, the swap leg of the buyback, runs inside a Jupiter CPI that the program signs for and bounds by the vault's balance before and after. Buyback totals of the bought token count base units

of the buyback mint, at that mint's decimals. Fee and split arithmetic multiplies in `u128` before dividing, every counter uses checked addition and subtraction, and no value anywhere in the program is a floating-point number.

Canonical associated token accounts. Where an account holds funds, its address follows from derivation. The bond vault is the associated token account (ATA) of the Operator PDA for USDC, a lane vault is the ATA of the Lane PDA, a dispute deposit vault is the ATA of the Dispute PDA, and the buyback vault is the USDC ATA of the protocol PDA `["protocol"]`, whose ATA for the buyback mint receives the swap output. Recipients are held to the same rule: each split entry's `recipient_ata` must be the canonical USDC ATA of its `recipient_owner`, an operator's `payout_ata` must be the canonical ATA of the operator authority, and the dispute forfeiture account must be a canonical ATA whose owner is recorded at the moment it is bound. Deferred claims pay into the beneficiary's canonical ATA unless governance selects the alternative destination policy.

Custody by the program, with no admin path to user funds. Funds leave a vault only when the program signs with that vault's PDA seeds, and every such path names its destination by rule:

Vault	Token authority	Instructions that move funds out	Destination
Bond vault	Operator PDA <code>["op", authority]</code>	<code>withdraw_bond; settle_slash</code>	A USDC account owned by the operator authority; the deposit vault of a ruled dispute
Lane vault	Lane PDA <code>["lane", model, node]</code>	<code>distribute; claim_deferred; claim_protocol_fee</code>	The buyback vault and the lane's resolved split recipients; the beneficiary of a Claim; the buyback vault for a protocol-fee Claim
Deposit vault	Dispute PDA <code>["dispute", operator, sequence]</code>	<code>claim_dispute; claim_forfeited_deposit</code>	The claimant; the forfeiture beneficiary recorded in the Dispute
Buyback vault	Protocol PDA <code>["protocol"]</code>	<code>buyback_swap_and_burn</code>	A Jupiter v6 route into the protocol PDA's output account, whose entire increase is burned in the same instruction

The admin key signs none of these instructions. Its influence over destinations is limited to the forfeiture recipient, which changes only through a 48-hour proposal that anyone can read on chain before it takes effect, to the claim destination policy, which under either setting pays only accounts owned by the beneficiary, and to the buyback settings: it names the buyback authority and selects the buyback mint once. The buyback vault's address follows from the program id and the USDC mint alone. The buyback authority signs `buyback_swap_and_burn` but holds nothing: the program signs for the protocol PDA inside the swap and burns every token the swap returns before the instruction ends.

Timelocks on changes that redirect value or trust, immediate effect for protective actions.

Allowing a runtime, changing the fee rate, changing dispute settings and changing a model's split all wait a fixed delay compiled into the program. Revoking a runtime, revoking an attestation, freezing an operator and pausing the protocol take effect in the slot they land, and so do an owner's `update_price` and `update_policy`, which change what new calls cost and which nodes qualify without moving any funds.

Fail-closed configuration. Configuration values that represent a policy decision are `Option` fields, and `None` means the decision has not been made. Instructions that depend on them refuse to run until they are set: distribution and claims require a claim policy, dispute admission requires a deposit, an arbitration authority, a deposit policy and a per-claimant cap, a positive arbitration slash requires a configured deposit, and the buyback requires a buyback authority and a selected buyback mint.

Separated authorities. The admin and the arbitration authority must be different keys at initialization, at every dispute-configuration proposal and again when that proposal is applied. The arbitration authority must also differ from the owner of the forfeiture account, so the party that dismisses a complaint is never the party that receives the forfeited deposit.

Permissionless cranks. `apply_fee`, `apply_dispute_config`, `apply_split`, `open_lane`, `prepare_claim`, `distribute`, `claim_protocol_fee` and `settle_slash` take no privileged signer, and `record_attestation` accepts any relayer as fee payer. Their outcome is determined entirely by on-chain state and by the arguments, which the program validates, so the keeper service, a model owner or any third party can submit them.

Bounded accounts. Each account is allocated at its maximum size when it is created, with every vector and string bounded by `max_len`, and instructions check the same bounds before writing. No account is reallocated. The largest, Model, is 2,348 bytes.

Events through self-CPI. Every instruction except `initialize` emits its events with Anchor's `emit_cpi!`, which records the event bytes as an inner instruction signed by the `__event_authority` PDA. Indexers read those bytes from the transaction's inner instructions, so events survive the 10 KB limit on textual program logs. `initialize` emits its single event to the program log and is reconciled from the Config account it creates.

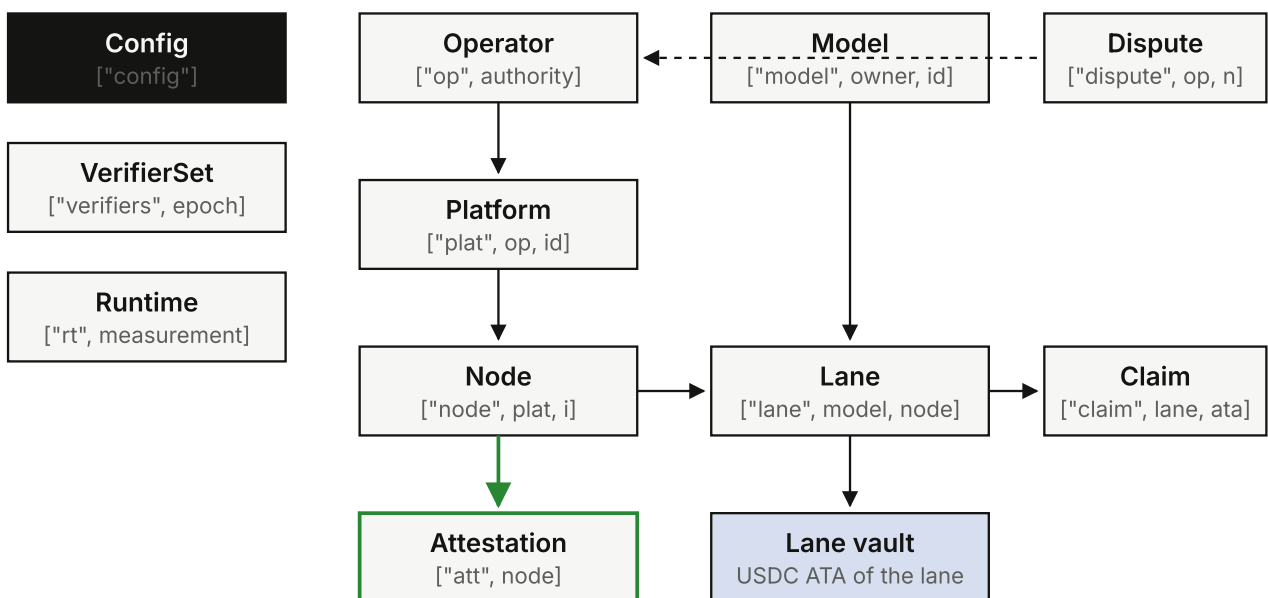
7.2 Accounts

7.2.1 Encoding

Every account begins with Anchor's eight-byte discriminator, followed by `version: u8` and `bump: u8`. The version is 1 for every account written by this program. Fields are Borsh-encoded in declaration order: public keys and hashes are raw 32-byte arrays, the runtime measurement is a raw 48-byte array, amounts, identifiers, slots and sequence numbers are little-endian `u64`, timestamps are little-endian `i64` Unix seconds, basis points are `u16`, and enum variants are one-byte tags. An `Option<T>` takes one tag byte plus the size of `T`, a vector takes a four-byte length prefix plus its maximum element count times the element size, and a string takes a four-byte prefix plus its maximum byte length. The sizes below are the allocated maximums, including the discriminator.

Account	Seeds before bump	Bytes	Created by	Rent payer
Config	"config"	676	<code>initialize</code>	Admin
VerifierSet	"verifiers", epoch <code>u64</code> LE	264	<code>initialize</code>	Admin
Operator	"op", operator authority	193	<code>register_operator</code>	Admin

Account	Seeds before bump	Bytes	Created by	Rent payer
Platform	"plat", Operator PDA, platform id hash	343	register_platform	Operator authority
PlatformPin	"platform_pin", platform id hash	106	register_platform	Operator authority
GpuPin	"gpu_pin", GPU UEID hash	107	register_platform	Operator authority
Node	"node", Platform PDA, index u16 LE	503	register_node	Operator authority
Attestation	"att", Node PDA	384	record_attestation	Relayer
Runtime	"rt", SHA-256(measurement)	256	allow_runtime	Admin
Model	"model", owner, id u64 LE	2,348	create_model	Model owner
Lane	"lane", Model PDA, Node PDA	719	open_lane	Any payer
Claim	"claim", Lane PDA, recipient ATA	122	prepare_claim	Any payer
Dispute	"dispute", Operator PDA, sequence u64 LE	415	open_dispute , arbitrate_operator	Claimant or arbitration
ClaimantDisputes	"claimant_disputes", claimant, Operator PDA	78	open_dispute , arbitrate_operator	Claimant or arbitration



Seeds in brackets. Platform and GPU pins, one per identity hash, stop the same machine or GPU from being registered twice.

Figure: The fourteen program accounts, the seeds that derive them and the parent links that instructions check.

One further program address holds no account data: the protocol PDA ["protocol"] . It has no private key and is the token authority of the buyback vault, its canonical USDC ATA derived with the associated-token program under the original SPL Token program, and of the output account, its canonical ATA for the buyback mint derived under that mint's own token program (7.3.10).

Several value types are embedded inside accounts:

Type	Layout	Bytes
SplitEntry	recipient_ata 32, recipient_owner 32, bps: u16 2, locked: bool 1, role: SplitRole 1	68
PendingSplit	revision: u64 8, proposed_at: i64 8, effective_at: i64 8, entries: Vec<SplitEntry> (max 8) 548	572
PendingFee	protocol_fee_bps: u16 2, treasury_ata 32, treasury_owner 32 (retained treasury binding), effective_at: i64 8	74
PendingDisputeConfig	arbitration 32, dispute_deposit: u64 8, max_open_per_claimant_operator: u32 4, max_open_per_operator: u32 4, forfeiture_ata 32, forfeiture_owner 32, deposit_policy 1, effective_at: i64 8	121
TierBondMinimum	tier: u8 1, amount: u64 8	9
Price	tag 1, largest variant (three u64) 24	25

The enumerations are:

Enum	Variants
GpuModel	H100 , H200
NodeStatus	Registered , Disabled
TcbStatus	UpToDate , OutOfDate , ConfigurationNeeded , Revoked , Unknown
AttestationMethod	Committee , Zk
SplitRole	Owner , Operator , BaseModel , Contributor
Price	Exact { per_call }, Upto { input_per_million_tokens, output_per_million_tokens, max_per_call }
DisputeStatus	Open , RuledForClaimant , Dismissed
DisputeDepositPolicy	ReturnToClaimant (not selectable), ForfeitOnDismissal
ClaimPolicy	BeneficiaryOnlyNoExpiry
ClaimDestinationPolicy	CanonicalAta , AnyOwnedUsdc
SettlementMechanism	LaneBoundary

Enum	Variants
RescuePolicy , SplitSettlementPolicy	ThreePartyOwnerRefund , DistributionTime (retained in the interface, never stored)

7.2.2 Config

Seeds ["config"]. One per deployment, created by initialize . It holds every protocol-wide parameter, the authorities and the two governance proposals that can be pending at once.

Field	Type	Bytes	Meaning
discriminator	[u8; 8]	8	Anchor account discriminator
version	u8	1	Layout version, 1
bump	u8	1	PDA bump
genesis_hash	[u8; 32]	32	Genesis hash of the cluster, bound into every committee message
usdc_mint	Pubkey	32	The USDC mint (six decimals)
admin	Option<Pubkey>	33	Admin authority, the upgrade authority that signed initialize
arbitration	Option<Pubkey>	33	Arbitration authority
fee_treasury_ata	Option<Pubkey>	33	Retained treasury binding, written by initialize and apply_fee; distribute pays the buyback vault and does not read it
protocol_fee_bps	Option<u16>	3	Protocol fee in basis points; 500 at launch
default_operator_share_bps	Option<u16>	3	Reserved; written as None
dispute_deposit	Option<u64>	9	Complaint deposit in base units
mainnet_total_bonds_cap	Option<u64>	9	Reserved aggregate bond cap; written as None
mainnet_total_vaults_cap	Option<u64>	9	Reserved aggregate vault cap; written as None
tier_bond_minimums	Vec<TierBondMinimum> , max 3	31	Minimum bond for tiers 1, 2 and 3
attestation_ttl_seconds	u64	8	86,400
unbonding_period_seconds	u64	8	2,592,000
split_delay_seconds	u64	8	604,800
runtime_timelock_seconds	u64	8	172,800

Field	Type	Bytes	Meaning
active_verifier_epoch	Option<u64>	9	Epoch of the verifier set whose signatures count; 0 at initialization
paused	bool	1	Pause flag
fee_treasury_owner	Option<Pubkey>	33	Owner of fee_treasury_ata when it was bound; not read by distribute or any claim
pending_fee	Option<PendingFee>	75	Proposed fee rate, with its treasury binding, and its effective time
claim_policy	Option<ClaimPolicy>	2	BeneficiaryOnlyNoExpiry once selected
rescue_policy	Option<RescuePolicy>	2	Retained; always None
split_settlement_policy	Option<SplitSettlementPolicy>	2	Retained; always None
dispute_deposit_policy	Option<DisputeDepositPolicy>	2	ForfeitOnDismissal once dispute settings apply
claim_destination_policy	ClaimDestinationPolicy	1	CanonicalAta at initialization, or AnyOwnedUsdc
pending_dispute_config	Option<PendingDisputeConfig>	122	Proposed dispute settings with their effective time
max_open_per_claimant_operator	Option<u32>	5	Cap on simultaneously open disputes per (claimant, operator)
max_open_per_operator	u32	4	Cap on simultaneously open disputes per operator; 8 at initialization
dispute_forfeiture_ata	Option<Pubkey>	33	Canonical ATA that receives dismissed deposits
dispute_forfeiture_owner	Option<Pubkey>	33	Owner of that ATA, the forfeiture beneficiary
settlement_mechanism	SettlementMechanism	1	LaneBoundary
buyback_mint	Option<Pubkey>	33	Token the buyback buys and burns; None until set_buyback_mint, then immutable
buyback_authority	Option<Pubkey>	33	Key that may sign buyback_swap_and_burn; set by set_buyback_authority
usdc_spent_total	u64	8	USDC base units spent by every completed buyback
burned_total	u64	8	Base units of the buyback mint burned by every completed buyback
Total		676	

The four duration fields record the compiled constants for readers; handlers apply the constants themselves, so no instruction can shorten a delay by editing configuration. The buyback vault and output account are not stored: every instruction that uses them derives them from the protocol PDA, the USDC mint and the buyback mint.

7.2.3 VerifierSet

Seeds `["verifiers", epoch.to_le_bytes()]`. The committee whose Ed25519 signatures admit attestations. `initialize` creates the epoch-0 set with exactly five distinct keys and threshold three; the allocation holds up to seven members.

Field	Type	Bytes	Meaning
discriminator, <code>version</code> , <code>bump</code>		10	Header
<code>epoch</code>	<code>u64</code>	8	Committee epoch
<code>members</code>	<code>Vec<Pubkey></code> , max 7	228	Verifier Ed25519 public keys; a member's index is its bit in an attestation's <code>signer_bitmap</code>
<code>threshold</code>	<code>u8</code>	1	Signatures required, 3
<code>activated_at</code>	<code>i64</code>	8	Activation time
<code>retired_at</code>	<code>Option<i64></code>	9	Retirement time; a retired set admits nothing
Total		264	

7.2.4 Operator

Seeds `["op", authority]`. One per vetted operator, created by the admin after off-chain vetting. The PDA is also the token authority of the operator's bond vault.

Field	Type	Bytes	Meaning
discriminator, <code>version</code> , <code>bump</code>		10	Header
<code>authority</code>	<code>Pubkey</code>	32	Operator wallet
<code>agreement_hash</code>	<code>[u8; 32]</code>	32	Hash of the signed Operator Agreement
<code>tier</code>	<code>u8</code>	1	Tier 1, 2 or 3
<code>bond_vault</code>	<code>Pubkey</code>	32	ATA of this PDA for USDC
<code>bond_amount</code>	<code>u64</code>	8	Withdrawable bond principal; excludes reserved slashes and direct donations to the vault
<code>unbond_at</code>	<code>Option<i64></code>	9	Earliest withdrawal time once unbonding is requested
<code>frozen</code>	<code>bool</code>	1	Frozen by admin or arbitration

Field	Type	Bytes	Meaning
payout_ata	Pubkey	32	Canonical USDC ATA of authority ; destination of every Operator-role share
open_dispute_count	u32	4	Open ordinary disputes against this operator
next_dispute_sequence	u64	8	Sequence number the next Dispute must use
registered_at	i64	8	Registration time
attestation_epoch	u64	8	Revocation epoch committed into every attestation of this operator
pending_slash_amount	u64	8	Slashed amount reserved in the bond vault and not yet settled
Total		193	

7.2.5 Platform, PlatformPin and GpuPin

A Platform is one physical server identity (the hash of its Intel PPID or PCK identity) registered by one operator, with the UEID hashes of the GPUs installed in it. Seeds ["plat", operator, platform_id_hash] .

Field	Type	Bytes	Meaning
discriminator, version, bump		10	Header
operator	Pubkey	32	Operator PDA
platform_id_hash	[u8; 32]	32	Hash of the platform identity
gpu_ueid_hashes	Vec<[u8; 32]> , max 8	260	Hashes of the GPU UEIDs pinned to this platform
registered	bool	1	Registration flag, checked by node registration and attestation
registered_at	i64	8	Registration time
Total		343	

The Platform seeds include the operator, so platform identity is made unique across operators by a second, global account. A PlatformPin at ["platform_pin", platform_id_hash] is created with init in the same instruction, so a platform identity can be registered exactly once across all operators.

Field	Type	Bytes	Meaning
discriminator, version, bump		10	Header
platform	Pubkey	32	Platform PDA that owns this identity
operator	Pubkey	32	Operator PDA

Field	Type	Bytes	Meaning
platform_id_hash	[u8; 32]	32	The pinned identity hash
Total		106	

A GpuPin at ["gpu_pin", gpu_ueid_hash] pins one GPU globally to one platform and, once a node claims it, to one node. GpuPins are created and read by the program directly from the instruction's remaining accounts.

Field	Type	Bytes	Meaning
discriminator, version, bump		10	Header
platform	Pubkey	32	Platform the GPU belongs to
gpu_ueid_hash	[u8; 32]	32	Hash of the GPU UEID
node	Option<Pubkey>	33	Node that claimed the GPU, or None
Total		107	

7.2.6 Node

Seeds ["node", platform, index.to_le_bytes()] with a u16 index. One confidential VM on a registered platform, with one to eight of the platform's GPUs.

Field	Type	Bytes	Meaning
discriminator, version, bump		10	Header
platform	Pubkey	32	Platform PDA
operator	Pubkey	32	Operator PDA
index	u16	2	Index of the node on its platform
node_key	Pubkey	32	Node's registered public key
endpoint	String, max 128	132	HTTPS endpoint of the node
gpu_model	GpuModel	1	H100 or H200
gpu_count	u8	1	Number of GPUs claimed
gpu_ueid_hashes	Vec<[u8; 32]>, max 8	260	GPUs claimed by this node
status	NodeStatus	1	Registered or Disabled
Total		503	

7.2.7 Attestation

Seeds ["att", node]. One per node, overwritten by each committee-signed renewal. It is the record the key broker and callers read before trusting a node.

Field	Type	Bytes	Meaning
discriminator, version, bump		10	Header
node	Pubkey	32	Node PDA
sequence	u64	8	Replay barrier; advanced by each record and by revocation
signed_sequence	u64	8	Sequence of the latest committee-signed record
operator_epoch	u64	8	Operator revocation epoch the committee signed
runtime_measurement	[u8; 48]	48	TDX runtime measurement
gpu_rim_hash	[u8; 32]	32	Hash of the NVIDIA reference measurements matched
evidence_hash	[u8; 32]	32	Hash of the archived attestation evidence
enclave_key_hash	[u8; 32]	32	Hash of the enclave's HPKE, TLS and voucher public keys
tcb_status	TcbStatus	1	UpToDate for every record the program accepts
slot	u64	8	Recent slot bound into the evidence
recent_slot_hash	[u8; 32]	32	Hash of that slot
expires_at	i64	8	Expiry, at most 24 hours after recording
recorded_at	i64	8	Recording time
method	AttestationMethod	1	Committee
verifier_epoch	u64	8	Epoch of the signing committee
signer_bitmap	u8	1	Bit <i>i</i> set when member <i>i</i> signed
canonical_message_hash	[u8; 32]	32	SHA-256 of the 200-byte signed message
revoked_at	Option<i64>	9	Revocation time; a set value latches the record as revoked
revoked_by	Option<Pubkey>	33	Revoking authority
revoke_reason_hash	Option<[u8; 32]>	33	Hash of the published reason
Total		384	

7.2.8 Runtime

Seeds ["rt", SHA-256(measurement)]. The 48-byte measurement is hashed only to form the seed; the full measurement is stored. A Runtime is the protocol-level allowlist entry for one measured image; each model additionally pins its own list of measurements.

Field	Type	Bytes	Meaning
discriminator, version, bump		10	Header
measurement	[u8; 48]	48	Runtime measurement
image_name	String, max 64	68	Image name, 1 to 64 bytes
image_version	String, max 32	36	Image version, up to 32 bytes
source_commit	String, max 64	68	Source commit, exactly 40 lowercase hexadecimal characters
proposed_at	i64	8	Time allow_runtime landed
effective_at	i64	8	proposed_at + 172,800 seconds
revoked	bool	1	Revocation flag
revoked_at	Option<i64>	9	Revocation time
Total		256	

7.2.9 Model

Seeds ["model", owner, id.to_le_bytes()]. A model listing: the commitment to its encrypted weights, the key broker that releases its data key, its eligibility policy, its price and its revenue split.

Field	Type	Bytes	Meaning
discriminator, version, bump		10	Header
owner	Pubkey	32	Model owner
id	u64	8	Owner-local model identifier
manifest_hash	[u8; 32]	32	Hash of the encrypted-weights manifest
key_broker	Pubkey	32	Key broker public key
key_broker_url	String, max 128	132	HTTPS URL of the key broker
allowed_runtimes	Vec<[u8; 48]>, max 8	388	Runtime measurements this model accepts
min_tier	u8	1	Minimum operator tier, 1 to 3
min_bond	u64	8	Minimum operator bond in base units
operator_allowlist	Vec<Pubkey>, max 16	516	Operators this model accepts
price	Price	25	Exact or Upto price in base units
derivative_royalty_bps	u16	2	Royalty that derivatives of this model must pay to it

Field	Type	Bytes	Meaning
base_model	Option<Pubkey>	33	Model this one derives from
split_revision	u64	8	Revision of the current split, 1 at creation
split	Vec<SplitEntry>, max 8	548	Current split
pending_split	Option<PendingSplit>	573	Proposed split and its effective time
Total		2,348	

7.2.10 Lane

Seeds ["lane", model, node]. The settlement unit for one model served on one node. Callers pay into its vault, the lane's own ATA, and distribute allocates the vault balance.

Field	Type	Bytes	Meaning
discriminator, version, bump		10	Header
model	Pubkey	32	Model PDA
node	Pubkey	32	Node PDA
vault_ata	Pubkey	32	ATA of this lane for USDC
total_in	u64	8	Revenue counted into the lane; equal to total_allocated after each distribution
total_distributed	u64	8	USDC actually transferred out, including later claims
last_distributed	Option<i64>	9	Time of the last allocating distribution
total_allocated	u64	8	Revenue assigned to fee and recipients
reserved_amount	u64	8	Deferred entitlements still held in the vault
active_split_revision	u64	8	Model split revision this lane settles under
active_split	Vec<SplitEntry>, max 8	548	Snapshot of that split
eligible_protocol_fees_paid	u64	8	Protocol fees actually delivered into the buyback vault, directly or through claim_protocol_fee; the basis of dispute admission
admission_credit_consumed	u64	8	Part of that amount already used to admit disputes
Total		719	

Two identities reconcile every lane, where payments_in is all USDC that has entered the vault:

```

payments_in = lane.total_distributed + vault.amount
lane.total_allocated = lane.total_distributed + lane.reserved_amount

```

7.2.11 Claim

Seeds `["claim", lane, recipient_ata]`. A deferred entitlement for one recipient on one lane. Claims are prepared for recipients whose positive share cannot be paid directly; healthy payouts need none. The protocol fee's Claim has the buyback vault as `recipient_ata` and the protocol PDA as `beneficiary`.

Field	Type	Bytes	Meaning
discriminator, version, bump		10	Header
lane	Pubkey	32	Lane PDA
recipient_ata	Pubkey	32	Recipient token account the share was addressed to
beneficiary	Pubkey	32	Wallet entitled to claim; fixed when the Claim is created
amount	u64	8	Accrued, unclaimed amount
eligible_protocol_fee_amount	u64	8	Part of <code>amount</code> that is an eligible protocol fee
Total		122	

7.2.12 Dispute and ClaimantDisputes

Seeds `["dispute", operator, sequence.to_le_bytes()]`. One complaint, or one arbitration action, against an operator. The Dispute PDA is the token authority of its deposit vault.

Field	Type	Bytes	Meaning
discriminator, version, bump		10	Header
operator	Pubkey	32	Operator PDA
sequence	u64	8	Operator-local sequence number
model	Pubkey	32	Claimant's model, or the default key for a security-only action
claimant	Pubkey	32	Claimant wallet
evidence_hash	[u8; 32]	32	Hash of the complaint evidence
deposit	u64	8	Deposit taken at admission; 0 for arbitration actions
deposit_vault	Pubkey	32	ATA of this PDA for USDC
status	DisputeStatus	1	<code>Open</code> , <code>RuledForClaimant</code> or <code>Dismissed</code>
ruling_hash	Option<[u8; 32]>	33	Hash of the published ruling

Field	Type	Bytes	Meaning
slash_amount	u64	8	Bond slashed by the ruling
opened_at	i64	8	Admission time
ruled_at	Option<i64>	9	Ruling time
claimable_amount	u64	8	Amount the claimant can withdraw
slash_pending_amount	u64	8	Slash reserved in the bond vault, not yet moved
lane	Pubkey	32	Lane the admission credit came from
node	Pubkey	32	Node of that lane
forfeiture_ata	Pubkey	32	Forfeiture destination snapshotted at admission
forfeiture_beneficiary	Pubkey	32	Forfeiture beneficiary snapshotted at admission
deposit_policy	DisputeDepositPolicy	1	ForfeitOnDismissal
forfeited_amount	u64	8	Deposit forfeited on dismissal, not yet claimed
arbitration_initiated	bool	1	True for records created by <code>arbitrate_operator</code>
admission_credit_used	u64	8	Lane credit consumed at admission
admission_required_amount	u64	8	Paid-fee threshold that applied
Total		415	

ClaimantDisputes at `["claimant_disputes", claimant, operator]` counts the open disputes one claimant has against one operator, across all of that claimant's models.

Field	Type	Bytes	Meaning
discriminator, version, bump		10	Header
claimant	Pubkey	32	Claimant wallet
operator	Pubkey	32	Operator PDA
open_count	u32	4	Currently open disputes
Total		78	

7.3 Instructions

7.3.1 Conventions

Every instruction except `initialize` carries two trailing accounts for self-CPI events: `event_authority`, the PDA at seed `__event_authority`, and `program`, the `sealroom` program itself. Three checks recur. The *admin check* compares the signer with `Config.admin` and raises

Unauthorized. The *running check* raises `Paused` while `Config.paused` is set. The *healthy-operator check* raises `OperatorFrozen` for a frozen operator and `InsufficientBond` when `bond_amount` is below the minimum for the operator's tier (`ConfigurationUnset` if the tier has none).

Anchor validates account constraints before a handler runs. An account typed as a `Sealroom` account must be owned by the program and carry the right discriminator. A failed `seeds`, `has_one`, `address`, `token::mint`, `token::authority` or `associated_token` constraint aborts with Anchor's constraint error (`ConstraintSeeds`, `ConstraintHasOne`, `ConstraintAddress`, `ConstraintTokenMint`, `ConstraintTokenOwner`, `ConstraintAssociated`), and creating an account with `init` at an address already in use fails in the System Program. The program's own error codes, listed in 7.5, are raised by handler checks and by the constraints that name them explicitly. Every instruction is atomic: any failure, including a failed token CPI, reverts the whole transaction.

7.3.2 Admin and governance

`initialize(genesis_hash, protocol_fee_bps, arbitration, tier_bond_minimums, verifiers)`

- Signer: `admin`, which pays for `Config` and the epoch-0 `VerifierSet`.
- Accounts: `program` must be `sealroom` with `programdata_address` equal to `program_data`; `program_data.upgrade_authority_address` must equal `admin` (`Unauthorized`). `config` is created at `["config"]` and `verifier_set` at `["verifiers", 0]`. `fee_treasury` is a token account of `usdc_mint`.
- Checks: `fee` $\leq 10,000$ bps (`InvalidBps`); exactly five distinct, non-default verifier keys (`InvalidVerifierSet`); exactly three tier minima (`ConfigurationUnset`) listed for tiers 1, 2, 3 in order (`InvalidTier`); mint decimals 6 (`InvalidMint`); `arbitration` different from the `admin` (`AuthoritiesMustDiffer`); the treasury is the canonical ATA of its owner for the mint (`InvalidRecipients`).
- Effects: writes `Config` with the `admin`, mint, genesis hash, fee, the treasury binding and its owner, the four duration constants, `active_verifier_epoch = 0`, `claim_destination_policy = CanonicalAta`, `max_open_per_operator = 8`, `settlement_mechanism = LaneBoundary`, every undecided policy as `None`, no buyback mint or authority, and both buyback totals at zero; writes the `VerifierSet` with threshold 3.
- Event: `ConfigInitialized` (program log).

`set_paused(paused)` — Signer: `admin` (admin check). Sets `Config.paused`. No event.

`set_settlement_policy(claim_policy, rescue_policy, split_settlement_policy, claim_destination_policy)` — Signer: `admin`. Sets `claim_policy` and `claim_destination_policy`. The two retained arguments are accepted and ignored; their fields stay `None`, so the admin cannot alter the lane-boundary settlement mechanism. No event.

`propose_fee(protocol_fee_bps)`

- Signer: `admin`. Accounts: `treasury`, a token account of the `Config` mint.
- Checks: `treasury` is the canonical ATA of its owner (`InvalidRecipients`); `fee` $\leq 10,000$ bps (`InvalidBps`); no fee proposal pending (`PendingSplitExists`).

- Effects: `pending_fee = { protocol_fee_bps, treasury_ata, treasury_owner, effective_at = now + 172,800 }`. The treasury binding is retained in the interface and directs no funds: `distribute` pays every fee into the buyback vault, whose address no proposal can change.
- Event: `FeeProposed`.

`cancel_fee()` — Signer: admin. Clears `pending_fee` (`NoPendingSplit` if none). No event.

`apply_fee()` — No privileged signer. Requires a pending fee (`NoPendingSplit`) whose `effective_at` has passed (`TimelockPending`). Writes `protocol_fee_bps`, `fee_treasury_ata` and `fee_treasury_owner` and clears the proposal. Every later `distribute` applies the new rate. Event: `FeeApplied`.

`propose_dispute_config(arbitration, dispute_deposit, max_open_per_claimant_operator, max_open_per_operator, deposit_policy)`

- Signer: admin. Accounts: `forfeiture_treasury`, a token account of the Config mint.
- Checks: `arbitration` is neither the admin nor the default key (`AuthoritiesMustDiffer`); `deposit` and both caps are positive and the policy is `ForfeitOnDismissal` (`ConfigurationUnset`); no dispute proposal pending (`PendingSplitExists`); the forfeiture treasury is canonical (`InvalidRecipients`) and its owner is not the proposed arbitration key (`AuthoritiesMustDiffer`).
- Effects: `pending_dispute_config` with the snapshotted forfeiture owner and `effective_at = now + 172,800`.
- Event: `DisputeConfigProposed`.

`cancel_dispute_config()` — Signer: admin. Clears the proposal (`NoPendingSplit` if none). No event.

`apply_dispute_config()` — No privileged signer. Requires a pending proposal (`NoPendingSplit`) past its effective time (`TimelockPending`), and re-checks that the proposed arbitration differs from the current admin and from the forfeiture owner (`AuthoritiesMustDiffer`). Writes `arbitration`, `dispute_deposit`, both caps, `dispute_forfeiture_ata`, `dispute_forfeiture_owner` and `dispute_deposit_policy`, and clears the proposal. Event: `DisputeConfigApplied`.

`set_dispute_config(arbitration, dispute_deposit, deposit_policy)` — Retained in the interface. It always fails with `GovernanceTimelockRequired` and changes no state; dispute settings change only through the proposal path above.

7.3.3 Operators

`register_operator(agreement_hash, tier)`

- Signer: admin, who pays rent. The operator's `authority` is named but does not sign; the admin admits it after off-chain vetting.
- Accounts: `operator` created at `["op", authority]`; `bond_vault` created if needed as the ATA of the Operator PDA for USDC; `payout_ata` must already exist as the canonical USDC ATA of `authority`; `usdc_mint` must equal `Config.usdc_mint`.
- Checks: admin check, running check, the tier has a configured minimum (`ConfigurationUnset`).

- Effects: a new Operator with zero bond, no unbond request, not frozen, zero counters and `attestation_epoch = 0`.
- Event: `OperatorRegistered`.

`set_tier(tier)` — Signer: admin. The tier must have a minimum (`ConfigurationUnset`). Sets `Operator.tier`. Event: `OperatorTierSet`.

`freeze_operator(frozen)` — Signer: admin. Sets or clears `Operator.frozen`. A frozen operator cannot deposit, unbond, withdraw, register platforms or nodes, or record attestations. Event: `OperatorFrozen`.

`deposit_bond(amount)`

- Signer: operator authority. Accounts: `operator` at `["op", authority]` with `has_one = authority` and `has_one = bond_vault`; `source`, a USDC account whose token authority is the signer; `bond_vault`, a USDC account whose token authority is the Operator PDA.
- Checks: running; `amount > 0` (`InvalidAmount`); not frozen (`OperatorFrozen`); no unbond request (`Unbonding`).
- Effects: `transfer_checked` of `amount` into the vault; `bond_amount += amount`. Tokens sent to the vault by any other route do not raise `bond_amount`.
- Event: `BondDeposited`.

`request_unbond()` — Signer: operator authority (`has_one = authority`). Requires a positive bond (`InvalidAmount`), no freeze (`OperatorFrozen`), no open dispute (`OpenDispute`) and no existing request (`Unbonding`). Sets `unbond_at = now + 2,592,000`. Event: `UnbondRequested`.

`cancel_unbond()` — Signer: operator authority. Requires a request (`UnbondNotRequested`) and clears it; no funds move. Event: `UnbondCancelled`.

`withdraw_bond()`

- Signer: operator authority. Accounts: Operator with `has_one = authority`, `bond_vault`; `destination`, a USDC account owned by the authority.
- Checks: not frozen (`OperatorFrozen`); no open dispute (`OpenDispute`); a request exists (`UnbondNotRequested`) and `now ≥ unbond_at` (`TimelockPending`); the vault holds at least `bond_amount` (`InvalidAmount`).
- Effects: transfers exactly `bond_amount`, signed by the Operator PDA, and resets `bond_amount` and `unbond_at`. Reserved slashes are already outside `bond_amount` and stay in the vault for `settle_slash`. A matured request with zero principal, as after a full slash, clears `unbond_at` without a transfer.
- Event: `BondWithdrawn` when a transfer occurs.

`register_platform(platform_id_hash, gpu_ueid_hashes)`

- Signers: operator authority (rent payer) and admin. Accounts: Operator with `has_one = authority`; `platform` created at `["plat", operator, platform_id_hash]`; `platform_pin`

created at `["platform_pin", platform_id_hash]`; one remaining account per GPU, each the GpuPin PDA for its UEID hash.

- Checks: admin check; running; healthy operator; 1 to 8 distinct GPU hashes with one remaining account each (`InvalidGpuSet`); each GpuPin address is the canonical PDA, writable, empty and System-owned (`GpuAlreadyClaimed`).
- Effects: creates every GpuPin with `node = None`, the PlatformPin and the Platform with `registered = true`. The program tops up a GpuPin address only by the lamports it lacks for rent, so lamports sent to that address in advance cannot block its creation.
- Event: `PlatformRegistered`.

```
register_node(index, node_key, endpoint, gpu_model, gpu_ueid_hashes)
```

- Signer: operator authority (rent payer). Accounts: Operator with `has_one = authority`; Platform at its seeds with `has_one = operator`; `node` created at `["node", platform, index]`; one GpuPin per GPU as remaining accounts.
- Checks: running; healthy operator; platform registered (`PlatformNotRegistered`); `endpoint` is a valid HTTPS URL of at most 128 bytes (`InvalidEndpoint`); 1 to 8 distinct GPU hashes, all on the platform, with one account each (`InvalidGpuSet`); each GpuPin is the program-owned canonical PDA for its hash (`InvalidGpuSet`), belongs to this platform and has no node (`GpuAlreadyClaimed`).
- Effects: sets each GpuPin's `node`; writes the Node with `gpu_count` equal to the number of hashes and `status = Registered`.
- Event: `NodeRegistered`.

The endpoint validator accepts only ASCII `https://` URLs with no user information, ports from 1 to 65535, DNS labels of at most 63 characters without punycode prefixes, a host of at most 253 characters, dotted-quad IPv4 without leading zeros, or bracketed IPv6. The same validator governs `Model.key_broker_url`.

7.3.4 Attestation

```
record_attestation(commitment, epoch, sequence, recent_slot, recent_slot_hash)
```

- Signer: any relayer, which pays for the Attestation account on first use. In service the coordinator submits it after collecting committee signatures.
- Accounts: `verifier_set` at `["verifiers", epoch]`; `operator` at its seeds; `platform` at its seeds with `has_one = operator`; `node` at its seeds with `has_one = platform, operator`; `runtime` at `["rt", SHA-256(runtime.measurement)]`; `attestation` created if needed at `["att", node]`; `instructions` and `slot_hashes` at the exact sysvar addresses.

The handler checks, in order:

1. Running (`Paused`) and healthy operator (`OperatorFrozen`, `InsufficientBond`).
2. Platform registered (`PlatformNotRegistered`) and node status `Registered` (`NodeDisabled`).
3. Every GPU hash on the node is on the platform (`InvalidGpuSet`).
4. Runtime not revoked and `now ≥ runtime.effective_at` (`RuntimeNotAllowed`).

5. The commitment's measurement equals the Runtime's and its platform equals the Platform PDA (`AttestationMismatch`).
6. `tcb_status = UpToDate` and `method = Committee` (`InvalidTcb`).
7. `now < expires_at ≤ now + 86,400` (`AttestationExpired`) and, during unbonding, `expires_at ≤ unbond_at` (`AttestationBeyondUnbond`).
8. `epoch` equals `Config.active_verifier_epoch` and the set's epoch, and the set is not retired (`InvalidVerifierSet`).
9. The commitment's `operator_epoch` equals `Operator.attestation_epoch` (`AttestationReplay`).
0. The record is not latched revoked (`AttestationRevoked`), and `sequence` is 1 for a new account or the stored sequence plus 1 (`AttestationReplay`).
1. The pair (`recent_slot`, `recent_slot_hash`) appears in the `SlotHashes` sysvar, which holds up to 512 recent entries (`InvalidRecentSlot`).
2. At least three distinct members of the set signed the canonical message (below).

It then writes every commitment field, sets `sequence` and `signed_sequence` to the new sequence, stores the signer bitmap and `canonical_message_hash = SHA-256(message)` , clears the revocation fields and emits `AttestationRecorded` .

The committee signs a 200-byte message. Its 32-byte commitment field is the SHA-256 of the 194-byte Borsh encoding of the `AttestationCommitment` :

Canonical message (200 bytes)

offset	size	field
0	16	"sealroom.att.v0\0"
16	32	program id
48	32	Config.genesis_hash
80	32	Node PDA
112	8	verifier epoch (u64 LE)
120	8	sequence (u64 LE)
128	32	SHA-256(AttestationCommitment)
160	8	recent slot (u64 LE)
168	32	recent slot hash

AttestationCommitment (194 bytes, Borsh)

offset	size	field
0	32	platform (Platform PDA)
32	48	runtime_measurement
80	32	gpu_rim_hash
112	32	evidence_hash
144	32	enclave_key_hash
176	1	tcb_status
177	8	expires_at (i64 LE)
185	1	method
186	8	operator_epoch (u64 LE)

The message binds the program, the cluster, the node, the committee epoch, the replay sequence, the evidence and a recent slot, so a signature cannot be replayed against another deployment, another node or another revision.

Signatures arrive in native Ed25519 precompile instructions placed earlier in the same transaction; the runtime verifies them before the program executes. The program reads the Instructions sysvar, visits every Ed25519 instruction before the current one and parses each. The instruction must carry no accounts, a signature count from 1 to 7 and a zero padding byte; each 14-byte offsets entry must reference data inside that same instruction (all three instruction indices equal to `0xFFFF`), and each referenced signature, key and message must lie within bounds (`InvalidSignatureInstruction`). The message must be exactly the 200 canonical bytes (`WrongAttestationMessage`), the key must be a member of the set (`UnknownVerifier`), and no member may count twice, including across separate Ed25519 instructions (`DuplicateVerifier`). Fewer than three distinct members raises `InsufficientSignatures`. The set itself must still be five distinct keys at threshold three (`InvalidVerifierSet`).

```
revoke_attestation(reason_hash)
```

- Signer: the admin, or a member of the VerifierSet whose epoch is the active epoch (`Unauthorized`).
- Effects: if the record is not already revoked, `sequence += 1`, `revoked_at = now`, `revoked_by = signer`, `revoke_reason_hash = reason_hash`, and `AttestationRevoked` is emitted. A second revocation is a no-op. `signed_sequence` is unchanged.

```
clear_attestation_revocation()
```

- Signer: admin or arbitration (`Unauthorized`). The record must be revoked (`AttestationNotRevoked`).
- Effects: clears the three revocation fields and sets `expires_at = now`, so the record is expired at once. The sequence keeps the value revocation consumed, and a node returns to service only through a new committee record at the next sequence.
- Event: `AttestationRevocationCleared`.

A worked sequence: a record at sequence 10 has `signed_sequence = 10`. Revocation moves `sequence` to 11. A renewal signed for 11 before the revocation fails with `AttestationRevoked` while the latch is set, and with `AttestationReplay` after a clear, because the next acceptable sequence is then 12. Arbitration invalidates every attestation of an operator at once by advancing `Operator.attestation_epoch`: existing records carry the old epoch, the key broker and callers treat an epoch mismatch as revocation, and new records must commit to the new epoch.

7.3.5 Runtimes

```
allow_runtime(measurement, image_name, image_version, source_commit)
```

- Signer: admin, who pays rent. Accounts: `runtime` created if needed at `["rt", SHA-256(measurement)]`.
- Checks: admin check; image name 1 to 64 bytes, version at most 32 bytes and a source commit of exactly 40 lowercase hexadecimal characters (`BoundsExceeded`); the Runtime is new or revoked (`RuntimeAlreadyAllowed`).
- Effects: writes the metadata, `proposed_at = now`, `effective_at = now + 172,800`, `revoked = false`. Re-allowing a revoked runtime starts a full new delay.

- Event: `RuntimeProposed`.

`revoke_runtime()` — Signer: `admin`. Sets `revoked = true` and `revoked_at = now` with immediate effect: no new attestation for that measurement can be recorded. Event: `RuntimeRevoked`.

7.3.6 Models and splits

`create_model(id, manifest_hash, key_broker, key_broker_url, allowed_runtimes, min_tier, min_bond, operator_allowlist, price, derivative_royalty_bps, base_model, split)`

- Signer: owner, who pays rent. Accounts: `model` created at `["model", owner, id]`; `base_model` optional; one remaining account for each split recipient ATA, including a derivative's royalty recipient.
- Running check (`Paused`).
- Split checks: 1 to 8 entries, each with positive weight and a distinct, non-default recipient ATA, summing to 10,000 bps (`InvalidSplit`); exactly one entry with role `Operator` (`InvalidOperatorPayout`); that entry is not locked (`OperatorLockUnsupported`); every recipient's account is present, owned by SPL Token, of the Config mint, owned by `recipient_owner`, and is that owner's canonical ATA, with a non-default owner (`InvalidRecipients`).
- Policy checks: 1 to 8 distinct allowed runtimes and at most 16 distinct, non-default allowlisted operators (`BoundsExceeded`); `min_tier` from 1 to 3 (`InvalidTier`); royalty $\leq 10,000$ bps (`InvalidBps`); non-default key broker key and a valid HTTPS broker URL of at most 128 bytes (`InvalidEndpoint`); positive `per_call` or `max_per_call` (`InvalidAmount`).
- Base model: when `base_model` is named, the matching Model account must be supplied (`InvalidBaseModel`); if its `derivative_royalty_bps` is positive, the split must contain a locked `BaseModel` entry with at least that weight, paid to the base owner's canonical ATA (`RoyaltyRequired`). With no base model, no base account may be supplied (`InvalidBaseModel`).
- Effects: writes the Model with `split_revision = 1` and no pending split.
- Event: `ModelCreated`.

The Operator-role entry declares a weight only. At distribution its destination is always the serving operator's `payout_ata` with the operator authority as beneficiary, whatever address the model stored, so one model served by several operators pays each operator on its own lanes.

`update_price(price)`

- Signer: owner, with `has_one = owner` on the Model at `["model", owner, id]`.
- Checks: running (`Paused`); positive `per_call` for `Exact` or positive `max_per_call` for `Upto` (`InvalidAmount`).
- Effects: replaces `Model.price` in the transaction that carries it. The split, its revision, every lane and every vault are unchanged.
- Event: `ModelPriceUpdated { model, price, timestamp }`.

`update_policy(policy)`, with `policy = { allowed_runtimes, min_tier, min_bond, operator_allowlist }`

- Signer: owner, as for `update_price`.

- Checks: running (`Paused`); 1 to 8 distinct allowed runtimes and at most 16 distinct, non-default allowlisted operators (`BoundsExceeded`); `min_tier` from 1 to 3 (`InvalidTier`).
- Effects: replaces the four policy fields in the transaction that carries it. Verifiers, key brokers and nodes read these fields directly from the chain, so key release and serving apply the new policy from their next read.
- Event: `ModelPolicyUpdated` with the model, the four new values and the timestamp.

Neither instruction has a timelock, and neither moves funds. The key broker, the manifest hash, the derivative royalty and the split are not part of either; the split changes only through the seven-day path below.

`propose_split(entries)`

- Signer: owner, with `has_one = owner` on the Model; locked-entry beneficiaries co-sign as remaining accounts when needed.
- Checks: running; the split and recipient-account checks of `create_model`; no pending split (`PendingSplitExists`). Every locked entry of the current split must either survive unchanged in recipient, owner and role, still locked and with weight not lower, or its `recipient_owner` must sign the transaction (`LockedSplit`).
- Effects: `pending_split = { revision: split_revision + 1, proposed_at: now, effective_at: now + 604,800, entries }`.
- Event: `SplitProposed`.

`cancel_split()` — Signer: owner. Removes the pending split (`NoPendingSplit`). Available while paused. Event: `SplitCancelled`.

`apply_split()` — No privileged signer. Running check; a pending split exists (`NoPendingSplit`) and its `effective_at` has passed (`TimelockPending`). Sets `split_revision` and `split` from the proposal. Event: `SplitApplied`. Lanes adopt the new revision one at a time at their next settlement boundary (7.3.7), so revenue already in a vault settles under the split it accrued under.

7.3.7 Lanes and distribution

`open_lane()`

- Signer: any payer. Accounts: Model and Node at their seeds, Node with `has_one = operator`, Operator at its seeds; `lane` created at `["lane", model, node]`; `vault` created if needed as the canonical ATA of the Lane PDA for USDC, which also accepts a vault created in advance by another party.
- Checks: running.
- Effects: a Lane with zero counters and a snapshot of the model's current split and revision. Opening a lane grants no key eligibility; the key broker applies the model's policy separately.
- Event: `LaneOpened`.

`distribute()`

- No privileged signer. Accounts: Config, Model, Node (`has_one = operator`), Operator; Lane with `has_one = model`, node and `vault_ata == vault`; `vault`, the Lane's canonical ATA; `fee_treasury`, whose address must equal the buyback vault, the canonical USDC ATA of the protocol PDA, and differ from the vault (the account keeps its interface name); `fee_claim`, the Claim PDA `["claim", lane, buyback vault]`; then one pair of remaining accounts (recipient ATA, Claim PDA) per split entry, in split order.

The handler proceeds as follows:

1. Running (`Paused`) and claim policy set (`ConfigurationUnset`).
2. Resolve the lane's active split, replacing the Operator-role entry's destination with `Operator.payout_ata` and `Operator.authority`.
3. Compute `available = vault.amount - reserved_amount` (`InvalidAmount` if negative).
4. If `available = 0`, the call is a settlement boundary: when the model's revision differs from the lane's and the vault holds only reserved funds, copy the model split into the lane and emit `LaneSplitActivated`; then return.
5. Require two remaining accounts per entry (`InvalidRecipients`), authenticate the fee Claim with the protocol PDA as its beneficiary (`InvalidClaim`), and read the fee rate (`ConfigurationUnset`).
6. Compute the allocation (formula below).
7. Before any transfer, validate every pair: the ATA equals the resolved recipient, is writable and is not the vault (`InvalidRecipients`); the Claim slot is either an authenticated Claim for this lane, recipient and beneficiary or an empty System-owned placeholder at the right PDA (`InvalidClaim`); a positive share whose account cannot receive it requires an initialized Claim (`InvalidClaim`). The fee is held to the same rule.
8. Pay the fee and each share. An account is payable when it is an SPL Token account of the Config mint, owned by the beneficiary and in the `Initialized` state; for the fee, the beneficiary is the protocol PDA. A payable recipient receives a `transfer_checked` signed by the Lane PDA. An unpayable share (frozen, closed or owner-changed account) is added to its Claim and to `reserved_amount`, and the other payouts proceed. A fee the buyback vault cannot receive is added to the fee Claim, with its eligible portion in `eligible_protocol_fee_amount`.
9. Assert that paid plus deferred equals `available` (`Overflow`).
0. Update the lane: `total_allocated += available`, `total_distributed += paid`, `total_in = total_allocated`, `reserved_amount += deferred`, `last_distributed = now`, and add the fee to `eligible_protocol_fees_paid` when it was paid directly into the buyback vault, whose owner, the protocol PDA, is neither the model owner nor the operator authority.
1. Emit `LaneDistributed` with the fee paid, allocated and deferred, the paid and deferred totals, the fee rate, the buyback vault in its `fee_treasury_ata` field, the hash of the active split, its revision and one payout record per entry.
2. Reload the vault and run the boundary step of 4. Because the lane and its vault are write-locked by the instruction, a payment lands wholly before or wholly after the boundary.

```

available = vault.amount - lane.reserved_amount
fee = floor(available * protocol_fee_bps / 10000)
net = available - fee
share_i = floor(net * bps_i / 10000)           for every entry
except the last
share_last = net - (sum of all other shares)
fee + sum(share_i) = available

```

A worked example at the launch fee of 500 bps, with an illustrative split of 7,000 bps to the owner, 2,000 to the operator and 1,000 to a contributor listed last: an available balance of 1,234,567 base units gives a fee of $\text{floor}(61,728.35) = 61,728$ and a net of 1,172,839. The owner receives $\text{floor}(820,987.3) = 820,987$ and the operator $\text{floor}(234,567.8) = 234,567$. The contributor's floor is 117,283, and as the final entry it also receives the 2 base units of rounding remainder, for 117,285. Fee and shares sum to exactly 1,234,567. With no fee and eight equal entries of 1,250 bps, 11 base units divide as 1, 1, 1, 1, 1, 1, 1 and 4.

`rescue_unopened_lane()` — Retained in the interface. It always fails with `RescueDisabled`, whoever signs, and never moves or closes a vault. Funds sent to a lane vault before its lane exists are settled by `open_lane` and `distribute` like any other revenue, under the model's split and fee.

7.3.8 Claims

`prepare_claim(recipient_ata, recipient_owner)`

- Signer: any payer. Accounts: Model, Lane (`has_one = model, node`), Node (`has_one = operator`), Operator; `claim` created if needed at `["claim", lane, recipient_ata]`.
- Checks: claim policy set (`ConfigurationUnset`); the pair is an entry of the lane's resolved split or the buyback vault with the protocol PDA as owner (`InvalidClaim`); an existing Claim must match lane, recipient and beneficiary (`InvalidClaim`).
- Effects: a zero-amount Claim bound to the beneficiary; an existing matching Claim is left unchanged. The keeper prepares Claims only for recipients whose positive share cannot be paid directly. No event.

`claim_deferred()`

- Signer: the Claim's beneficiary. Accounts: Lane; Claim at `["claim", lane, claim.recipient_ata]` with `has_one = lane, beneficiary`; vault at `lane.vault_ata` with token authority the Lane PDA; `destination`, a USDC account owned by the beneficiary and different from the vault.
- Checks: the destination is owned by the beneficiary and of the mint, and under `CanonicalAta` it is the beneficiary's canonical ATA (`InvalidClaimDestination`); claim policy set (`ConfigurationUnset`); `reserved_amount ≥ amount` (`InvalidClaim`). A zero amount returns without effect.
- Effects: transfers the amount, signed by the Lane PDA; adds the Claim's eligible fee portion to `eligible_protocol_fees_paid`; zeroes the Claim; reduces `reserved_amount`; increases

`total_distributed`. Claims are not gated by the pause flag.

- Event: `DeferredClaimPaid`.

The protocol fee's Claim names the protocol PDA as beneficiary, and no key signs for that PDA, so `claim_deferred` never pays it. `claim_protocol_fee` does.

`claim_protocol_fee()`

- Signer: any payer. Accounts: Config; Lane at its seeds; Claim at `["claim", lane, claim.recipient_ata]` with `has_one = lane`, the protocol PDA as beneficiary and `recipient_ata` equal to `destination`; protocol at `["protocol"]`; vault at `lane.vault_ata` with token authority the Lane PDA; `destination`, the buyback vault, with token authority the protocol PDA.
- Checks: claim policy set (`ConfigurationUnset`); `reserved_amount ≥ amount` (`InvalidClaim`). A zero amount returns without effect.
- Effects: transfers the amount into the buyback vault, signed by the Lane PDA; adds the Claim's `eligible_protocol_fee_amount` to `eligible_protocol_fees_paid`; zeroes the Claim; reduces `reserved_amount`; increases `total_distributed`. It is not gated by the pause flag, and its only destination is the buyback vault.
- Event: `DeferredClaimPaid`, with the protocol PDA as beneficiary.

7.3.9 Disputes and arbitration

`open_dispute(sequence, evidence_hash)`

- Signer: the claimant, who must own the model (`model` derives from the claimant's key and `model.owner == claimant`, else `Unauthorized`) and pays rent.
- Accounts: Operator at its seeds; Node with `has_one = operator`; Lane with `has_one = model`, node; `claimant_disputes` created if needed at `["claimant_disputes", claimant, operator]`; `dispute` created at `["dispute", operator, sequence]`; source, a USDC account of the claimant; `deposit_vault` created if needed as the Dispute PDA's ATA.
- Checks: deposit, arbitration and deposit policy configured and the deposit positive (`ConfigurationUnset`); `sequence == Operator.next_dispute_sequence` (`AttestationReplay`); policy `ForfeitOnDismissal` (`ConfigurationUnset`); the lane's unused credit covers the deposit (`InsufficientPaidUsage`); the per-claimant cap is configured (`ConfigurationUnset`); the operator is below `max_open_per_operator` and the claimant below its cap (`DisputeLimitReached`).

```
credit = lane.eligible_protocol_fees_paid -  
lane.admission_credit_consumed  
admitted only if credit >= config.dispute_deposit  
lane.admission_credit_consumed += config.dispute_deposit
```

- Effects: consumes the credit, increments the claimant counter, transfers the deposit into the deposit vault, increments `next_dispute_sequence` and `open_dispute_count`, and writes an `Open`

Dispute that snapshots the deposit, the lane and node, the forfeiture destination and beneficiary, and `admission_credit_used = admission_required_amount = deposit`. Consumed credit never returns, whatever the ruling. At the launch fee of 500 bps, the eligible fees behind one deposit correspond to lane revenue of 20 times the deposit whose fee has reached the buyback vault.

- Events: `DisputeAdmission` (credit used and remaining), then `DisputeOpened`.

```
rule_dispute(for_claimant, slash_amount, freeze, revoke_attestations, ruling_hash)
```

- Signer: `Config.arbitration` (`Unauthorized`). A proposed but unapplied arbitration key has no authority.
- Accounts: Operator with `has_one = bond_vault`; Dispute with `has_one = operator, deposit_vault`; ClaimantDisputes matching the Dispute's claimant and operator; the bond vault and deposit vault.
- Checks: the dispute is `Open` (`DisputeAlreadyRuled`); `slash_amount ≤ bond_amount`, and a dismissal carries no slash, freeze or revocation (`InvalidAmount`).
- Effects: `bond_amount -= slash_amount`, `pending_slash_amount += slash_amount`; `frozen = true` if ordered; `attestation_epoch += 1` if ordered; `open_dispute_count` and the claimant counter each decrease by one. A claimant ruling sets `claimable_amount = deposit`; a dismissal sets `forfeited_amount = deposit`. `slash_pending_amount` records the slash. The ruling performs no token transfer, so a frozen token account cannot block it.
- Event: `DisputeRuled`, which records the ruling authority, the deposit returned or forfeited, the forfeiture destination, the freeze state and the new attestation epoch.

```
arbitrate_operator(sequence, evidence_hash, slash_amount, freeze, revoke_attestations, ruling_hash)
```

- Signer: `Config.arbitration` (`Unauthorized`), which pays rent.
- Accounts: Operator; `claimant`, a named wallet; optional `model`, `node` and `lane`; ClaimantDisputes and Dispute created as in `open_dispute`; the deposit vault created if needed.
- Relationship: either all three of model, node and lane are supplied, or none. When supplied, each must be the canonical PDA for its own fields, the node must belong to this operator, the lane must link that model and node, and the model's owner must be the claimant (`Unauthorized` on any mismatch or a partial set).
- Admission: a zero slash needs nothing more. A positive slash requires a configured, positive deposit (`ConfigurationUnset`) and a lane whose lifetime `eligible_protocol_fees_paid` is at least that deposit (`InsufficientPaidUsage`); no credit is consumed.
- Checks: `sequence == next_dispute_sequence` (`AttestationReplay`); `slash_amount ≤ bond_amount` (`InvalidAmount`).
- Effects: advances the sequence; reserves the slash as in `rule_dispute`; applies freeze and revocation; writes a Dispute already `RuledForClaimant` with zero deposit, `arbitration_initiated = true`, `admission_credit_used = 0` and `admission_required_amount` equal to the threshold applied. It takes no open-dispute slot, so a security action is available even when every slot is occupied.
- Event: `ArbitrationAction`.

settle_slash()

- Signer: any; the account in the `arbitration` position is not compared with `Config.arbitration`, so settlement is permissionless. Accounts as in `rule_dispute`.
- Checks: the dispute is no longer `Open` (`DisputeAlreadyRuled`). A zero pending slash returns without effect.
- Effects: transfers `slash_pending_amount` from the bond vault to the deposit vault, signed by the Operator PDA; `claimable_amount += amount`; `pending_slash_amount -= amount`; `slash_pending_amount = 0`.
- Event: `SlashSettled`.

`claim_dispute()` — Signer: the claimant (`has_one = claimant, deposit_vault`). The destination must satisfy the destination policy (`InvalidClaimDestination`) and the dispute must be ruled (`DisputeAlreadyRuled`). Transfers `claimable_amount` from the deposit vault, signed by the Dispute PDA, and zeroes it. Event: `DisputeClaimPaid`.

`claim_forfeited_deposit()` — Signer: the Dispute's `forfeiture_beneficiary` (`Unauthorized`). The dispute must be `Dismissed` (`DisputeAlreadyRuled`) and the destination must satisfy the policy (`InvalidClaimDestination`). Transfers `forfeited_amount` to the beneficiary and zeroes it. Event: `DisputeDepositForfeited`.

The dispute amounts move through these states:

State	claimable_amount	forfeited_amount	slash_pending_amount
Open	0	0	0
Ruled for claimant by <code>rule_dispute</code>	deposit	0	slash
Dismissed by <code>rule_dispute</code>	0	deposit	0
Recorded by <code>arbitrate_operator</code>	0	0	slash
After <code>settle_slash</code>	previous + slash	unchanged	0

7.3.10 Buyback

The protocol fee accumulates in the buyback vault, the canonical USDC ATA of the protocol PDA `["protocol"]`. The program signs for that PDA in one place only, inside `buyback_swap_and_burn`, and no instruction withdraws from the vault or from the PDA's output account.

`set_buyback_authority(authority)` — Signer: admin (admin check). `authority` must not be the default key (`Unauthorized`). Sets `Config.buyback_authority`; a later call replaces it. Event: `BuybackAuthoritySet`.

`set_buyback_mint()`

- Signer: admin. Accounts: Config; `buyback_mint`, a mint account of SPL Token or Token-2022.
- Checks: admin check; no mint selected yet (`BuybackMintAlreadySet`); the mint is not the USDC mint, has no mint authority, has a positive supply of at most 1,000,000,000 tokens at its decimals

($1,000,000,000 \times 10^{\text{decimals}}$ base units), and is either an SPL Token mint or a Token-2022 mint whose only extensions are `MetadataPointer` and `TokenMetadata` (`UnsupportedBuybackMint`).

- Effects: sets `Config.buyback_mint`. No instruction changes it afterwards.
- Event: `BuybackMintSet`, with the mint and its token program.

`buyback_swap_and_burn(amount, min_out, swap_data)`

- Signer: `Config.buyback_authority` (`Unauthorized`).
- Accounts: `Config` (writable); `protocol` at `["protocol"]`; `usdc_mint` equal to `Config.usdc_mint`; `buyback_vault` at the protocol PDA's canonical USDC ATA with token authority the PDA; `buyback_mint` equal to `Config.buyback_mint` (`ConfigurationUnset`); `output_vault` at the PDA's canonical ATA for that mint under the mint's own token program; `swap_program`, an executable account; `token_program`; `buyback_token_program`, the program that owns the mint; then the Jupiter route accounts as remaining accounts.
- Checks before the swap: running (`Paused`); `amount > 0`, `min_out > 0` and a vault balance of at least `amount` (`InvalidAmount`); the mint still passes the `set_buyback_mint` checks (`UnsupportedBuybackMint`); neither the vault nor the output account has a delegate or a close authority (`InvalidBuybackSettlement`).
- Route checks (`InvalidBuybackRoute`): `swap_program` is Jupiter v6, `JUP6LkbZbjS1jKKwapdHNy74zcZ3tLUZoi5QNYVTaV4`; `swap_data` is a `route` or `shared_accounts_route` instruction whose encoded input amount equals `amount` and whose platform fee is zero; the route accounts carry the SPL Token program, the protocol PDA as transfer authority, the buyback vault as source, the output account as destination and the buyback mint at their fixed positions; no route account other than the protocol PDA is a signer.
- Swap: the program invokes Jupiter with the route bytes unchanged and signs only for the protocol PDA.
- Checks after the swap (`InvalidBuybackSettlement`): both accounts keep their owner and mint and still have no delegate or close authority; the mint supply is unchanged; the vault fell by exactly `amount`; the output account rose by at least `min_out`.
- Burn: `burn_checked` of the entire output increase through the mint's token program, signed by the protocol PDA. The output account must return to its balance before the swap and the supply must fall by exactly the burned amount (`InvalidBuybackSettlement`).
- Effects: `usdc_spent_total += amount`; `burned_total += burned` (`Overflow`).
- Event: `BuybackBurned { usdc_in, burned, slot }`.

A failure at any step reverts the swap and the burn together. The buyback authority starts the instruction and pays its network fee; the USDC moves only from the buyback vault into the route, and every token the route returns is burned before the instruction ends, so neither ever reaches the authority or any other wallet.

7.3.11 Events

Event	Emitted by
<code>ConfigInitialized</code>	<code>initialize</code> (program log)

Event	Emitted by
OperatorRegistered, OperatorTierSet, OperatorFrozen	register_operator, set_tier, freeze_operator
BondDeposited, UnbondRequested, UnbondCancelled, BondWithdrawn	deposit_bond, request_unbond, cancel_unbond, withdraw_bond
PlatformRegistered, NodeRegistered	register_platform, register_node
RuntimeProposed, RuntimeRevoked	allow_runtime, revoke_runtime
AttestationRecorded, AttestationRevoked, AttestationRevocationCleared	record_attestation, revoke_attestation, clear_attestation_revocation
ModelCreated, SplitProposed, SplitCancelled, SplitApplied	create_model, propose_split, cancel_split, apply_split
ModelPriceUpdated, ModelPolicyUpdated	update_price, update_policy
LaneOpened, LaneDistributed, LaneSplitActivated	open_lane, distribute
DeferredClaimPaid	claim_deferred, claim_protocol_fee
BuybackAuthoritySet, BuybackMintSet, BuybackBurned	set_buyback_authority, set_buyback_mint, buyback_swap_and_burn
FeeProposed, FeeApplied	propose_fee, apply_fee
DisputeConfigProposed, DisputeConfigApplied	propose_dispute_config, apply_dispute_config
DisputeAdmission, DisputeOpened	open_dispute
DisputeRuled, ArbitrationAction, SlashSettled	rule_dispute, arbitrate_operator, settle_slash
DisputeClaimPaid, DisputeDepositForfeited	claim_dispute, claim_forfeited_deposit
UnopenedLaneRescued	Retained in the IDL; no instruction emits it

set_paused, set_settlement_policy, cancel_fee, cancel_dispute_config and prepare_claim emit no event; their effects are read from the accounts they change.

7.4 Timelocks and governance actions

Four durations are compiled into the program as constants and copied into Config for readers:

Constant	Seconds	Duration	Applies to
ATTESTATION_TTL	86,400	24 hours	Maximum lifetime of an attestation record
RUNTIME_TIMELOCK	172,800	48 hours	allow_runtime, propose_fee, propose_dispute_config
SPLIT_DELAY	604,800	7 days	propose_split

Constant	Seconds	Duration	Applies to		
UNBONDING_PERIOD	2,592,000	30 days	request_unbond		

A proposal records `effective_at = now + delay` using the cluster Clock's Unix timestamp, and its application succeeds in any later transaction whose Clock timestamp is at least `effective_at`. A fee proposed at Unix time T can be applied from T + 172,800 onward; a split proposed at T can be applied from T + 604,800; an unbond requested at T can be withdrawn from T + 2,592,000.

Change	Proposed by	Delay	Applied by	Cancelled by	Pending slots
Protocol fee rate	Admin, <code>propose_fee</code>	48 hours	Anyone, <code>apply_fee</code>	Admin, <code>cancel_fee</code>	One
Arbitration, deposit, caps, forfeiture recipient	Admin, <code>propose_dispute_config</code>	48 hours	Anyone, <code>apply_dispute_config</code>	Admin, <code>cancel_dispute_config</code>	One
Runtime allowlist entry	Admin, <code>allow_runtime</code>	48 hours before attestations may use it	Takes effect at <code>effective_at</code>	Admin, <code>revoke_runtime</code>	One per measurement
Model split	Owner, <code>propose_split</code>	7 days	Anyone, <code>apply_split</code> , then each lane at its boundary	Owner, <code>cancel_split</code>	One per model
Bond withdrawal	Operator, <code>request_unbond</code>	30 days	Operator, <code>withdraw_bond</code>	Operator, <code>cancel_unbond</code>	One per operator

The arbitration authority changes only through the dispute-configuration path, so a newly proposed arbitrator waits 48 hours, is visible on chain throughout, and cannot rule until the change applies. Combined with the separation checks, the admin can neither appoint itself arbitrator nor pair a settings change with an immediate ruling.

These actions take effect in the transaction that carries them:

Action	Authority	Effect
<code>revoke_runtime</code>	Admin	No new attestation for the measurement
<code>revoke_attestation</code>	Admin or active verifier	Record latched revoked, sequence advanced
<code>freeze_operator</code>	Admin	Operator blocked from deposits, unbonding, registration and attestation
<code>rule_dispute</code> , <code>arbitrate_operator</code>	Arbitration	Slash reserved, operator frozen and attestation epoch advanced as ordered

Action	Authority	Effect
<code>set_paused</code>	Admin	Gated instructions stop
<code>set_tier</code> , <code>set_settlement_policy</code>	Admin	Tier or claim policy changes
<code>set_buyback_authority</code>	Admin	A new key may sign <code>buyback_swap_and_burn</code>
<code>set_buyback_mint</code>	Admin, once	The buyback mint is fixed permanently
<code>update_price</code> , <code>update_policy</code>	Model owner	New calls are priced, and nodes judged, under the new values

The pause flag stops the instructions that create registry entries, change a model's terms, allocate revenue or spend the protocol fee: `register_operator` , `deposit_bond` , `register_platform` , `register_node` , `record_attestation` , `create_model` , `update_price` , `update_policy` , `propose_split` , `apply_split` , `open_lane` , `distribute` and `buyback_swap_and_burn` . All other instructions run while paused, including unbonding and withdrawal, cancellations, revocations, freezes, governance proposals and applications, buyback settings, claim preparation, deferred claims and protocol-fee claims, dispute admission, rulings, arbitration actions, slash settlement and dispute claims.

7.5 Errors

The program defines 58 error codes, numbered from 6000 in Anchor's custom-error range:

Code	Name	Raised when
6000	<code>Unauthorized</code>	Signer is not the required authority (admin; admin or active verifier; admin or arbitration; arbitration; upgrade authority at <code>initialize</code> ; model owner as claimant; forfeiture beneficiary; buyback authority), a buyback authority is the default key, an arbitration relationship is partial or mismatched, a counter belongs to another pair, or a program write targets a non-writable or foreign account
6001	<code>Paused</code>	A gated instruction runs while <code>Config.paused</code> is set
6002	<code>ConfigurationUnset</code>	A required setting is unset: tier minima count, unknown tier, claim policy, fee rate, dispute deposit, arbitration, deposit policy or cap, buyback mint (or a mint other than <code>Config.buyback_mint</code> supplied); invalid dispute proposal values
6003	<code>Overflow</code>	Checked arithmetic overflows or underflows, distribution conservation fails, or a buyback total overflows
6004	<code>InvalidBps</code>	Fee or derivative royalty above 10,000 bps
6005	<code>InvalidTier</code>	Tier minima at <code>initialize</code> are not tiers 1, 2, 3 in order; model <code>min_tier</code> outside 1 to 3 at creation or in <code>update_policy</code>

Code	Name	Raised when
6006	<code>InvalidVerifierSet</code>	Verifiers are not five distinct non-default keys at threshold three; attestation epoch differs from the active epoch or the set is retired
6007	<code>InvalidMint</code>	The mint at <code>initialize</code> does not have six decimals
6008	<code>InvalidAmount</code>	Zero bond deposit or zero-bond unbond request; vault below recorded bond; vault below reserved amount; zero price at creation or in <code>update_price</code> ; slash above bond or a dismissal with sanctions; credit underflow; zero buyback amount or <code>min_out</code> , or a buyback amount above the buyback vault balance
6009	<code>OperatorFrozen</code>	The operator is frozen
6010	<code>InsufficientBond</code>	The bond is below the tier minimum
6011	<code>Unbonding</code>	A deposit or a second request while unbonding
6012	<code>OpenDispute</code>	Unbonding or withdrawal with an open dispute
6013	<code>UnbondNotRequested</code>	Cancellation or withdrawal without a request
6014	<code>TimelockPending</code>	Withdrawal before <code>unbond_at</code> , or an apply before <code>effective_at</code>
6015	<code>InvalidGpuSet</code>	GPU list empty, above eight, duplicated, off-platform or without matching accounts; malformed GpuPin
6016	<code>PlatformNotRegistered</code>	The platform is not registered
6017	<code>InvalidEndpoint</code>	Node endpoint or key broker URL is not a valid HTTPS URL within 128 bytes, or the key broker key is default
6018	<code>BoundsExceeded</code>	Runtime metadata out of bounds; allowed runtimes not 1 to 8 distinct; allowlist above 16 or not distinct, at creation or in <code>update_policy</code>
6019	<code>RuntimeAlreadyAllowed</code>	<code>allow_runtime</code> on a runtime that is allowed or pending
6020	<code>RuntimeNotAllowed</code>	Runtime revoked or before <code>effective_at</code>
6021	<code>NodeDisabled</code>	Node status is not <code>Registered</code>
6022	<code>AttestationMismatch</code>	Commitment measurement or platform differs from the accounts
6023	<code>InvalidTcb</code>	TCB status not <code>UpToDate</code> or method not <code>Committee</code>
6024	<code>AttestationExpired</code>	Expiry not in the future or beyond 24 hours
6025	<code>AttestationReplay</code>	Attestation sequence or operator epoch wrong; dispute sequence not the next one
6026	<code>InvalidRecentSlot</code>	Slot and hash pair absent from SlotHashes
6027	<code>InvalidSignatureInstruction</code>	Malformed Ed25519 precompile instruction or offsets
6028	<code>WrongAttestationMessage</code>	A verifier signed bytes other than the canonical message
6029	<code>UnknownVerifier</code>	Signing key is not a member of the set

Code	Name	Raised when
6030	<code>DuplicateVerifier</code>	A member's signature appears more than once
6031	<code>InsufficientSignatures</code>	Fewer than three distinct members signed
6032	<code>InvalidSplit</code>	Split not 1 to 8 entries with positive weights, distinct non-default ATAs and a 10,000-bps total
6033	<code>InvalidOperatorPayout</code>	Split does not contain exactly one Operator-role entry
6034	<code>InvalidBaseModel</code>	Base model account missing, mismatched or supplied without a reference
6035	<code>RoyaltyRequired</code>	Derivative split lacks the locked royalty to the base owner's canonical ATA
6036	<code>PendingSplitExists</code>	A split, fee or dispute-configuration proposal is already pending
6037	<code>LockedSplit</code>	A locked entry is reduced, removed or unlocked without its beneficiary's signature
6038	<code>NoPendingSplit</code>	No split, fee or dispute-configuration proposal to cancel or apply
6039	<code>InvalidRecipients</code>	Recipient account missing, of the wrong program, mint or owner, or not canonical; distribution accounts wrong in count, order or writability; treasury binding or forfeiture account not canonical
6040	<code>AuthoritiesMustDiffer</code>	Arbitration equals the admin or the default key, or the forfeiture owner equals arbitration
6041	<code>GovernanceTimelockRequired</code>	<code>set_dispute_config</code> is called
6042	<code>DisputeLimitReached</code>	The operator-wide or per-claimant open-dispute cap is reached
6043	<code>RescueDisabled</code>	<code>rescue_unopened_lane</code> is called
6044	<code>InvalidClaimDestination</code>	Claim destination not owned by the beneficiary, of another mint, or not canonical under <code>CanonicalAta</code>
6045	<code>AttestationRevoked</code>	Recording while the revocation latch is set
6046	<code>AttestationNotRevoked</code>	Clearing a record that is not revoked
6047	<code>GpuAlreadyClaimed</code>	A GpuPin already exists, or the GPU belongs to another platform or node
6048	<code>InvalidClaim</code>	Claim PDA or contents mismatch; recipient neither in the lane split nor the buyback vault; unpayable positive share or fee without a Claim; reserve below claim
6049	<code>DisputeAlreadyRuled</code>	Ruling a dispute that is not open; settling or claiming one that is open; claiming a forfeiture of one not dismissed
6050	<code>AttestationBeyondUnbond</code>	Expiry later than the operator's unbond deadline
6051	<code>OperatorLockUnsupported</code>	An Operator-role split entry is marked locked
6052	<code>LaneSplitBalanceNotEmpty</code>	Split activation while unallocated revenue remains in the vault
6053	<code>InsufficientPaidUsage</code>	Unused eligible paid fees on the lane are below the dispute deposit

Code	Name	Raised when
6054	<code>BuybackMintAlreadySet</code>	<code>set_buyback_mint</code> after a mint has been selected
6055	<code>UnsupportedBuybackMint</code>	The buyback mint is USDC, has a mint authority, has zero supply or more than 1,000,000,000 tokens at its decimals, or is a Token-2022 mint with an extension other than <code>MetadataPointer</code> and <code>TokenMetadata</code>
6056	<code>InvalidBuybackRoute</code>	The swap program is not Jupiter v6; the route is not an exact-input <code>route</code> or <code>shared_accounts_route</code> ; its input differs from <code>amount</code> ; its platform fee is not zero; a fixed route account is out of place; a route account other than the protocol PDA signs
6057	<code>InvalidBuybackSettlement</code>	A buyback account has a delegate or close authority or changes owner or mint; the swap changes the mint supply; the vault debit differs from <code>amount</code> ; the output is below <code>min_out</code> ; the burn leaves the output balance or the supply off its expected value

7.6 Upgrade authority and program deployment model

The program is deployed through Solana's upgradeable BPF loader. Its executable lives in a `ProgramData` account that records the upgrade authority, the only key that can replace the executable.

At launch the upgrade authority is the deployer, and the program binds its own initialization to that key. The `initialize` accounts require `program` to be the `sealroom` program whose `ProgramData` address is the supplied `program_data`, and `program_data.upgrade_authority_address` to equal the `admin` signer. Because `Config` sits at the fixed seed `config` and is created with `init`, initialization happens exactly once, and only the holder of the upgrade authority can perform it. A party watching the deployment cannot initialize first with its own mint, fee or verifier set. The launch transaction writes the configuration described in 7.3.2: the USDC mint, the genesis hash, the protocol fee of 500 bps, the three tier bond minima, the five-member epoch-0 committee at threshold three, the optional arbitration key, and the signer as `Config.admin`. The admin then names the buyback authority with `set_buyback_authority` and selects the buyback mint with `set_buyback_mint`; the buyback vault's address is fixed by derivation, and the vault accumulates fees until the mint is selected.

After launch the upgrade authority passes to governance: the loader's authority-transfer instruction moves it from the deployer to the governance multisig, a Squads multisig operated with timelocks, and from then on a new executable can be written only with that multisig's approval. A Squads authority signs program instructions through the Squads program, so the same signing path serves every governance role. Arbitration is a separate named multisig, supplied at initialization or installed through the 48-hour dispute-configuration path, and each role's powers inside the program are exactly those listed in 7.3 and 7.4.

Upgrades keep the wire format. The upgrade procedure compares the instruction arguments, the account fields and their order, and the signer requirements of the new build with the deployed interface, and refuses a build that changes them. Retired behavior stays in the interface with fixed results instead of being removed: `set_dispute_config` and `rescue_unopened_lane` remain callable

and always fail with their own error codes, `rescue_policy` and `split_settlement_policy` remain in `Config` and always hold `None`, the treasury binding remains in `propose_fee`, `PendingFee` and `Config.fee_treasury_ata` and `fee_treasury_owner` without routing any fee, the distribution account keeps the name `fee_treasury` while its address is fixed to the buyback vault, and the `UnopenedLaneRescued` event type remains decodable. Every client, indexer and historical transaction therefore keeps decoding against the current IDL. Each account carries a `version` byte, 1 for every account in this release, as the marker for any future layout.

The program identity is part of what verifiers sign. Bytes 16 to 47 of the canonical attestation message carry the program id and bytes 48 to 79 the genesis hash stored in `Config`, so committee signatures produced for this deployment are valid nowhere else.

7.7 Transaction sizes and compute

Legacy and v0 Solana transactions are limited to 1,232 bytes on the wire, and every transaction to 1.4 million compute units. Three of the program's transactions carry enough data that their size is a design input: `record_attestation`, with its committee signatures, `distribute`, with its per-recipient accounts, and `buyback_swap_and_burn`, with the Jupiter route it carries.

Attestation. A committee attestation is one transaction with two instructions. The first is a single native Ed25519 instruction carrying all three signatures over one shared copy of the 200-byte message; the second is `record_attestation`:

Component	Bytes
Ed25519 header (count and padding)	2
Three offsets entries of 14 bytes	42
Three public keys and signatures of 96 bytes	288
Shared canonical message	200
Ed25519 instruction data	532
<code>record_attestation</code> discriminator	8
<code>AttestationCommitment</code>	194
Epoch, sequence and recent slot	24
Recent slot hash	32
<code>record_attestation</code> instruction data	258

`record_attestation` names thirteen accounts, and the transaction also references the Ed25519 program. In a legacy message every one of those addresses costs 32 bytes, and the complete transaction is 1,361 bytes, over the 1,232-byte limit. The coordinator therefore sends it as a v0 transaction with an address lookup table that holds the instruction's fixed accounts: `Config`, the `VerifierSet`, the `Operator`, `Platform`, `Node`, `Runtime` and `Attestation` accounts, the `Instructions` and

SlotHashes sysvars and the System Program. Each looked-up account costs a one-byte index. The fee payer and the invoked programs stay in the static key list.

Distribution. `distribute` takes no arguments, so its instruction data is the eight-byte discriminator, but it names twelve accounts plus two per split entry: 28 accounts for eight recipients. It performs one `transfer_checked` CPI for the fee into the buyback vault and one for each payable share, plus the self-CPIs that record its events. Healthy payouts and zero-amount shares use empty placeholder addresses in their Claim slots, so a healthy distribution allocates no Claim rent. With the lane's accounts in a lookup table, one signer and a compute-budget instruction, the eight-recipient transaction is 308 bytes.

Buyback. `buyback_swap_and_burn` carries Jupiter's route instruction data as its `swap_data` argument and the route's accounts as remaining accounts, so its size follows the route. The keeper loads Jupiter's returned lookup tables together with the deployment's own. A 10-USDC route with 36 bytes of route data and 24 route accounts, built this way, serializes to 459 bytes.

Measured on the compiled program:

Transaction	Format	Bytes	Compute units
<code>record_attestation</code> , three signatures	v0 + lookup table	1,056	49,314
<code>record_attestation</code> as sent by the coordinator	v0 + lookup table	1,087	46,465 to 46,505
<code>record_attestation</code> , three signatures	Legacy	1,361	Exceeds the 1,232-byte limit

Serialized with the generated interface:

Transaction	Format	Bytes
<code>distribute</code> , buyback vault and eight recipients	v0 + lane lookup table, compute-budget instruction	308
<code>buyback_swap_and_burn</code> , 10-USDC Jupiter route, 24 route accounts	v0 + Jupiter and protocol lookup tables	459

Compute varies slightly with the addresses involved, because each PDA derivation searches for its bump; the coordinator figure is the range observed across submissions.

Transaction builders follow the same budget. They place idempotent ATA-creation instructions in the same transaction as `create_model` or `propose_split` when the packet allows. Splits whose eight recipient ATAs are all new use a configured lookup table or a separate account-setup transaction, and a preview that would exceed 1,232 bytes is returned to the wallet as blocked instead of as a transaction it cannot submit. The keeper and the API load the deployment's configured lookup tables before building, so the attestation and distribution transactions above are the forms the service submits.

8. Payments over x402

A call on Sealroom is an HTTPS request to a node's registered endpoint, paid in USDC over x402 version 2. The caller needs no account with the operator, the owner or Sealroom; a Solana wallet holding USDC is enough, so AI agents can pay. Every payment for a model served by a given node lands in one token account, the vault of the lane for that (model, node) pair. No Sealroom instruction runs per call: the split between protocol, operator and contributors happens later, in one `distribute` per lane (chapter 9).

The **caller** signs a USDC transfer for the quoted amount, filling only its own signature slot. The **node runtime** quotes the price, has the payment verified, runs inference inside the enclave and has the payment settled; it holds no facilitator or channel custody key. The **facilitator** issues the requirements, verifies the signed transaction, pays the network fee as fee payer, broadcasts, confirms and settles usage vouchers.

8.1 Parties and endpoints

Party	Endpoint	Purpose
Node runtime	<code>POST /v1/chat/completions</code> at <code>Node.endpoint</code> (HTTPS, at most 128 bytes)	Answers 402 without payment; serves and settles with payment
Facilitator	<code>POST /requirements</code>	Creates a payment session bound to the request and returns the x402 requirements
Facilitator	<code>POST /verify</code>	Checks the signed payment against the session without moving funds
Facilitator	<code>POST /settle</code>	Adds the fee-payer signature, broadcasts, confirms and checks the token balance change
Facilitator	<code>POST /settle-voucher</code>	Settles an <code>upto</code> session against an enclave-signed usage voucher and refunds the unused ceiling
Facilitator	<code>POST /reconcile</code>	Refunds expired <code>upto</code> ceilings and reconciles ambiguous payment intents; the keeper calls it

All write endpoints are rate-limited per client address and answer `429` with `Retry-After` when the limit is reached. Browser callers are admitted only from exact configured origins.

8.2 Reading the attestation before the call

Before it sends a prompt, the caller establishes from Solana that the node is attested and that the TLS session it is about to open terminates inside the attested enclave. The node's runtime generated three key pairs inside the CVM at boot: an HPKE key pair (X-Wing: ML-KEM-768 plus X25519), a TLS key pair and an Ed25519 voucher key pair. The hash of their public keys is the `enclave_key_hash` that three verifiers signed into the node's `Attestation` account.

```
transcript = hpke_public_key (1216 bytes) || tls_public_key (32 bytes)
            || voucher_public_key (32 bytes)
enclave_key_hash = SHA-256(transcript)
tls_key_hash = SHA-256(tls_public_key)
report_data = SHA-256(transcript || recent_slot_hash)
```

The caller's SDK performs these steps:

1. Derive the accounts: `Model` at `["model", owner, id:u64]`, `Node` at `["node", platform, index:u16]`, `Attestation` at `["att", node]`, `Lane` at `["lane", model, node]`.
2. Read the `Attestation` and check that it is current: `revoked_at` is empty, `expires_at` is in the future (the program never accepts an expiry more than 86,400 seconds after recording), `tcb_status` is `UpToDate`, and `operator_epoch` equals the operator's current `attestation_epoch`.
3. Check the model's policy against the node: the attestation's 48-byte `runtime_measurement` is on the model's `allowed_runtimes` and its `Runtime` account is active; the platform is registered to the node's operator; the operator is not frozen, its tier is at least `min_tier`, its bond is at least the larger of `min_bond` and its tier minimum, and it is on the model's `operator_allowlist` when that list is non-empty; the node is not disabled; the protocol is not paused. The registry API serves the same rules at `/v1/eligibility?modelId=...&nodeId=...` with one reason code per failed check, such as `attestation_revoked`, `runtime_not_allowed` or `operator_frozen`.
4. Obtain the node's public-key transcript, recompute `enclave_key_hash` and compare it with the 32 bytes stored in the `Attestation` account. The registry publishes `tlsPublicKeyHash` and `voucherPublicKey` for an attestation only after this recomputation succeeds against the chain value.
5. Open TLS to `Node.endpoint`, hash the server's 32-byte TLS public key and compare it with `tls_key_hash`. On a mismatch the caller aborts before sending the prompt.
6. Keep the voucher public key. For `upto` calls it is the only key whose usage vouchers the caller accepts.
7. Read the model's `price` and the lane's `vault_ata`.

The on-chain price is a Borsh enum in the `Model` account:

Variant	Fields	Validation at <code>create_model</code> and <code>update_price</code>
Exact	<code>per_call: u64</code>	<code>per_call > 0</code>

Variant	Fields	Validation at <code>create_model</code> and <code>update_price</code>
Upto	<code>input_per_million_tokens: u64</code> , <code>output_per_million_tokens: u64</code> , <code>max_per_call > 0</code> <code>max_per_call: u64</code>	

All amounts are USDC base units with 6 decimals, so 1,000,000 base units are 1 USDC. The owner replaces the price with `update_price`, which takes effect in the transaction that carries it; every session opened after that is quoted from the new price.

8.3 The request and the 402 challenge

The caller posts the call to the node with a caller-chosen `requestId` (a UUID that serves as idempotency key), the `modelId`, `nodeId` and `laneId` read in 8.2, the paying wallet as `caller`, the `scheme` (`exact` or `upto`), for `upto` a `ceiling` in base units, and the `prompt`.

The node hashes the prompt, `bodyHash = SHA-256(prompt)`, and forwards everything except the prompt, together with `bodyHash`, to the facilitator's `/requirements`. The facilitator then:

1. Confirms that its RPC endpoint reports the configured genesis hash.
2. Reads the registry accounts from chain and runs the eligibility rules. An ineligible node fails with `NODE_INELIGIBLE`: followed by the reason codes.
3. Reads the `Lane` and checks that `lane.model` and `lane.node` are the requested model and node (`LANE_IDENTITY_MISMATCH`).
4. Computes the amount from the scheme and the on-chain price:

```
exact:                amount = per_call
upto, Upto-priced model: 1 <= ceiling <= max_per_call and ceiling
                        <= facilitator_max_ceiling
upto, Exact-priced model: per_call <= ceiling <=
                        facilitator_max_ceiling
```

A metered (`Upto`) model cannot be bought with `exact`; the request fails with `METERED_MODEL_REQUIRES_UPTO`. A ceiling outside its bounds fails with `PAYMENT_CEILING_INVALID`.

5. Derives the session key that makes the call idempotent:

```
request_id_hash = SHA-256(JSON([genesis_hash, requestId, modelId,
                                nodeId, caller]))
```

If a session with this hash exists, the facilitator returns its stored requirements unchanged, provided the scheme, amount, `bodyHash` and lane are the same; otherwise it answers

`IDEMPOTENCY_CONFLICT` . A retried request therefore gets the same invoice and can never be charged twice.

6. Sets the session expiry to the chain clock plus 120 seconds and takes a recent blockhash.

7. Stores the session with status `offered` , the requirements, the attestation account and its signed revision, and, for `upto` , the voucher public key read from the attested evidence.

The requirements object is the x402 `PaymentRequirements` record:

Field	Value
<code>scheme</code>	<code>exact</code> or <code>upto</code>
<code>network</code>	<code>solana</code> : followed by the first 32 characters of the cluster's genesis hash
<code>asset</code>	The USDC mint from <code>Config.usdc_mint</code>
<code>payTo</code>	The lane vault: <code>Lane.vault_ata</code> , the USDC token account owned by the lane PDA
<code>amount</code>	Base units: <code>per_call</code> for <code>exact</code> , the ceiling for <code>upto</code>
<code>maxTimeoutSeconds</code>	120
<code>extra.feePayer</code>	The facilitator's fee-payer public key
<code>extra.memo</code>	<code>sealroom:<sessionId></code>
<code>extra.recentBlockhash</code> , <code>extra.lastValidBlockHeight</code>	The blockhash the payment must use and its validity bound
<code>extra.sessionId</code>	UUID of the payment session
<code>extra.laneId</code>	The <code>Lane</code> account
<code>extra.channelId</code> , <code>extra.escrow</code>	<code>upto</code> only: the payment channel and its escrow token account

`payTo` is the vault token account itself. A wallet must send to it directly and must not treat it as an owner address from which to derive another associated token account.

When the request carries no `PAYMENT-SIGNATURE` header, the node answers `402 Payment Required` . The body and the `PAYMENT-REQUIRED` header carry the same JSON, the header as base64:

```

{
  "x402Version": 2,
  "resource": { "url": "https://<node endpoint>/v1/chat/completions", "mimeType":
"application/json" },
  "accepts": [{
    "scheme": "exact",
    "network": "solana:<genesis hash prefix>",
    "asset": "<USDC mint>",
    "payTo": "<lane vault token account>",
    "amount": "<per_call>",
    "maxTimeoutSeconds": 120,
    "extra": {
      "feePayer": "<facilitator fee payer>",
      "memo": "sealroom:<sessionId>",
      "recentBlockhash": "<blockhash>",
      "lastValidBlockHeight": "<height>",
      "sessionId": "<sessionId>",
      "laneId": "<lane>"
    }
  }]
}

```

8.4 The payment transaction

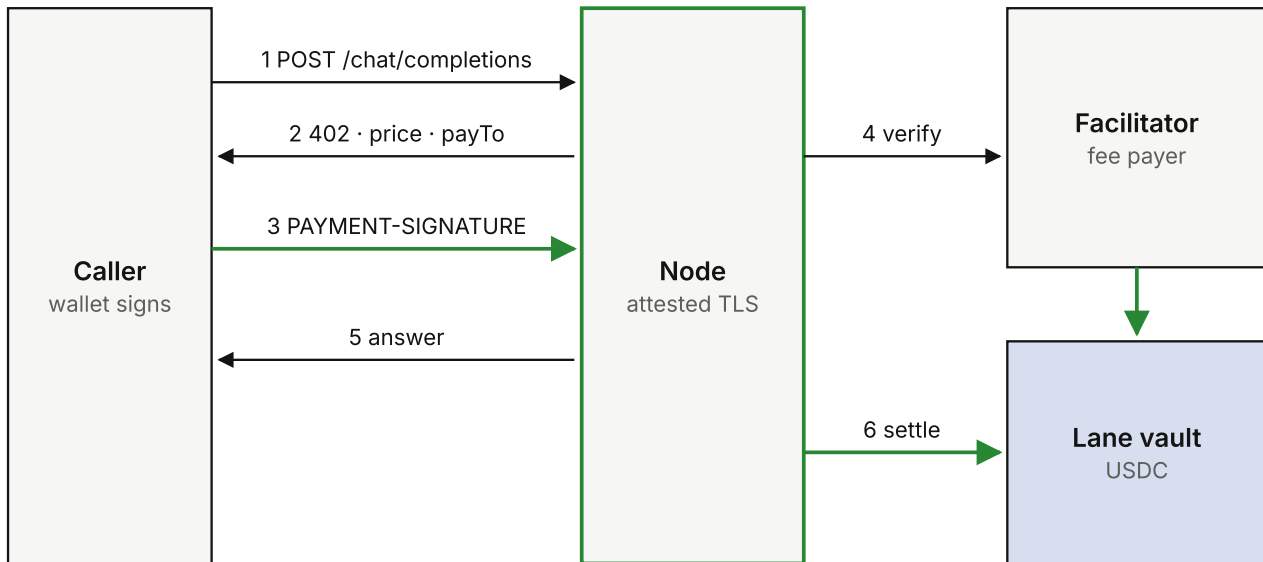
The caller builds exactly one transaction shape. The facilitator rejects anything else, which keeps it compatible with the fixed instruction layout that SVM x402 facilitators validate around a single `TransferChecked` . The SDK function `buildX402PaymentTransaction(requirements, caller)` produces it.

#	Program	Instruction	Data	Accounts
0	<code>ComputeBudget11111111111111111111111111111111</code>	<code>SetComputeUnitLimit</code>	5 bytes: tag <code>0x02</code> , u32 LE <code>100000</code>	none
1	<code>ComputeBudget11111111111111111111111111111111</code>	<code>SetComputeUnitPrice</code>	9 bytes: tag <code>0x03</code> , u64 LE <code>0</code>	none
2	<code>TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA</code>	<code>TransferChecked</code>	10 bytes: tag <code>0x0c</code> , u64 LE amount , u8 <code>6</code>	caller's USDC associated token account (writable), USDC mint, destination (writable), caller (signer)

#	Program	Instruction	Data	Accounts
3	MemoSq4gqABAXKb96qnH8TysNcWxMyWCqXgDLGmfchR	Memo	UTF-8 sealroom: <sessionId> (45 bytes)	none
The destination of instruction 2 is <code>payTo</code> (the lane vault) for <code>exact</code> and <code>extra.escrow</code> for <code>upto</code>. The message is a v0 message whose fee payer is <code>extra.feePayer</code>, so the caller needs no SOL. It uses <code>extra.recentBlockhash</code>, no address lookup tables and exactly two signatures, the facilitator's in slot 0 and the caller's in slot 1, and it serializes to between 100 and 1,232 bytes. The caller's USDC associated token account must already exist; the transaction contains no account-creation instruction. The wallet signs with <code>signTransaction</code>, filling only the caller's slot. It never uses <code>sign-and-send</code>: the facilitator broadcasts after its own checks. The caller wraps the signed transaction in an x402 payment payload and retries the identical request with the header <code>PAYMENT-SIGNATURE: base64(JSON(payload))</code>. The node accepts a header of at most 8,192 characters.				
<pre>{ "x402Version": 2, "resource": { "url": "https://<node endpoint>/v1/chat/completions" }, "accepted": { "...": "the requirements object from the 402, unchanged" }, "payload": { "transaction": "<base64 of the partially signed v0 transaction>" } }</pre>				
The facilitator accepts the transaction only when every check below passes. Each failure has its own error code.				
Check	Rule	Error		
Wire	100 to 1,232 bytes, canonical base64, no lookup tables	PAYMENT_WIRE_INVALID		
Sponsor policy	Payer is the facilitator, two required signatures, four instructions, quoted blockhash	SPONSOR_POLICY_REJECTED		
Fee-payer isolation	No instruction names the fee payer as program or account	FEE_PAYER_NOT_ISOLATED		
Compute budget	Limit exactly 100,000 units, price exactly 0 micro-lamports	COMPUTE_POLICY_REJECTED		
Memo	Memo program, no accounts, data equals <code>extra.memo</code>	PAYMENT_MEMO_MISMATCH		
Transfer	Amount equals the session amount, 6 decimals, USDC mint, destination as quoted, owner is the caller, source is the caller's associated token account, exactly four accounts	PAYMENT_TRANSFER_MISMATCH		
Caller signature	Caller is one of the two signers and its Ed25519 signature verifies over the message bytes	CALLER_SIGNATURE_INVALID		

Fee-payer isolation means the facilitator's signature can only pay the network fee. Nothing in the transaction can move the facilitator's own lamports or tokens.

8.5 Verify, serve, settle



`TransferChecked` → `lane vault · memo` binds the request

Figure: an `x402` call from the attestation read through the 402 challenge, verification, inference and settlement into the lane vault.

The node orders the paid call so that the caller is charged only for a completion that an eligible enclave has already produced, and so that a completion that has been paid for is always delivered.

- Challenge.** The caller sends the request without a payment header and receives the 402 described in 8.3.
- Signed retry.** The caller signs the payment transaction and repeats the identical request with `PAYMENT-SIGNATURE`.
- Binding.** The node fetches the requirements for this request again (the facilitator returns the stored session) and requires the payload's `accepted` object to equal them field for field (`PAYMENT_REQUEST_BINDING_MISMATCH`). A payment signed for another request, lane or amount is rejected here.
- Verify.** The node posts the payload to `/verify`. The facilitator checks `x402Version` is 2, the resource URL is the session's, `accepted` equals the stored requirements, the transaction matches any earlier submission for the session, the session is in a payable state and unexpired, the node is still eligible, and the wire passes every check in 8.4. It returns `{ isValid: true, payer, sessionId }`. It signs nothing and moves no funds.
- Serving check.** The node reads chain state and requires that it is still eligible and that the attestation account and signed revision are the ones its data-key release was bound to (`NODE_KEY_RELEASE_STALE` otherwise).

6. **Inference.** The runtime executes the call inside the enclave.
7. **Second serving check.** The node repeats step 5 with fresh chain state.
8. **Settle.** The node posts the same payload to `/settle`. The facilitator validates it again, adds its fee-payer signature, and stores the signed bytes and their hash with status `processing` before it broadcasts. It then broadcasts, waits for `confirmed` commitment, and compares the destination account's pre- and post-transaction token balances in the confirmed transaction: the increase must equal the session amount exactly (`PAYMENT_POST_BALANCE_MISMATCH`). The session moves to `settled` for `exact` and to `funded` for `upto`.
9. **Voucher (`upto` only).** The enclave signs a usage voucher and the node posts it to `/settle-voucher` (8.7).
0. **Response.** The node returns `200` with the completion and a `PAYMENT-RESPONSE` header carrying the base64 JSON receipt `{ success, transaction, network, payer, sessionId }`. For `upto` the body also carries the voucher and its signature, so the caller can check the charge against the voucher key it read in 8.2.

If the node loses eligibility at step 5 or step 7, the call ends with an error before step 8, and nothing has been broadcast. Once step 8 has confirmed, the node returns the completion it already holds, even if the attestation is renewed or revoked in the meantime.

The facilitator records each session in one of these states:

Status	Meaning
<code>offered</code>	Requirements issued; no payment received
<code>processing</code>	Signed payment stored; broadcast or confirmation in progress
<code>funded</code>	<code>upto</code> : ceiling confirmed in escrow, voucher pending
<code>settling</code>	<code>upto</code> : voucher settlement transaction stored, confirmation in progress
<code>settled</code>	<code>exact</code> : transfer confirmed. <code>upto</code> : usage paid to the lane vault and remainder refunded
<code>refunding</code>	<code>upto</code> : expiry refund transaction stored, confirmation in progress
<code>refunded</code>	<code>upto</code> : full ceiling returned to the caller
<code>expired</code>	The session passed its expiry and no transfer landed

Every state change is a compare-and-set on the previous status and payload, and signed bytes are stored before any broadcast. A retry first looks the stored signature up in transaction history and rebroadcasts the same bytes only while their blockhash is valid. A settled session returns its stored receipt; a refunded or expired one is never accepted as proof of payment.

8.6 The `exact` scheme

`exact` is a fixed price per call. The caller's `TransferChecked` moves `per_call` straight into the lane vault, and settlement is the confirmation that the vault balance rose by exactly that amount. The

vault is the USDC associated token account of the lane PDA, so only the `sealroom` program, signing for the lane through `distribute`, `claim_deferred` and `claim_protocol_fee`, can move funds out of it. Whatever the model's split, each call stays one transfer to one recipient; an eight-recipient split costs nothing extra per call and is paid for once per `distribute`.

8.7 The `upto` scheme

`upto` is metered billing under a ceiling. The caller escrows the ceiling in a payment channel of the Solana Foundation payment-channels program (`CHNLxYvVA28MJP9PrFuDXccuoGXA7jBacfLEkahyGsX`). After serving, the enclave signs a voucher for the actual charge, the facilitator settles that amount to the lane vault and returns the rest to the caller. The lane vault is the channel's only payee; the split stays in `sealroom`.

The requirements for an `upto` session add two fields. `extra.escrow` is the channel's escrow token account, and it is the destination of the caller's `TransferChecked`. `extra.channelId` identifies the channel for this session:

```
channel_id = SHA-256(request_id_hash_hex || session_id)    (32 bytes,
base58)
```

The caller signs the same four-instruction transaction with `amount` set to its ceiling. Settling it moves the ceiling into escrow, and the session becomes `funded`.

Usage price

The enclave counts the call's input and output tokens and prices them with integer arithmetic. The rates are USDC base units per million tokens.

```
raw = input_tokens * input_per_million_tokens + output_tokens *
output_per_million_tokens
amount = floor((raw + 999999) / 1000000)
require amount <= max_per_call
refund = ceiling - amount
```

The division rounds the whole call up once, so the charge exceeds the exact fractional price by less than one base unit. Token counts, rates and the result are checked against the u64 range, and a charge above `max_per_call` is refused (`PAYMENT_USAGE_EXCEEDS_MODEL_MAX`).

Worked example, with a sample price of 3,500,000 base units per million input tokens (3.50 USDC) and 10,000,000 per million output tokens (10 USDC), and a sample ceiling of 20,000 base units:

```

raw = 1235 * 3500000 + 401 * 10000000 = 4322500000 + 4010000000 =
8332500000
amount = floor((8332500000 + 999999) / 1000000) = floor(8333.499999) =
8333
refund = 20000 - 8333 = 11667

```

The lane vault receives 8,333 base units (0.008333 USDC) and the caller gets 11,667 back.

The usage voucher

The voucher is a JSON object signed by the enclave's Ed25519 voucher key. That key was generated inside the CVM, and its public key is one of the three inputs to the `enclave_key_hash` that the verifier committee signed. The operator never holds the private key, so it cannot produce a voucher for a larger charge.

Field	Type	Content
<code>domain</code>	string	<code>sealroom:usage:v1</code>
<code>laneId</code>	public key	The lane being paid
<code>channelId</code>	public key	The session's channel
<code>requestId</code>	32-byte hex	<code>request_id_hash</code> of the session
<code>nonce</code>	u64 string	<code>1</code> : one cumulative voucher per session
<code>cumulativeAmount</code>	u64 string	The charge in base units
<code>expiresAt</code>	ISO timestamp	The session expiry
<code>attestationId</code>	public key	The node's <code>Attestation</code> account
<code>attestationRevision</code>	u64 string	The attestation's <code>signed_sequence</code> at serving time
<code>signer</code>	public key	The enclave voucher public key

The signed message is the UTF-8 JSON serialization of these fields in table order, with `genesisHash` (the cluster genesis hash) inserted after `domain`. The 64-byte Ed25519 signature travels as base64.

`/settle-voucher` accepts the voucher only when all of these hold; otherwise it answers `VOUCHER_INVALID`:

1. The session is `upto`, funded, and has no refund in progress.
2. `laneId`, `channelId` and `requestId` equal the session's lane, channel and request hash, and `nonce` is `1`.
3. `attestationId` and `attestationRevision` equal the node's current attestation on chain and the revision stored when the session was offered.

4. `signer` equals the voucher key in the attested evidence. The facilitator loads the evidence archived under the on-chain `evidence_hash`, recomputes both `evidence_hash` and `enclave_key_hash`, and requires both to match the `Attestation` account and the evidence's node to be this node.
5. `expiresAt` is not in the past and not later than the session expiry.
6. `cumulativeAmount` is at most the escrowed ceiling.
7. The Ed25519 signature verifies over the message for `signer`.

The facilitator then settles in one transaction. A `TransferChecked` from escrow to the lane vault pays `cumulativeAmount` when it is positive; a second `TransferChecked` from escrow to the caller's USDC associated token account returns `ceiling - cumulativeAmount` when it is positive; a memo `sealroom:settle:<sessionId>` closes the instruction list. Payment and refund land together or not at all. The session stores the voucher hash: the same voucher submitted again returns the same receipt, and a different voucher for a settled session fails with `VOUCHER_REPLAY`.

A funded session whose expiry passes without a voucher is refunded in full. The keeper calls `/reconcile`, which sends the whole ceiling from escrow back to the caller with the memo `sealroom:expire:<sessionId>` and marks the session `refunded`. The same pass finishes interrupted payment and settlement transactions and marks a `processing` session `expired` when its transfer never landed.

8.8 Header reference

Header	Direction	Encoding	Content
<code>PAYMENT-REQUIRED</code>	Node to caller, with <code>402</code>	base64 of JSON	<code>{ x402Version: 2, resource, accepts: [requirements] }</code>
<code>PAYMENT-SIGNATURE</code>	Caller to node, on the retry	base64 of JSON	<code>{ x402Version: 2, resource: { url }, accepted, payload: { transaction } }</code>
<code>PAYMENT-RESPONSE</code>	Node to caller, with <code>200</code>	base64 of JSON	<code>{ success, transaction, network, payer, sessionId }</code>

For browser callers the node exposes `PAYMENT-REQUIRED`, `PAYMENT-RESPONSE` and `Retry-After` to scripts and accepts `PAYMENT-SIGNATURE` as a request header. A rejected call returns `{ success: false, errorReason }` carrying one of the codes named in this chapter.

9. Settlement and distribution

Payments accumulate in lane vaults. Distribution turns a vault balance into payouts: the protocol fee to the buyback vault, the operator's share to the operator that runs the node, and every other split entry to its recipient. One permissionless instruction, `distribute`, does this for one lane in one transaction, and either every transfer in it succeeds or none does.

9.1 Lanes

A lane exists for each (model, node) pair that serves traffic. Anyone can open one with `open_lane`; the payer signs and pays rent, and the protocol must not be paused. The instruction creates the `Lane` account at `["lane", model, node]` and the vault, the USDC associated token account whose authority is the lane PDA. If someone created that token account in advance, `open_lane` adopts it. The lane copies the model's current split and split revision into `active_split` and `active_split_revision`. Opening a lane gives the node no right to a data key; key release depends only on the attestation and the model's policy.

Field	Type	Bytes	Meaning
<code>discriminator</code>	<code>[u8; 8]</code>	8	Anchor account type
<code>version, bump</code>	<code>u8, u8</code>	2	Layout version and PDA bump
<code>model</code>	<code>Pubkey</code>	32	The <code>Model</code> account
<code>node</code>	<code>Pubkey</code>	32	The <code>Node</code> account
<code>vault_ata</code>	<code>Pubkey</code>	32	USDC token account owned by the lane PDA
<code>total_in</code>	<code>u64</code>	8	Revenue taken into allocation; equals <code>total_allocated</code> after every distribution
<code>total_distributed</code>	<code>u64</code>	8	USDC actually transferred out, including later claims
<code>last_distributed</code>	<code>Option<i64></code>	9	Time of the last distribution that allocated revenue
<code>total_allocated</code>	<code>u64</code>	8	Revenue assigned to recipients, paid or deferred
<code>reserved_amount</code>	<code>u64</code>	8	Deferred entitlements still held in the vault
<code>active_split_revision</code>	<code>u64</code>	8	Model split revision this lane currently pays under
<code>active_split</code>	<code>Vec<SplitEntry></code>	$4 + 8 \times 68$	The split this lane currently pays under
<code>eligible_protocol_fees_paid</code>	<code>u64</code>	8	Protocol fees that reached the buyback vault (chapter 10)

Field	Type	Bytes	Meaning
<code>admission_credit_consumed</code>	<code>u64</code>	8	Part of that amount already used to open disputes

The account occupies 719 bytes. Its counters obey fixed relations at all times:

```

available = vault.amount - reserved_amount
total_allocated = total_distributed + reserved_amount
payments_received = total_distributed + vault.amount

```

`available` is revenue that has arrived but has not yet been assigned. `reserved_amount` is revenue assigned to recipients who could not be paid directly; it stays in the vault until they claim it and is never allocated a second time.

9.2 Splits

A model's split lists up to eight recipients. Each entry is 68 bytes:

```

SplitEntry (68 bytes)
  recipient_ata  Pubkey  32  canonical USDC associated token account of the recipient
  recipient_owner Pubkey  32  the recipient wallet (beneficiary)
  bps            u16     2   weight in basis points
  locked         bool    1   reduction or removal needs the recipient's signature
  role           u8      1   0 Owner, 1 Operator, 2 BaseModel, 3 Contributor

```

The program accepts a split only when all of these hold:

1. It has between one and eight entries.
2. Every `bps` is positive and the weights sum to exactly 10,000.
3. No two entries share a `recipient_ata`, and no recipient is the default key.
4. Exactly one entry has the `Operator` role, and that entry is not locked.
5. Each `recipient_ata` is passed in and checked on chain: a token account of the USDC mint, owned by `recipient_owner`, at the canonical associated-token address for that owner.

The `operator` entry carries a weight, not a destination. At distribution the program replaces its address with the payout account of the operator that runs the lane's node: `Operator.payout_ata` as destination and `Operator.authority` as beneficiary. One model served by three operators therefore pays each operator from its own lanes, and the address the owner typed into the entry is never paid. Since that address cannot consent on behalf of every future operator, the role cannot be locked.

A `locked` entry protects a recipient against later changes. A new split proposal must either keep each locked entry, with the same account, owner, role, lock and at least the same `bps`, or carry the signature of that entry's `recipient_owner`. When a model is a derivative of a base model whose `derivative_royalty_bps` is positive, `create_model` requires a locked `BaseModel` entry paying at

least that royalty to the base owner's canonical USDC account. The base owner's key broker can then release its key only to lanes whose split carries the royalty.

9.3 The `distribute` instruction

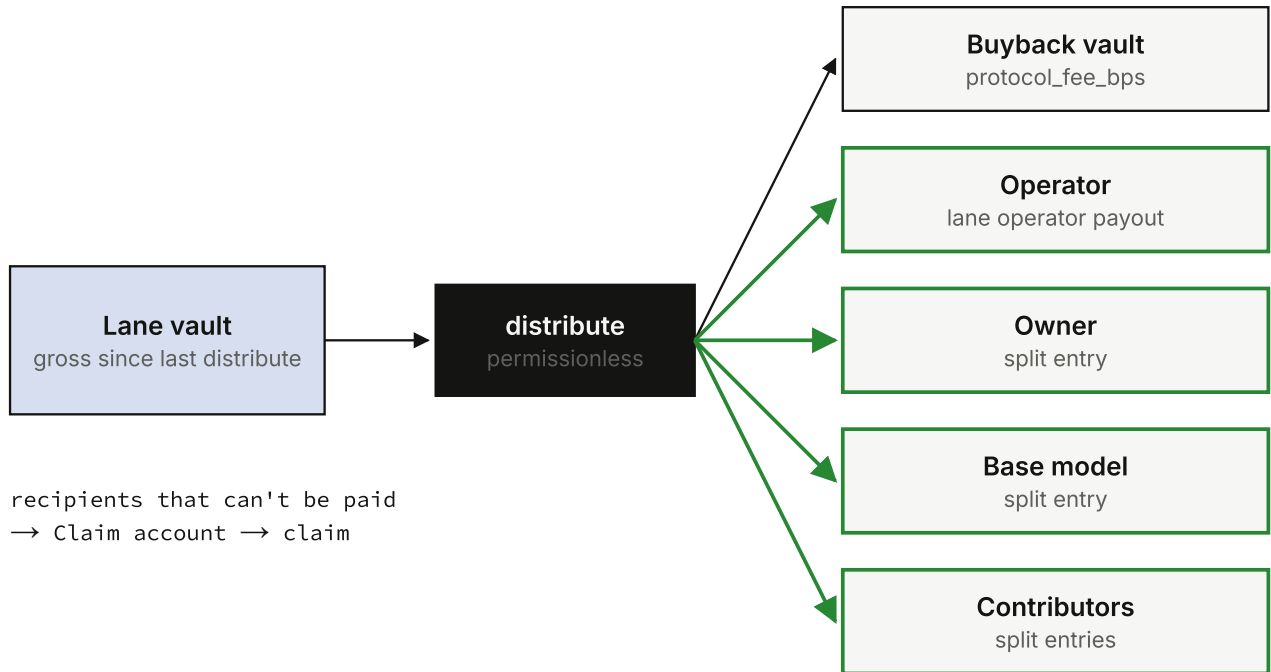


Figure: one `distribute` transaction splits a lane's available balance into the protocol fee and the split shares, paying usable accounts and reserving the rest as claims.

`distribute` takes no arguments and needs no particular signer; whoever submits it pays the network fee. Its accounts are:

#	Account	Constraint
0	<code>config</code>	<code>["config"]</code>
1	<code>model</code>	<code>["model", owner, id]</code>
2	<code>node</code>	<code>["node", platform, index], has_one = operator</code>
3	<code>operator</code>	<code>["op", authority]</code>
4	<code>lane</code> (writable)	<code>["lane", model, node], has_one = model, node, vault_ata = vault</code>
5	<code>usdc_mint</code>	<code>Config.usdc_mint</code>
6	<code>vault</code> (writable)	USDC associated token account with authority <code>lane</code>
7	<code>fee_treasury</code> (writable)	The buyback vault: canonical USDC ATA of the protocol PDA <code>["protocol"]</code> , not the vault
8	<code>fee_claim</code> (writable)	Claim PDA <code>["claim", lane, fee_treasury]</code>

#	Account	Constraint
9	token_program	SPL Token
10, 11	event_authority , program	Self-CPI event emission

After them come two remaining accounts per split entry, in split order: the entry's resolved recipient token account (writable) and its Claim PDA `["claim", lane, recipient_ata]`. A Claim PDA that has never been initialized is accepted as a placeholder when its recipient can be paid directly. Account 7 keeps its interface name, `fee_treasury`, and the program fixes its address to the buyback vault; no configuration can point it elsewhere.

The instruction runs these steps:

1. Require the protocol to be unpaused and a claim policy to be configured.
2. Resolve the lane's `active_split` against the node's operator, replacing the `Operator` entry's destination with `Operator.payout_ata` and its beneficiary with `Operator.authority`.
3. Compute `available = vault.amount - reserved_amount`. If it is zero, try to activate a newer split (9.8) and return without moving tokens or touching the counters.
4. Require exactly two remaining accounts per entry.
5. Compute the fee and every share from `available`, `Config.protocol_fee_bps` and the lane's `active_split` (9.4).
6. Before any transfer, check every pair: the token account is the resolved destination, writable and not the vault; an initialized Claim belongs to this lane, recipient and beneficiary; and every positive share is either payable directly or backed by an initialized Claim. Check the buyback vault the same way, with the protocol PDA as its beneficiary.
7. Pay the protocol fee into the buyback vault, or add it to the vault's Claim if the vault cannot receive it.
8. For each entry in order, transfer its share from the vault, signed by the lane PDA, or add it to the entry's Claim.
9. Require `paid + deferred = available`.
0. Update the lane: `total_allocated += available`, `total_distributed += paid`, `total_in = total_allocated`, `reserved_amount += deferred`, `last_distributed = now`, and `eligible_protocol_fees_paid += fee` when the fee was paid directly into the buyback vault.
1. Emit `LaneDistributed` with the gross amount, the fee and payouts split into paid and deferred parts, the fee rate, the buyback vault, the hash and revision of the active split, and one record per entry.
2. Reload the vault and activate a newer model split if the lane's revision is behind (9.8).

Any unexpected token-program error aborts the whole transaction, and the lane is left exactly as before.

9.4 Distribution arithmetic

All arithmetic is integer: products are formed in u128 and every counter update is a checked u64 operation.

```
A = vault.amount - reserved_amount
fee = floor(A * protocol_fee_bps / 10000)
net = A - fee
share_i = floor(net * bps_i / 10000)          for i = 1 .. n
share_n = share_n + (net - (share_1 + ... + share_n))
fee + share_1 + ... + share_n = A
```

The protocol fee rounds down, so its remainder stays with the recipients. Each split share rounds down, and the last entry in split order receives every base unit the floors left over. Because the weights sum to 10,000, the leftover is a whole number below n: at most 7 base units for an eight-entry split. The shares are computed from the split's own weights; resolving the `operator` destination changes where a share goes, never its size. Fee and shares together always equal `A` exactly, which `distribute` enforces before it writes the lane.

The fee rate is read from `Config` when `distribute` runs. The launch parameter is 500 bps, 5% of each distributed amount, and the whole fee goes to the buyback vault (14.7).

9.5 Worked example: a four-entry split

A lane has a vault balance of 123,456,789 base units (123.456789 USDC), nothing reserved, the launch fee of 500 bps, and this sample split in this order: owner 6,500 bps, operator 2,000 bps, base-model royalty 1,000 bps (locked), contributor 500 bps.

```
A    = 123456789 - 0 = 123456789
fee = floor(123456789 * 500 / 10000) = floor(6172839.45) = 6172839
net = 123456789 - 6172839 = 117283950
owner      = floor(117283950 * 6500 / 10000) = floor(76234567.5) =
76234567
operator   = floor(117283950 * 2000 / 10000) = 23456790
base model = floor(117283950 * 1000 / 10000) = 11728395
contributor = floor(117283950 * 500 / 10000) = floor(5864197.5) =
5864197
leftover   = 117283950 - (76234567 + 23456790 + 11728395 + 5864197) = 1
contributor = 5864197 + 1 = 5864198
```

The buyback vault receives 6.172839 USDC, the owner 76.234567, the lane operator's `payout_ata` 23.456790, the base-model owner 11.728395 and the contributor 5.864198; together they are

exactly 123.456789 USDC. With all five accounts healthy, the lane afterwards has `total_allocated = total_distributed = 123,456,789`, `reserved_amount = 0` and an empty vault. Because the fee reached the buyback vault directly, `eligible_protocol_fees_paid` rises by 6,172,839.

9.6 Deferred payouts and Claim accounts

A share is paid directly when its destination is payable: an SPL Token account of the USDC mint, owned by the expected beneficiary, in the `Initialized` state. A frozen, closed or re-owned account is not payable. Its share is still allocated but deferred: it is added to a Claim account and stays in the vault as reserved revenue, while the other recipients are paid in the same transaction.

```
Claim (122 bytes)  PDA ["claim", lane, recipient_ata]
discriminator      8
version, bump      u8, u8
lane               Pubkey  lane the entitlement belongs to
recipient_ata      Pubkey  the destination that could not be paid
beneficiary        Pubkey  the wallet entitled to claim
amount             u64     unpaid entitlement
eligible_protocol_fee_amount  u64     protocol-fee part that earns dispute credit when
paid
```

`prepare_claim(recipient_ata, recipient_owner)` creates a Claim. Anyone may call it and pay the rent, but only for a recipient in the lane's resolved split or for the buyback vault with the protocol PDA as owner. Only a positive share that cannot be paid directly needs one; healthy recipients and zero shares use the placeholder and cost no rent. An existing Claim is reused, and its lane, recipient and beneficiary never change. A fee the buyback vault cannot receive, while that account is frozen or closed, accrues to the vault's Claim with the protocol PDA as beneficiary.

`claim_deferred` pays a Claim out. The beneficiary signs; there is no other authority over the money.

1. Check the destination against the configured destination policy. Under `CanonicalAta`, the default, it must be the beneficiary's canonical USDC associated token account. Under `AnyOwnedUsdc`, which an admin must select explicitly, it may be any USDC account the beneficiary owns.
2. If the Claim holds nothing, return.
3. Require `lane.reserved_amount >= claim.amount`.
4. Transfer `claim.amount` from the vault, signed by the lane PDA.
5. Add `claim.eligible_protocol_fee_amount` to `lane.eligible_protocol_fees_paid`, zero both Claim amounts, subtract the amount from `reserved_amount` and add it to `total_distributed`.
6. Emit `DeferredClaimPaid`.

The configured claim policy is `BeneficiaryOnlyNoExpiry`: only the beneficiary can claim, and an entitlement never expires. `claim_deferred` does not check the pause flag, so beneficiaries can claim while the protocol is paused.

A deferred protocol fee has the protocol PDA as beneficiary, which has no private key, so it is paid by `claim_protocol_fee` instead. Anyone may submit it, and its only destination is the buyback vault, held to the protocol PDA's canonical USDC ATA and to the protocol PDA as token authority. It requires the claim policy and `lane.reserved_amount >= claim.amount`, transfers the amount from the lane vault signed by the lane PDA, adds `claim.eligible_protocol_fee_amount` to `lane.eligible_protocol_fees_paid`, zeroes both Claim amounts, subtracts the amount from `reserved_amount`, adds it to `total_distributed` and emits `DeferredClaimPaid` with the protocol PDA as beneficiary. It runs while paused as well. Before each buyback the keeper submits it for lanes holding an outstanding protocol-fee Claim, up to four per pass.

9.7 Worked example: eight recipients, one deferred

A lane uses an eight-entry split with 1,250 bps per entry: the owner, the operator, a locked base-model royalty and five contributors. The vault holds 40,000,005 base units, nothing is reserved and the fee is 500 bps. The fifth entry's USDC account has been frozen by the token issuer.

```
A      = 40000005
fee = floor(40000005 * 500 / 10000) = floor(2000000.25) = 2000000
net = 40000005 - 2000000 = 38000005
share_1..7 = floor(38000005 * 1250 / 10000) = floor(4750000.625) =
4750000
leftover    = 38000005 - 8 * 4750000 = 5
share_8     = 4750000 + 5 = 4750005
```

Entries	Allocated each	Paid	Deferred
Buyback vault (fee)	2,000,000	2,000,000	0
1-4 and 6-7: owner, operator, base model, three contributors	4,750,000	28,500,000	0
5: contributor with frozen account	4,750,000	0	4,750,000
8: last contributor, with the leftover	4,750,005	4,750,005	0
Total		35,250,005	4,750,000

Afterwards `total_allocated = 40,000,005`, `total_distributed = 35,250,005`, `reserved_amount = 4,750,000` and the vault holds 4,750,000, so `total_allocated = total_distributed + reserved_amount` holds. When calls add 1,000,000 base units, the vault holds 5,750,000 but `available` is 1,000,000, and the next `distribute` allocates only that million. Once the contributor's account is usable again, the contributor signs `claim_deferred`: the vault pays 4,750,000, `reserved_amount` falls by that amount and `total_distributed` rises by it.

A distribution to the buyback vault and eight split recipients is a 308-byte v0 transaction with an address lookup table and a compute-budget instruction, inside a single 1,232-byte packet.

9.8 Split revisions and per-lane activation

An owner changes a split by proposal, and the change waits seven days.

1. **Propose.** The owner signs `propose_split(entries)` while the protocol is unpaused. The entries must pass the rules in 9.2, no other proposal may be pending, and every locked entry must be preserved or its beneficiary must co-sign. The program stores `pending_split = { revision: split_revision + 1, proposed_at: now, effective_at: now + 604800, entries }` and emits `SplitProposed`.
2. **Cancel.** The owner may call `cancel_split` at any time before application, including while the protocol is paused.
3. **Apply.** Once the chain clock reaches `effective_at`, anyone may call `apply_split` while the protocol is unpaused. The model's `split` and `split_revision` take the pending values and `SplitApplied` is emitted.

Applying a split changes the model's default; it does not reprice money already in a lane. Each lane keeps the split under which its balance arrived and switches only at a balance boundary. After the allocation in `distribute`, and on the zero-available path, the program compares

`lane.active_split_revision` with `model.split_revision`. If they differ, it requires `vault.amount = reserved_amount`, meaning every unreserved base unit has just been allocated under the old split, and only then copies the model's split and revision into the lane and emits `LaneSplitActivated`.

`distribute` write-locks the lane and its vault, so a payment lands wholly before the boundary and is paid under the old split, or wholly after it and is paid under the new one. Deferred Claims keep their original beneficiaries. An idle lane activates the new split through a zero-amount `distribute` that moves no tokens.

Price and policy follow a different path. `update_price` and `update_policy` are signed by the owner alone, take effect in the transaction that carries them, and leave the split, its revision and every lane balance unchanged; new sessions are quoted at the new price, and eligibility applies the new policy from the next chain read.

Example timeline for a model at split revision 1 with two lanes:

Time	Action	Model revision	Lane A	Lane B
Day 0	Owner proposes revision 2	1 (pending 2)	rev 1, revenue accrues	rev 1, empty
Day 7	<code>apply_split</code>	2	rev 1, revenue available	rev 1, empty
Day 7	<code>distribute(A)</code>	2	Allocates the revenue under rev 1, then activates rev 2	rev 1
Day 7	<code>distribute(B)</code>	2	rev 2	Nothing available; activates rev 2
After	Calls continue	2	Paid under rev 2	Paid under rev 2

The registry exposes each lane's active revision next to the model's revision, so a lane still behind is visible.

9.9 The activation keeper

Sealroom runs a keeper that applies due split proposals and moves every lane to the new revision, so owners and recipients need not submit these transactions themselves. Every activation transaction it sends is permissionless; for activation the keeper has no authority the program does not give everyone.

1. **Discovery.** On each pass it enumerates all `Model` and `Lane` accounts from chain. A model with a pending split gets a model job due at `effective_at`; a lane whose `active_split_revision` is below its model's `split_revision` gets a lane job.
2. **Model job.** When the chain clock has reached `effective_at` and the protocol is unpaused, the keeper submits `apply_split`.
3. **Lane job.** The keeper submits `prepare_claim` for any positive share that cannot be paid directly and has no Claim yet, then submits `distribute`. The job is complete when the lane's `active_split_revision` reaches the target on chain.
4. **Order and funding.** Owners take turns, one lane activation per owner at a time; within an owner, empty lanes go first, then lanes by descending available balance. A failing lane retries on its own schedule. Network fees and Claim rent come from the keeper's fee payer within global and per-owner lamport allowances over a rolling window.

Each lane job runs under a generation-fenced lease, and signed bytes are stored before broadcast, so two keeper workers never send a second financial operation for one boundary. The same keeper schedules attestation-expiry and unbond-maturity events, calls the facilitator's `/reconcile` (8.7), delivers deferred protocol fees with `claim_protocol_fee`, and, as `Config.buyback_authority`, starts `buyback_swap_and_burn` (14.7). It never withdraws an operator's bond.

9.10 Fee configuration

`Config.protocol_fee_bps` changes only through a timelocked proposal. The admin signs `propose_fee(protocol_fee_bps)`. The program requires `protocol_fee_bps <= 10000`, refuses a second pending proposal and sets `effective_at = now + 172800` (48 hours). The admin may cancel with `cancel_fee`. After the delay anyone may call `apply_fee`, which installs the rate and emits `FeeApplied`. Every `distribute` after that point uses the new rate on the lane's available balance. The launch parameter is 500 bps.

The fee's destination is not part of any proposal. `propose_fee` still takes a canonical USDC token account, and `apply_fee` records it in `Config.fee_treasury_ata` and `fee_treasury_owner`, but `distribute` reads neither field: it pays every fee into the buyback vault, whose address follows from the program id and the USDC mint.

10. Bonds, disputes and arbitration

An operator is a vetted company that has signed the Operator Agreement and posted a USDC bond. The bond backs the agreement: if an operator leaks weights or attests fraudulently, the arbitration panel can freeze it, invalidate all of its attestations and pay part or all of its bond to the harmed owner. This chapter describes the bond, the dispute process that leads to a ruling, and the arithmetic of slashing.

Bonds, deposits and slashes are denominated in USDC. No token, \$SEALROOM included, is ever accepted as bond collateral (14.7).

10.1 Bond accounts and tiers

Each operator has an `Operator` account at `["op", authority]` (193 bytes). Its bond fields are:

Field	Type	Bytes	Meaning
<code>tier</code>	<code>u8</code>	1	Admission tier, 1 to 3
<code>bond_vault</code>	<code>Pubkey</code>	32	USDC associated token account owned by the <code>Operator</code> PDA
<code>bond_amount</code>	<code>u64</code>	8	Bond principal: deposited, not withdrawn, not slashed
<code>unbond_at</code>	<code>Option<i64></code>	9	Time from which the principal may be withdrawn
<code>frozen</code>	<code>bool</code>	1	Set by the admin or by an arbitration ruling
<code>open_dispute_count</code>	<code>u32</code>	4	Ordinary disputes open against this operator
<code>next_dispute_sequence</code>	<code>u64</code>	8	Nonce of the next dispute record
<code>attestation_epoch</code>	<code>u64</code>	8	Revocation epoch that every attestation must match
<code>pending_slash_amount</code>	<code>u64</code>	8	Slashes ruled but not yet transferred out of the vault

`register_operator`, signed by the admin after off-chain vetting, creates the account and the bond vault with a principal of zero. `Config.tier_bond_minimums` holds exactly three entries, one minimum in USDC base units for each of tiers 1, 2 and 3, and `register_operator` and `set_tier` accept only a configured tier. An operator is healthy when it is not frozen and its principal meets its tier's minimum; `register_platform`, `register_node` and `record_attestation` all require a healthy operator. A model can demand more through its policy fields `min_tier` (1 to 3) and `min_bond`:

```

tier_minimum = Config.tier_bond_minimums[tier]
healthy = (frozen == false) and (bond_amount >= tier_minimum)
bond_eligible_for_model = bond_amount >= max(Model.min_bond,
tier_minimum)
tier_eligible_for_model = tier >= Model.min_tier

```

Only `bond_amount` counts. USDC sent to the bond vault outside `deposit_bond` does not raise the principal and cannot be withdrawn as principal.

10.2 Depositing, unbonding and withdrawing

`deposit_bond(amount)` . The operator authority signs. The program requires the protocol to be unpaused, `amount > 0` , the operator not frozen and no unbond request pending. It moves `amount` with `TransferChecked` from a USDC account owned by the authority into the bond vault, adds it to `bond_amount` and emits `BondDeposited` .

`request_unbond()` . The operator authority signs. The program requires `bond_amount > 0` , the operator not frozen, `open_dispute_count = 0` and no request already pending. It sets `unbond_at = now + 2592000` , which is 30 days, and emits `UnbondRequested` .

`cancel_unbond()` . The operator authority signs. The program clears `unbond_at` and emits `UnbondCancelled` . No funds move, and the operator may deposit again.

`withdraw_bond()` . The operator authority signs. The program requires the operator not frozen, `open_dispute_count = 0` , a pending request and `now >= unbond_at` , and that the vault holds at least `bond_amount` . It transfers the whole principal, signed by the `operator` PDA, to a USDC account owned by the authority, sets `bond_amount = 0` , clears `unbond_at` and emits `BondWithdrawn` . If a full slash has already reduced the principal to zero, the call just clears the matured request.

Unbonding also bounds the node's attestations. While `unbond_at` is set, `record_attestation` refuses any attestation whose `expires_at` falls after it (`AttestationBeyondUnbond`). No node of an unbonding operator can hold a valid attestation, and therefore cannot receive keys or serve paid calls, beyond the moment its bond can leave.

Operator state	<code>deposit_bond</code>	<code>request_unbond</code>	<code>withdraw_bond</code>
Healthy, no request	allowed	allowed if principal > 0	refused: no request
Unbond requested, before <code>unbond_at</code>	refused	refused: already requested	refused: timelock
Unbond requested, after <code>unbond_at</code>	refused	refused: already requested	allowed
Ordinary dispute open	allowed	refused	refused
Frozen	refused	refused	refused

The keeper tracks unbond maturity and surfaces it in the registry. It never withdraws a bond.

10.3 Dispute configuration

Disputes run under settings in `Config` that only a timelocked proposal can change:

Field	Meaning
<code>arbitration</code>	The arbitration multisig, the only key that may rule
<code>dispute_deposit</code>	Deposit in base units that a claimant posts to open a dispute
<code>dispute_deposit_policy</code>	<code>ForfeitOnDismissal</code>
<code>max_open_per_claimant_operator</code>	Open ordinary disputes allowed per (claimant, operator)
<code>max_open_per_operator</code>	Open ordinary disputes allowed per operator across all claimants; 8 at initialization
<code>dispute_forfeiture_ata</code> , <code>dispute_forfeiture_owner</code>	Where a dismissed claimant's deposit goes
<code>pending_dispute_config</code>	A proposal waiting for its delay

The admin signs `propose_dispute_config(arbitration, dispute_deposit, max_open_per_claimant_operator, max_open_per_operator, deposit_policy)` with the forfeiture account. The program requires an arbitration key distinct from the admin and from the default key, a positive deposit and positive caps, the `ForfeitOnDismissal` policy, a forfeiture account that is a canonical USDC associated token account not owned by the arbitration key, and no other pending proposal.

The proposal takes effect after `now + 172800` (48 hours), when anyone calls `apply_dispute_config`; the program checks the separation of keys again at that moment. The admin may cancel with `cancel_dispute_config`. The older direct setter `set_dispute_config` always fails with `GovernanceTimelockRequired`. The admin therefore cannot appoint itself arbitrator, and a new arbitrator cannot be installed and rule in one step: a proposed arbitrator has no authority until its proposal is applied.

`open_dispute` refuses to run until the arbitrator, the deposit, the policy, the claimant cap and the forfeiture recipient are all set.

10.4 Who may open a dispute

An ordinary dispute is opened by a model owner against an operator that has served its model under paid traffic. In program terms, the claimant owns a `Model` (the model PDA is derived from the claimant's key), a `Lane` exists for that model on a node of the target operator, and that lane holds unused admission credit of at least the configured deposit.

Admission credit measures protocol fees that left this lane and reached the buyback vault. The program counts them in `eligible_protocol_fees_paid`:

- In `distribute`, when the fee is transferred directly into the buyback vault, the fee is added. The program confirms that the vault's owner, the protocol PDA, is neither the model owner nor the authority of the lane's operator.
- When the buyback vault cannot receive the fee, the fee is deferred into the vault's Claim, recorded in `Claim.eligible_protocol_fee_amount`, and added to the lane only when `claim_protocol_fee` actually delivers it into the vault.
- Gross payments into the vault, the owner's and operator's shares, other split shares and fees still reserved are never counted.

```
protocol = PDA("protocol")
eligible = (protocol != model.owner) and (protocol !=
operator.authority)
paid into the buyback vault and eligible:
lane.eligible_protocol_fees_paid += fee
deferred into its Claim and eligible:
claim.eligible_protocol_fee_amount += fee    (moved to the lane by
claim_protocol_fee)
credit = lane.eligible_protocol_fees_paid -
lane.admission_credit_consumed
admitted = credit >= D
on admission: lane.admission_credit_consumed += D
```

Here `D` is `Config.dispute_deposit`. Admission consumes exactly `D` of credit, and the credit does not come back when the dispute is dismissed or upheld, or when governance later changes the deposit. A second dispute needs another `D` of fresh credit and another deposit.

Under the launch fee of 500 bps, credit accrues at one twentieth of the revenue distributed through the lane:

```
fee_k = floor(A_k * 500 / 10000)
admitted when fee_1 + fee_2 + ... >= D
single distribution: floor(A / 20) >= D  <=>  A >= 20 * D
```

For every 1 USDC of configured deposit, the lane must have distributed at least 20 USDC of revenue, with its fee delivered into the buyback vault, before its owner can open an ordinary dispute against the operator. Opening it then costs the deposit on top. Credit belongs to one lane: fees paid through the claimant's lanes on other operators admit nothing against this one.

10.5 Opening a dispute

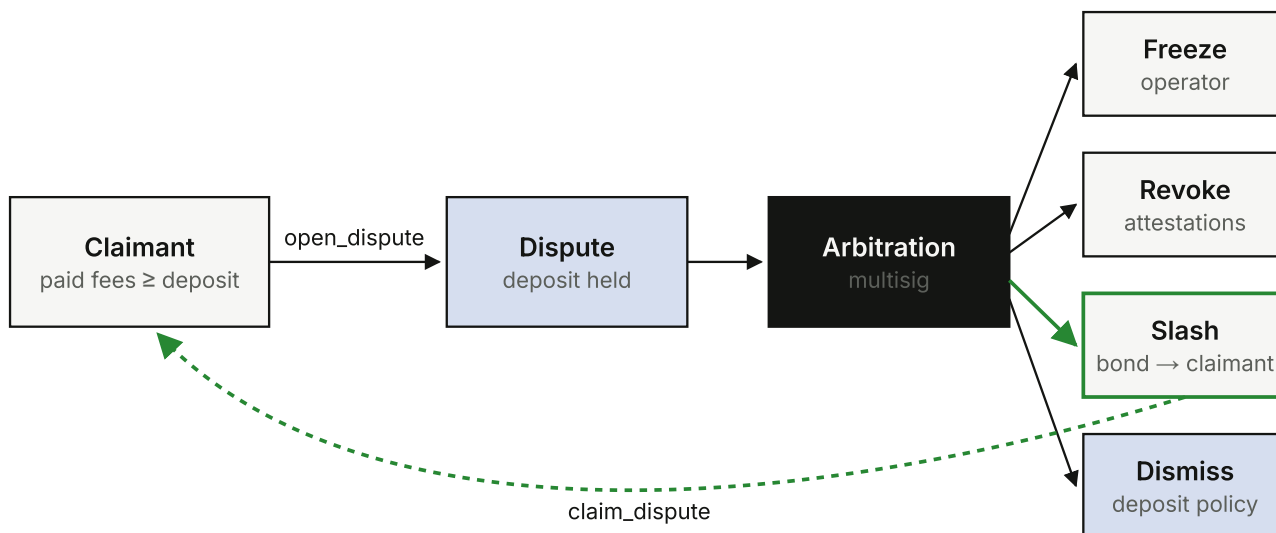


Figure: a dispute from admission through ruling to slash settlement and the claimant's or forfeiture beneficiary's claim.

`open_dispute(sequence, evidence_hash)` takes the claimant's signature and these accounts: `config`, the target `operator` (writable), the claimant's `model`, a `node` of that operator, the `lane` for `model` and `node` (writable), the `claimant_disputes` counter `["claimant_disputes", claimant, operator]`, the new `dispute` at `["dispute", operator, sequence:u64]`, the USDC mint, the claimant's USDC `source` account and the `deposit_vault`, the USDC associated token account of the dispute PDA. The `evidence_hash` is the 32-byte hash of the evidence the claimant submits to the arbitration panel off chain.

The instruction runs these steps:

1. Require a configured arbitrator, a positive deposit `D` and the `ForfeitOnDismissal` policy.
2. Require `sequence = operator.next_dispute_sequence`; a stale or replayed nonce fails.
3. Compute the lane's credit and require `credit >= D` (`InsufficientPaidUsage`). Add `D` to `admission_credit_consumed`.
4. Require `operator.open_dispute_count < max_open_per_operator` and `claimant_disputes.open_count < max_open_per_claimant_operator` (`DisputeLimitReached`), creating the claimant counter if needed. Increment the claimant counter.
5. Transfer `D` from the claimant to the deposit vault.
6. Increment `operator.next_dispute_sequence` and `operator.open_dispute_count`.
7. Write the `Dispute` account and emit `DisputeAdmission` (credit used and remaining) and `DisputeOpened`.

Any failure rolls back the whole instruction: credit, tokens and both counters. The `Dispute` account is 415 bytes:

Field	Type	Meaning
<code>operator, sequence</code>	<code>Pubkey, u64</code>	PDA seeds

Field	Type	Meaning
<code>model</code> , <code>lane</code> , <code>node</code> , <code>claimant</code>	<code>Pubkey</code>	The paid relationship and the claimant
<code>evidence_hash</code> , <code>ruling_hash</code>	<code>[u8; 32]</code> , <code>Option<[u8; 32]></code>	Commitments to the evidence and the published ruling
<code>deposit</code> , <code>deposit_vault</code>	<code>u64</code> , <code>Pubkey</code>	The posted deposit and the account holding it
<code>status</code>	<code>enum</code>	<code>Open</code> , <code>RuledForClaimant</code> or <code>Dismissed</code>
<code>slash_amount</code> , <code>slash_pending_amount</code>	<code>u64</code>	Slash ordered, and the part not yet moved from the bond vault
<code>claimable_amount</code> , <code>forfeited_amount</code>	<code>u64</code>	Owed to the claimant, or to the forfeiture beneficiary
<code>forfeiture_ata</code> , <code>forfeiture_beneficiary</code> , <code>deposit_policy</code>	<code>Pubkey</code> , <code>Pubkey</code> , <code>enum</code>	Forfeiture terms copied at opening
<code>opened_at</code> , <code>ruled_at</code>	<code>i64</code> , <code>Option<i64></code>	Timestamps
<code>arbitration_initiated</code> , <code>admission_credit_used</code> , <code>admission_required_amount</code>	<code>bool</code> , <code>u64</code> , <code>u64</code>	Origin of the record, credit consumed and threshold applied

Because the forfeiture terms are copied when the dispute opens, a later governance change cannot redirect a deposit already at stake.

10.6 Open-dispute caps

Two counters bound the number of ordinary disputes open at the same time:

- `ClaimantDisputes.open_count` (78-byte account at `["claimant_disputes", claimant, operator]`) counts one claimant's open disputes against one operator, across all of that claimant's models. It may not exceed `max_open_per_claimant_operator`.
- `Operator.open_dispute_count` counts open disputes against the operator from all claimants. It may not exceed `max_open_per_operator`, which is 8 at initialization and changes only through the 48-hour proposal.

Opening increments both counters together, and a ruling decrements both exactly once. Since every admission also spends paid-fee credit and a forfeitable deposit, fresh wallet keys without a paid lane cannot fill the operator-wide queue. Arbitration actions (10.8) use neither counter, so the panel can act on an operator while its queue is full.

10.7 Ruling a dispute

`rule_dispute(for_claimant, slash_amount, freeze, revoke_attestations, ruling_hash)` is signed by the current arbitration key, which must equal `Config.arbitration`. Its accounts are the operator (writable), the dispute, the claimant counter, the USDC mint, the bond vault and the deposit vault.

1. Require `dispute.status = Open` (`DisputeAlreadyRuled` otherwise); each dispute is ruled once.
2. Require `slash_amount <= operator.bond_amount`.
3. A dismissal (`for_claimant = false`) must have `slash_amount = 0`, `freeze = false` and `revoke_attestations = false`.
4. Reduce `bond_amount` by the slash and add it to `pending_slash_amount`.
5. Decrement `open_dispute_count` and the claimant counter.
6. If `freeze`, set `operator.frozen = true`. The operator can no longer register platforms or nodes, record attestations, deposit, unbond or withdraw, and every model's eligibility check fails with `operator_frozen`.
7. If `revoke_attestations`, increment `operator.attestation_epoch`. Every attestation of every node of the operator committed the old epoch in its signed message, so all of them fail the epoch check immediately, and new ones must be signed against the new epoch.
8. Record the outcome: status `RuledForClaimant` Or `Dismissed`, `ruling_hash`, `slash_amount`, `ruled_at`, `slash_pending_amount = slash_amount`, and either `claimable_amount = deposit` (upheld) or `forfeited_amount = deposit` (dismissed).
9. Emit `DisputeRuled` with the ruling authority, the slash, the deposit disposition, the frozen flag, the new attestation epoch and the ruling hash.

The ruling performs no token transfer, so freezing and revocation take effect even if the token issuer has frozen the bond vault or the deposit vault; moving the slashed USDC is a separate step (10.9). An upheld claimant gets the deposit back together with the slash; a dismissed claimant's deposit goes to the forfeiture beneficiary recorded in the dispute.

10.8 Arbitration actions without a slot

The panel can also act on an operator directly with `arbitrate_operator(sequence, evidence_hash, slash_amount, freeze, revoke_attestations, ruling_hash)`, signed by the current arbitration key. This path exists for security responses that cannot wait for, or do not come from, an ordinary dispute.

- A **zero-slash action** freezes the operator and/or advances its attestation epoch. It needs no model, node or lane, no deposit and no forfeiture configuration; the three relationship accounts are omitted together.
- A **positive slash** needs the complete relationship: a `Model`, a `Node` of this operator and the `Lane` for them, all at their canonical addresses, with the model owned by the named claimant. That lane's lifetime `eligible_protocol_fees_paid` must be at least the configured deposit. Lifetime

fees count here, including credit already consumed by ordinary disputes, because the panel itself is the admitting authority.

```
required = 0                                if slash_amount = 0
required = D                                if slash_amount > 0, requiring D > 0
and lane.eligible_protocol_fees_paid >= D
```

The action checks the dispute nonce and `slash_amount <= bond_amount`, advances `next_dispute_sequence`, moves the slash from `bond_amount` to `pending_slash_amount`, applies freeze and revocation, and creates a `Dispute` record that is already ruled: status `RuledForClaimant`, deposit 0, `arbitration_initiated = true`, `admission_credit_used = 0`, `admission_required_amount = required`. It creates the claimant counter if needed but does not increment it, does not touch `open_dispute_count` and consumes no admission credit. It emits `ArbitrationAction`. The slash then follows the same settlement and claim path as an upheld dispute.

10.9 Slashing arithmetic and claims

A slash first moves from principal to a pending liability and later moves as tokens from the bond vault to the dispute's deposit vault, from which the claimant withdraws.

```
at ruling:      bond_amount' = bond_amount - s,          s <= bond_amount
                pending_slash_amount' = pending_slash_amount + s
                dispute.slash_pending_amount = s
                dispute.claimable_amount = D (upheld) or 0 (dismissed or
arbitration action)
at settle:      bond_vault -= s,  deposit_vault += s
                dispute.claimable_amount += s,
dispute.slash_pending_amount = 0
                pending_slash_amount' = pending_slash_amount - s
claimant total: D + s (upheld dispute)   or   s (arbitration action)
```

`settle_slash` is permissionless: any signer may submit it, and the program checks only the account links and the dispute state. It requires a ruled dispute, transfers `slash_pending_amount` from the bond vault to the deposit vault signed by the `Operator` PDA, and updates the three counters above. A dispute with nothing pending returns without effect, so the transfer happens once. It emits `SlashSettled`.

`claim_dispute` is signed by the claimant. It requires a ruled dispute, checks the destination against the claim destination policy (the claimant's canonical USDC account by default), transfers `claimable_amount` from the deposit vault signed by the dispute PDA, sets it to zero and emits `DisputeClaimPaid`. A claimant may claim the returned deposit before the slash is settled and claim again afterwards; each call pays only what is claimable at that moment.

`claim_forfeited_deposit` is signed by the forfeiture beneficiary recorded in the dispute. It requires status `Dismissed`, applies the same destination policy, transfers `forfeited_amount` and sets it to zero. It emits `DisputeDepositForfeited`.

Walk-through of an upheld dispute with bond principal `B`, deposit `D` and slash `s ≤ B`:

Step	<code>bond_amount</code>	<code>pending_slash_amount</code>	Bond vault	Deposit vault	<code>claimable_amount</code>
Before opening	<code>B</code>	<code>0</code>	<code>B</code>	<code>0</code>	<code>0</code>
<code>open_dispute</code>	<code>B</code>	<code>0</code>	<code>B</code>	<code>D</code>	<code>0</code>
<code>rule_dispute</code> (upheld, slash <code>s</code>)	<code>B - s</code>	<code>s</code>	<code>B</code>	<code>D</code>	<code>D</code>
<code>settle_slash</code>	<code>B - s</code>	<code>0</code>	<code>B - s</code>	<code>D + s</code>	<code>D + s</code>
<code>claim_dispute</code>	<code>B - s</code>	<code>0</code>	<code>B - s</code>	<code>0</code>	<code>0</code>

For a dismissal the ruling leaves `bond_amount` at `B`, sets `forfeited_amount = D`, and `claim_forfeited_deposit` empties the deposit vault to the forfeiture beneficiary.

10.10 Pending slashes and bond withdrawals

The program keeps the bond's principal and the pending slashes apart, so a withdrawal can never take money a ruling has already assigned to a claimant.

```
bond_vault.amount >= bond_amount + pending_slash_amount
withdrawable = bond_amount
```

A ruling lowers `bond_amount` immediately and raises `pending_slash_amount`, but leaves the tokens in the vault. `withdraw_bond` transfers only `bond_amount`, so the pending slash stays in the vault for `settle_slash`, whichever of the two runs first. Take principal `B` and an arbitration action that slashes `s` without freezing. After the action, `bond_amount = B - s` and `pending_slash_amount = s` while the vault still holds `B`. If the operator then unbonds and withdraws after 30 days, it receives `B - s` and the vault keeps `s`; `settle_slash` later moves `s` to the dispute's deposit vault and the claimant claims it.

A frozen operator cannot withdraw at all, and an operator with an ordinary dispute open can neither request unbonding nor withdraw. Each ruling decrements `open_dispute_count`; once no ordinary dispute is open and the operator is not frozen, the unbonding path in 10.2 is available again for the remaining principal.

11. Services and infrastructure

The `sealroom` program holds the registry and the money. The work around it runs as off-chain services: projecting chain state into a database, verifying vendor evidence, releasing data keys, serving inference, sponsoring payment transactions and starting time-based transitions that a Solana program cannot start by itself. Each service either reads the chain or submits transactions that the program checks on its own terms. No service holds a user's private key; every wallet action is signed in the user's wallet.

11.1 Service map

Service	Run by	Interface	Solana role	Secrets held
Web	Sealroom	Site, registry app, intake form	None; wallets sign	None
Public API	Sealroom	HTTPS JSON under <code>/v1</code>	Reads RPC to build previews	Admin bearer credential, rate-limit HMAC key, challenge secret
Indexer	Sealroom	None; writes Postgres	Reads accounts and history	None
Keeper	Sealroom	None	Fee payer for split activation and protocol-fee claims; <code>Config.buyback_authority</code> for <code>buyback_swap_and_burn</code>	One key, fee payer and buyback authority
Coordinator	Sealroom	<code>POST /v1/attest</code>	Fee payer for <code>record_attestation</code>	Fee-payer key, verifier endpoint credentials
Verifier (five)	Committee members; Sealroom runs at most two	<code>POST /v1/sign</code>	Signs the committee message	One Ed25519 verifier key
Facilitator	Sealroom, or any SVM x402 facilitator	<code>/requirements</code> , <code>/verify</code> , <code>/settle</code> , <code>/settle-voucher</code> , <code>/reconcile</code>	Fee payer of payment transactions	Sponsor key, payment-channel authority
Key broker	Model owner	<code>POST /v1/key-requests</code>	Reads accounts	Model data keys
Node runtime	Operator, inside the CVM	<code>POST /v1/chat/completions</code> over attested TLS	Reads accounts	Enclave HPKE, TLS and voucher keys

The API, coordinator, verifiers, facilitator, broker and node expose `GET /health`; the indexer and keeper have no listener. The coordinator, verifiers and broker are infrastructure endpoints: they reject requests carrying a browser `Origin` header and require a service bearer credential compared in constant time. Browsers reach Sealroom through the public API and node endpoints only.

11.2 Public API

The public API is the read model and transaction builder for the web app, SDKs and agents. Registry reads come from the Postgres projection kept by the indexer; transaction previews read the Solana RPC directly.

11.2.1 Envelope and encoding

Every response uses one envelope, `{ ok: true, data, meta }` or `{ ok: false, error, meta }`. `meta` carries the contract version (`1.0.0`), a request ID (also sent as `X-Request-Id`), `asOf` (server wall time) and `index`, the freshness record of 11.3.4. Values that are `u64` on chain are exact decimal strings; USDC amounts are base units with 6 decimals. Keys and signatures are base58, hashes lowercase hex, runtime measurements 48 bytes. An error carries a code (`VALIDATION_ERROR`, `NOT_FOUND`, `UNAUTHORIZED`, `FORBIDDEN`, `RATE_LIMITED`, `IDEMPOTENCY_CONFLICT`, `ABUSE_REJECTED`, `CONFIGURATION_UNSET`, `INTERNAL_ERROR` and others), a `retryable` flag and, where relevant, per-field messages or `retryAfterSeconds`. Responses are `no-store`, cross-origin requests are accepted from exact configured origins only, and request bodies are JSON of at most 16 KiB.

11.2.2 Routes and resources

Method	Path	Purpose
GET	<code>/v1/parameters</code>	Protocol parameters read from <code>Config</code> (chapter 13), with the buyback mint, authority, vault, vault balance, USDC spent and amount burned
GET	<code>/v1/summary</code>	Operators, attested nodes, models, open lanes, open disputes, total distributed, total burned
GET	<code>/v1/{resource}</code> , <code>/v1/{resource}/{id}</code>	Filtered list; one record
GET	<code>/v1/eligibility?</code> <code>modelId=&nodeId=</code>	Serving decision for one model on one node
POST	<code>/v1/transactions/preview</code>	Transaction for a wallet intent
POST	<code>/v1/applications</code>	Operator or model-owner application
Resource	Filters	
<code>operators</code>	<code>status</code> (bonded, unbonding, frozen, unbonded), <code>tier</code> , <code>operatorId</code>	
<code>platforms</code>	<code>operatorId</code>	

Resource	Filters
nodes	status (registered, serving, offline, disabled), operatorId, platformId, nodeId
attestations	status (valid, expiring_soon, expired, revoked), operatorId, platformId, nodeId, method
runtimes	status (pending, allowed, revoked)
models	owner, modelId
lanes	status, modelId, nodeId, laneId, operatorId
distributions	laneId, modelId, nodeId
disputes	status (open, ruled_for_claimant, dismissed), operatorId, modelId, claimant
key-releases	operatorId, nodeId, modelId, outcome (released, denied)
activity	operatorId, nodeId, modelId, kind

Lists take `limit` from 1 to 100 (default 20), a text query of at most 128 characters and an opaque cursor bound to a digest of the resource and filters, so a cursor from one query cannot page another. Unknown filters and mismatched cursors return 422. An attestation's identity is `(account, revision)`, and its detail returns the latest committee-signed revision.

`/v1/eligibility` runs the same rule set that the key broker and the node apply before releasing a key or serving a call, returning each check with its actual and required value. The twelve checks are `no_valid_attestation`, `attestation_expired`, `attestation_revoked` (including an outdated operator epoch), `runtime_not_allowed` (the measurement must be allowed globally and pinned by the model), `platform_not_registered`, `tier_below_min`, `bond_below_min` (the larger of the model's `min_bond` and the tier minimum), `operator_frozen`, `operator_not_on_allowlist`, `tcb_not_up_to_date`, `node_disabled` and `protocol_paused`. Evaluation time is the chain Clock observed by the indexer plus the wall time elapsed since that observation, so expiry follows chain time.

11.2.3 Transaction previews

`POST /v1/transactions/preview` turns a wallet intent into a transaction the user's wallet signs. Nineteen intents are supported: `create_model`, `update_price`, `update_policy`, `open_lane`, `propose_split`, `apply_split`, `cancel_split`, `distribute`, `prepare_claim`, `claim`, `deposit_bond`, `request_unbond`, `cancel_unbond`, `withdraw_bond`, `register_platform`, `register_node`, `open_dispute`, `claim_dispute` and `claim_forfeited_deposit`. The response holds the base64 `serializedTransaction` (unsigned, or carrying co-signatures already collected), `requiredSigners`, an exact `estimatedFeeLamports` from the RPC, `submittable`, machine-readable `blockedBy` reasons, the instruction list and the resolved intent.

1. Request a preview and read `submittable` and `blockedBy`.
2. Deserialize the bytes without changing the message, keeping any co-signature present.
3. Sign with the wallet's `signTransaction`.
4. Send the raw bytes to the RPC and poll `getSignatureStatuses`.

The builder derives every PDA and canonical associated token account, adds idempotent token-account creation for fresh USDC destinations, picks the owner's first free positive model ID when `create_model` omits one, and refuses intents the program would reject, with codes such as `PROTOCOL_PAUSED`, `GPU_UEIDS_REQUIRED`, `OPERATOR_LOCK_UNSUPPORTED`, `CLAIM_PREPARATION_REQUIRED`, `ADMIN_SIGNATURE_REQUIRED` and `TRANSACTION_TOO_LARGE` (over 1,232 bytes). A `register_platform` preview names the admin multisig as the remaining co-signer. A `claim` intent for the buyback vault's Claim builds `claim_protocol_fee`, which any wallet may submit.

11.2.4 Applications intake with a challenge

Operators and model owners apply through `POST /v1/applications`. Every application carries organisation, contact, explicit consent, an empty honeypot field and a challenge response; operators add regions, GPU fleet, certifications, confidential-compute readiness and an optional onboarding wallet, and model owners add their domain, buyers and weight-protection need.

1. Strict validation; bad or unknown fields return 422 with a message per field.
2. Rate limiting through an atomic Postgres bucket keyed by an HMAC-SHA256 of the socket address (forwarded headers are not trusted); the default is 20 public writes per minute, shared with previews.
3. An `Idempotency-Key` of 16 to 128 characters is required. An identical resubmission returns the stored receipt before any challenge is consumed; changed content returns 409.
4. A filled honeypot returns 403 `ABUSE_REJECTED`.
5. The server verifies the Cloudflare Turnstile response with Siteverify and accepts it only when the returned hostname and action match configuration. Missing configuration fails closed with 503.
6. The application is stored and a receipt with one support ID is returned.

Contact data and challenge responses never enter logs. Admission on chain is a separate act: after vetting, the admin multisig signs `register_operator` with the hash of the signed Operator Agreement and the operator's tier.

11.3 Indexer

The indexer mirrors the program into Postgres and never signs.

11.3.1 A reconciliation pass

A pass runs every 2,000 ms by default at `finalized` commitment:

1. Read, in parallel, the genesis hash, all program accounts with their context slot, and the Clock sysvar. A database belongs to one genesis hash and program ID; a mismatch stops the pass.
2. Decode each account by its 8-byte discriminator; program-owned accounts matching no type are kept as raw audit records.
3. Advance signature history for the program, every lane vault and every bond vault.
4. Decode events and lane payments from new transactions.

5. Project accounts and events, reconcile money, and commit under a checkpoint compare-and-swap.

11.3.2 Signatures, events and payments

Each address has its own durable cursor. `getSignaturesForAddress` pages of up to 1,000 signatures walk back from a head frozen at the snapshot slot to the last processed signature; a pass processes at most 1,000 transactions per address within a backfill bound of 10,000, and commits per batch, so a large backlog is worked off over successive polls. Before declaring a scan complete the indexer fetches the anchor transaction it is scanning toward; a missing anchor marks that history unavailable.

Program events are authenticated self-CPI instructions (`emit_cpi!`) signed by the program's event-authority PDA. The indexer decodes them from inner instructions (tag `e445a52e51cb9a1d`), which keeps them intact past Solana's 10 KB log limit. Lane payments are SPL Token `Transfer` (3) and `TransferChecked` (12) instructions, top-level or inner, whose destination is a lane vault. Vault balances come from each processed transaction's post-token balances, so a payment and the balance it produced are read at the same boundary.

11.3.3 Reconciliation and commit

```
payments_in = total_allocated + vault_balance - reserved_amount
sum(LaneDistributed.amount) = total_allocated
sum(fee_paid + payouts_paid) + sum(DeferredClaimPaid.amount) =
total_distributed
```

A lane that fails a check or whose history is unavailable keeps its last good values, is marked `degraded` , and retries alone while the rest of the registry continues. A bond vault must cover `bond_amount + pending_slash_amount` ; USDC sent outside `deposit_bond` never counts as principal. Attestations are projected per signed revision: a revocation marks the revision current at that time, and an epoch-advancing ruling marks every earlier-epoch revision revoked. When the private evidence archive holds evidence whose key transcript hashes to the on-chain `enclave_key_hash` , the projection adds the HPKE key hash, TLS key hash and voucher public key.

Commits take one Postgres advisory lock and compare the checkpoint the batch started from (indexed slot, last signature and a strictly increasing timestamp). A batch whose base moved returns `committed: false` , so an older snapshot never overwrites a newer one. Deterministic record keys (`chain:<signature>:<event index>` , `payment:<signature>:<instruction>:<inner index>`) make restarts and repeated passes idempotent.

11.3.4 Freshness and index states

`meta.index` reports `indexedSlot` , `chainSlot` , `lagSlots` , `indexedAt` (wall time at which the observation started), `chainTime` , `backfillPending` and `status` :

```
stale = no_checkpoint or backfillPending or (now - indexedAt > 30000 ms)
or (lagSlots > 150)
```

Age counts from the start of the observation, so a slow reconciliation cannot make old data look fresh.

State	Shown in	Cause
Fresh	<code>meta.index.status</code>	Observation under 30 s old, lag within 150 slots, history complete
Stale by lag	<code>lagSlots</code>	Indexed slot more than 150 slots behind the chain
Stale by age	<code>indexedAt</code>	Last observation started more than 30 s ago
Stale by backfill	<code>backfillPending</code>	A cursor is mid-scan or a money mirror is degraded
Lane degraded	<code>lanes[].reconciliation</code>	That lane failed conservation; it retries alone
Bond degraded	<code>operators[].bondReconciliation</code>	Vault does not cover principal plus pending slash

11.4 Keeper

The keeper submits time-driven transitions that any wallet may submit and none is obliged to, and it starts the buyback. Apart from `buyback_swap_and_burn`, which it signs as `Config.buyback_authority`, it signs only permissionless instructions; it never withdraws a bond, never signs for a user and never holds protocol USDC. Every 15,000 ms in production (`KEEPER_POLL_MS`; 5,000 ms locally) it reconciles its pending transactions, runs one indexer pass, asks the facilitator to `/reconcile` expired `upto` ceilings, discovers jobs, leases up to 32 of them, and runs the buyback crank when it is due (11.4.5).

11.4.1 Jobs

Kind	Dedupe key	Action
Attestation expiry	<code>expiry:<attestation>:<revision></code>	Marks expiry; the program and eligibility read <code>expires_at</code> directly
Unbond maturity	<code>unbond:<operator>:<unbond_at></code>	Marks maturity; withdrawal stays the operator's signed <code>withdraw_bond</code>
Split activation, model	<code>model-split:<model>:<revision>:<effective_at></code>	<code>apply_split</code> once mature
Split activation, lane	<code>lane-split:<lane>:<revision></code>	<code>prepare_claim</code> as needed, then a boundary <code>distribute</code>

Duplicate inserts are skipped, so several keepers discover the same work once. Protocol-fee delivery and the buyback run outside this queue, in the buyback crank (11.4.5).

11.4.2 Split activation per model revision

Each lane stores its own `active_split` and `active_split_revision`; `apply_split` changes only the model default. A lane adopts a new revision only when its available balance is zero, so revenue that arrived under the old split is paid under the old split.

1. **Model job.** Once the chain Clock passes `effective_at`, re-read the model; if its revision is still the target minus one with the same `effective_at`, submit `apply_split`.
2. **Lane jobs.** Create one job per `Lane` account whose revision is below the model's, enumerated from chain accounts so that a lane missing from the index is still found.
3. **Claims.** For each recipient whose positive old-split share cannot be paid directly (closed, frozen or owner-changed account), submit `prepare_claim` for its Claim PDA.
4. **Pre-send check.** The model revision is unchanged, no Claim is missing, and the available balance is still zero or still positive.
5. **Boundary crank.** `distribute` allocates every unreserved base unit under the old split, reloads the vault and, when `vault.amount == reserved_amount`, copies the model split into the lane. An idle lane advances with a zero-amount call that moves no money.
6. **Completion.** The lane account's `active_split_revision` reaches the target; completion is read from the account, not from the transaction status.

Owners take turns by the time they were last served. Within an owner the model job goes first, then empty or reserved-only lanes, then lanes in descending available balance; at most one activation per owner is leased at a time, and a recipient failure affects only its own lane's retry.

11.4.3 SOL budget

Sponsorship requires a global cap, an owner cap and a rolling window, with no defaults. The owner cap must be below the global cap and large enough for one preparation transaction (up to three Claim rents plus its fee); an optional per-model lifetime cap adds a further brake. Every fee and every Claim rent counts against both caps:

```
max_cost = fee_lamports + claim_rent_lamports * new_claims
pending  = sum(max_cost of unsettled reservations)
used     = pending + sum(actual_cost of settlements within the window)
admit    = (pending + max_cost <= cap) and (used + max_cost <= cap)
```

Direct canonical payouts need no Claim, and Claims are bound to their beneficiary and reused. Actual cost is taken from the fee payer's balance change, with each Claim's rent attributed once. When pending reservations fill a cap, the lane retries in 60 s; when settled costs fill the window, it retries as they age out; a transaction larger than a cap is terminal. A receipt that costs more than its reservation is recorded `OVERDRAWN` and halts all sponsorship until a recorded review acknowledges the exact amount. Blocked budgets re-alert every 900 s. The keeper sponsors split activation, protocol-fee delivery and the buyback; the last two count against the global cap and against an

owner cap keyed to the protocol PDA. Ordinary distributions are paid by whichever wallet sends them.

11.4.4 Leases with generations

A lease records a worker, a generation incremented at each lease, and an expiry on the database clock: 60,000 ms by default, never more than 180,000 ms from acquisition. There is no heartbeat; only confirmed chain progress extends a lease. Every RPC call is bounded by 10,000 ms. The reservation checks owner and generation in the same database transaction; a broadcast requires at least the RPC timeout plus 1,000 ms of lease remaining; signed bytes and their cost breakdown are committed before broadcast, and an ambiguous send is retried with the same bytes and signature, so it cannot become a second operation. Completion and failure updates match the generation, so a superseded lease cannot send. Failed jobs retry after `min(300 s, 1 s * 2^min(attempts, 8))`.

11.4.5 Buyback crank

Every `BUYBACK_POLL_MS` (60,000 ms by default) one keeper, holding a Postgres advisory lock for the deployment, runs one buyback pass:

1. Read `Config` and the buyback vault. A paused protocol, or a buyback vault that is absent or frozen, ends the pass.
2. Submit `claim_protocol_fee` for lanes whose protocol-fee Claim holds a positive amount, up to four per pass, so deferred fees reach the vault. Each delivery has its own ledger key, `protocol-fee:<claim>:<total_allocated>`.
3. With no buyback mint selected, end the pass; fees accumulate in the vault. Otherwise require the keeper key to equal `Config.buyback_authority` and the vault balance to reach `BUYBACK_MIN_USDC`.
4. Request a Jupiter quote for the whole vault balance: `ExactIn` from USDC to the buyback mint at `BUYBACK_SLIPPAGE_BPS`. The quote must name exactly those mints, that amount, that mode and that slippage, carry no platform fee, and set a minimum output no lower than the quoted output reduced by the slippage; that minimum becomes `min_out`.
5. Request the swap instruction with the protocol PDA as user and its output account as destination. Each setup instruction Jupiter returns must name a canonical protocol ATA for USDC, the buyback mint or wrapped SOL that already exists, and none is sent; cleanup, token-ledger and other instructions, and any route signer other than the protocol PDA, are refused.
6. Build one v0 transaction with the deployment's and Jupiter's lookup tables, refuse it above 1,232 bytes or above the optional `BUYBACK_MAX_FEE_LAMPORTS`, simulate it, store the signed bytes in the spending ledger, confirm that `Config` is still unpaused with the same authority and spent total, broadcast and confirm. The ledger key is `buyback:<usdc_spent_total>`, so an ambiguous send is retried with the same bytes and completion is read from the advanced total on chain.

A pass that cannot complete records a `BUYBACK_*` code and retries at the next poll; the vault's USDC stays in the vault until a `buyback_swap_and_burn` succeeds.

11.5 Coordinator and verifier services

A node submits evidence in the `sealroom-evidence-v1` format (at most 1 MiB): node PDA, a recent slot and its hash, the 1,216-byte X-Wing public key, the 32-byte TLS public key, the 32-byte Ed25519 voucher key, the TDX quote and 1 to 8 GPU evidence blobs. The evidence hash is SHA-256 of a fixed-order serialization; raw evidence is archived privately by that hash.

Each committee member runs one verifier service with one Ed25519 key, its own credential and its own collateral cache. Quotes are checked with Intel DCAP collateral and the NVIDIA local verifier (RIMs and OCSP), with NVIDIA's NRAS as an optional cross-check; the vendor adapter is the only component that parses evidence. On `POST /v1/sign` a verifier:

1. Verifies the vendor evidence and extracts the platform ID hash, GPU UEIDs, runtime measurement, RIM hash, REPORTDATA, GPU nonces and TCB status.
2. Reads `Config`, `Node`, `Platform`, `Operator`, `Attestation`, `VerifierSet`, `Runtime`, the global pins, `Clock` and `SlotHashes` from RPC in one context, and checks the genesis hash.
3. Checks the registry: not paused, operator not frozen and at its tier's bond minimum, node enabled, platform and GPU pins matching this platform and node, runtime allowed and effective, slot hash present in `SlotHashes`.
4. Checks the binding: REPORTDATA and every GPU nonce equal `H` (12.4); the sequence is the next one, the operator epoch is current, the verifier is a member of the active epoch, and the expiry is within 24 hours.
5. Archives the evidence and signs the 200-byte canonical message.

The coordinator holds no verifier key, and its fee payer must not be a member. It repeats the vendor and registry checks, sets `expiresAt` to chain time plus 86,400 s minus 1 s, sends the proposal to all five verifiers in parallel (30 s timeout each), accepts only signatures from each endpoint's configured key over the exact message, keeps three distinct ones, and submits one Ed25519 precompile instruction followed by `record_attestation` as a v0 transaction with a lookup table. Measured on the compiled program, that transaction is 1,087 bytes and 46,465 to 46,505 compute units.

11.6 Facilitator

The facilitator settles x402 payments. The node never holds facilitator or channel custody keys.

Endpoint	Function
<code>/requirements</code>	Opens a session bound to request, caller, lane, scheme, amount and body hash; returns x402 requirements
<code>/verify</code>	Checks the caller's signed payment and its binding, with no transfer
<code>/settle</code>	Co-signs as fee payer, broadcasts, confirms, checks the vault's post-token balance
<code>/settle-voucher</code>	Settles <code>upto</code> usage from the enclave voucher and refunds the unused ceiling

Endpoint	Function
<code>/reconcile</code>	Returns expired, unserved ceilings and finishes ambiguous intents

Requirements name the USDC mint, the lane vault as `payTo`, the amount, a 120 s timeout and, in `extra`, the fee payer, a memo `sealroom:<sessionId>`, a recent blockhash and the session and lane IDs. The payment transaction must have exactly four instructions: compute-unit limit 100,000, compute-unit price 0, one `TransferChecked` of the exact amount with 6 decimals from the caller's canonical ATA to `payTo`, and the memo. It must be at most 1,232 bytes, use no lookup tables, require exactly the fee payer's and the caller's signatures, and keep the fee payer out of every instruction. Session state moves by compare-and-set, signed bytes are persisted before broadcast, an ambiguous retry looks up the same signature, and a settled request returns its receipt idempotently.

One `TransferChecked` to one lane vault per call keeps the layout that SVM x402 facilitators validate; the multi-recipient split happens later in `distribute`. For `upto`, the caller escrows a ceiling in the payment-channels program (`CHNLxYvVA28MJP9PrFuDXccuoGXA7jBacfLEkahyGsX`); after inference the facilitator checks the enclave-signed voucher against the session and the attested voucher key, settles the actual amount to the lane vault and returns the rest to the caller in one transaction.

11.7 Owner key broker

The `Model` account names its broker: `key_broker` (public key) and `key_broker_url` (HTTPS, at most 128 bytes). A node's key client sends `POST /v1/key-requests` with a request UUID, the model ID and its evidence. The broker reads the chain accounts directly in one RPC context, runs the eligibility rules, checks that the model names this broker and that the evidence hash, enclave key hash, slot and slot hash equal the attested values, re-verifies the vendor evidence, and wraps the model's 32-byte data key to the enclave's HPKE key (12.7). It then reads the chain again and releases only if the attestation, its revision, the broker key and the policy hash are unchanged. An audit record (model, node, operator, attestation and revision, evidence hash, enclave key hash, policy hash, read slot, outcome and reason codes) is persisted before the ciphertext leaves; denials are recorded the same way. These records are public at `/v1/key-releases`, so the release log can be compared with the chain by anyone. Raw evidence, data keys and ciphertexts never enter the API or logs.

11.8 Operator node runtime

A node is one TDX confidential VM with 1 to 8 H100 or H200 GPUs in confidential-computing mode, running the measured Sealroom runtime, a reproducibly built image on Confidential Containers.

Component	Function
Attestation agent	Generates the enclave keys at boot, binds them into REPORTDATA and the GPU nonces, submits evidence, re-attests before expiry

Component	Function
Key client	Requests the data key, unwraps it after checking the release context, decrypts weights into CVM and GPU protected memory
vLLM	Serves architectures the pinned build supports; no owner-supplied code (<code>trust_remote_code</code> is off)
x402 middleware	Returns 402 for unpaid requests, verifies payment, settles after inference
Voucher signer	Signs <code>upto</code> usage vouchers with the enclave Ed25519 key
TLS termination	Terminates TLS inside the CVM with the attested TLS key
Egress allowlist	Outbound connections only to the key broker, the RPC and the facilitator

A call proceeds as follows:

1. The caller connects over attested TLS (12.8) and sends its request with request ID, model, node, lane and scheme.
2. Without a `PAYMENT-SIGNATURE` header the node answers 402 with x402 v2 `PAYMENT-REQUIRED` .
3. The caller's wallet signs only its own slot of the payment transaction, and the caller retries the identical request.
4. The node checks that the payload matches the quote and has the facilitator `/verify` it.
5. The node checks eligibility and that its key release belongs to the current attestation revision, runs inference, and checks both again.
6. The node calls `/settle` (and for `upto` signs the voucher and calls `/settle-voucher`), then returns the completion with `PAYMENT-RESPONSE` .

If eligibility fails before settlement, no payment is taken; once payment succeeds, the completion produced under a valid attestation is returned. The runtime stops serving when its attestation lapses.

11.9 Deployment topology

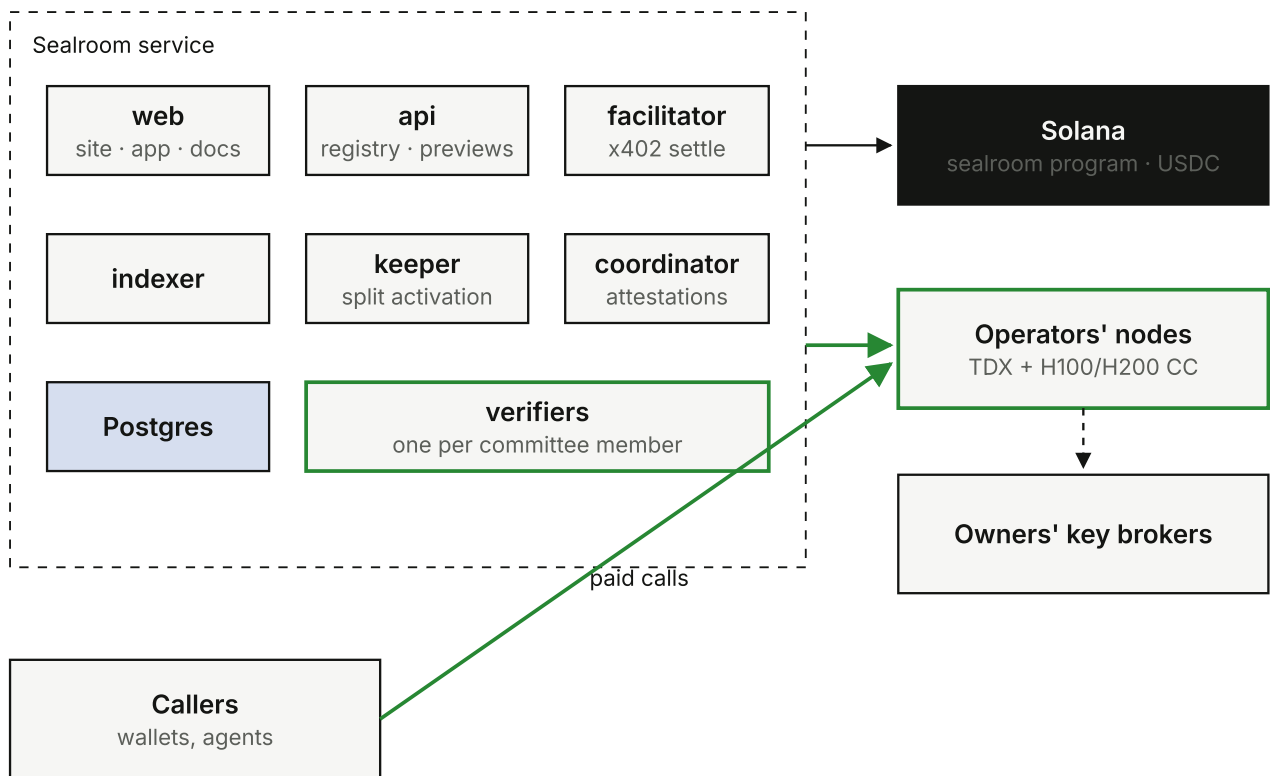


Figure: Sealroom's services over Postgres and a Solana RPC, with owner-run key brokers and operator-run nodes.

Sealroom runs the web app, public API, indexer, keeper, coordinator and facilitator over one Postgres database and a Solana RPC endpoint. The five verifiers run as five separate services, each with only its own key and credential; Sealroom runs at most two, and independent organisations run the others on their own hosts. Model owners run key brokers. Operators run nodes in their vetted datacenters.

Process	Talks to	Records it owns
Public API	Postgres, RPC	Applications, idempotency records, rate buckets
Indexer	RPC, Postgres	Raw accounts, projections, events, checkpoints, history cursors
Keeper	RPC, Postgres, facilitator, Jupiter swap API	Jobs, budgets, sponsored transactions, Claim charges
Coordinator and verifiers	Each other, RPC, vendor collateral services	Evidence archive by hash
Facilitator	RPC, Postgres	Payment sessions and signed intents
Key broker	RPC, its key store	Broker audit records
Node	Coordinator, broker, facilitator, RPC	None; keys stay in the CVM

Each service runs as its own process with its own health check, and schema migrations apply at service start. Chain state is authoritative for the registry and money; services append their own

records and never rewrite chain history.

12. Security design

Sealroom protects model weights and their data keys, callers' prompts, USDC in lane, bond, dispute and buyback vaults, and the registry that decides which machines receive keys.

12.1 Threat model

Threat	Adversary capability	Defense
Impersonate a node with a purchased server	Owns a genuine TDX machine and can extract its attestation keys physically	Platform pinning, operator vetting (12.2)
Run software that copies weights or prompts	Controls the host and boots another image	Measured runtime, timelocked allowlist, owner pins (12.3)
Replay evidence or borrow GPU evidence	Holds old quotes, controls other GPUs	Fresh keys bound to a slot hash, UEID pins (12.4)
Corrupt the committee	Controls up to two verifiers	Three-of-five threshold, broker re-verification (12.5)
Keep serving after a compromise is found	Holds a once-valid node	24-hour TTL, immediate revocation, operator epoch (12.6)
Capture the data key	Controls network and host OS	HPKE to the enclave key (12.7)
Read prompts at the operator	Terminates TLS outside the CVM	Attested TLS (12.8)
Overcharge metered calls	Controls the host's billing path	Enclave-signed vouchers (12.9)
Physically attack a registered server	Physical access in a vetted facility	Operator Agreement, USDC bond, arbitration (12.10)
Change rules quickly	Holds the admin key	Timelocks, separate multisigs (12.11)
Divert protocol fees	Holds the buyback authority key or crafts a swap route	Fixed buyback vault, program-signed swap and burn in one instruction (12.12, 14.7)
Substitute accounts	Crafts instruction account lists	Program account checks (12.12)

12.2 Platform pinning and operator vetting

A genuine Intel platform is not enough to receive a key: release accepts only quotes whose platform identity and GPU device IDs a vetted, bonded operator registered. A machine bought on the open

market therefore cannot be presented as a node, whatever attestation keys it holds.

1. Sealroom vets the operator's legal entity, datacenter certifications and signed Operator Agreement; the admin multisig then calls `register_operator`.
2. `register_platform`, signed by both operator and admin, stores the hash of the Intel PPID/PCK and of 1 to 8 distinct GPU UEIDs, and creates a global `PlatformPin` (`["platform_pin", platform_id_hash]`) and a global `GpuPin` (`["gpu_pin", ueid_hash]`) per GPU. Pins are created only at empty addresses, so a platform or GPU belongs to one operator and one platform.
3. `register_node` claims named UEIDs from its platform; each pin's `node` field is set once, so a GPU serves one node.
4. Platform, node and attestation registration require an operator that is not frozen and holds its tier's bond minimum.
5. Verifiers and brokers require the quoted platform ID and GPU UEIDs to match the pins for this node.

NVIDIA GPU evidence is not bound to a particular CVM; evidence produced on any GPU outside the node's pinned set fails the UEID check.

12.3 Measured runtime and owner-pinned measurements

The runtime is built reproducibly, so its 48-byte TDX measurement can be recomputed from source. Each allowed image has a `Runtime` account at `["rt", SHA256(measurement)]` with image name, version and 40-character source commit. `allow_runtime` makes a measurement effective 48 hours later; `revoke_runtime` is immediate. `record_attestation` requires an effective, unrevoked runtime equal to the attested measurement. Each model pins 1 to 8 measurements, and eligibility requires both lists, so a key goes only to an image that the protocol has allowed for at least 48 hours and that the owner chose.

12.4 Evidence bound to fresh keys and a slot hash

The CVM generates its keys at boot. Their public parts form a 1,280-byte transcript:

```
transcript      = hpke_public_key[1216] || tls_public_key[32] ||  
voucher_public_key[32]  
enclave_key_hash = SHA256(transcript)  
H               = SHA256(transcript || recent_slot_hash)  
REPORTDATA[0..32] = H  
gpu_nonce[i]    = H           for every GPU i
```

The slot and its hash must appear in the SlotHashes sysvar (at most 512 entries) when a verifier signs and when `record_attestation` executes, so evidence is recorded while its slot is recent. The canonical 200-byte message binds the domain `sealroom.att.v0`, program ID, genesis hash, node

PDA, committee epoch, next sequence, the hash of the 194-byte commitment (platform, measurement, RIM hash, evidence hash, enclave key hash, TCB status, expiry, method, operator epoch), slot and slot hash. A signature cannot be reused on another program, deployment, node, sequence or evidence, and the private keys never leave the CVM.

12.5 Three-of-five committee and broker re-verification

The `VerifierSet` holds exactly five distinct members at threshold three (allocation up to seven). `record_attestation` reads the preceding Ed25519 precompile instructions through the Instructions sysvar and requires every offset to point inside the same instruction, every message to equal the canonical message, every signer to be a member of the active epoch, and at least three distinct signers, tracked in an 8-bit bitmap. The Solana runtime verifies the signatures. Sealroom runs at most two verifiers, so every quorum includes an independent member, and the coordinator holds no verifier key.

The owner's broker checks independently of the committee: it reads the chain itself, re-verifies the vendor evidence and requires the evidence, keys, slot, measurement and RIM hash to equal the attested values. A committee-signed record built on evidence that fails vendor verification is refused at the broker.

12.6 Short attestation lifetime and immediate revocation

- **TTL.** `expires_at` lies after the current chain time, at most 86,400 s later, and no later than any pending unbond deadline.
- **Revocation.** The admin or any current verifier can `revoke_attestation`, effective in the same transaction. It consumes the next sequence and latches the record, so a renewal signed for that sequence cannot land. `clear_attestation_revocation` (admin or arbitration) lifts the latch and expires the record, so a fresh quorum is required.
- **Operator epoch.** Arbitration can advance an operator's `attestation_epoch`. Every commitment includes its epoch, and older epochs are treated as revoked, so one ruling invalidates all of that operator's attestations.
- **Freeze and pause.** A frozen operator fails platform and node registration, attestation and eligibility. While paused, registration, attestation, model creation, price and policy updates, split changes, lane opening, distribution and the buyback stop and brokers release nothing; revocation, freezing, rulings, withdrawals and claims remain available.
- **Runtime.** The node checks eligibility before inference and before settlement, and stops serving when its attestation lapses.

12.7 Key release only through HPKE to the enclave key

Weights are encrypted with a random 256-bit data key (AES-256-GCM, chunked); the ciphertext can be stored anywhere. The data key leaves the broker only as an HPKE ciphertext:

Element	Value
Mode	RFC 9180 base mode, one sender context per release
Suite	X-Wing (X25519 + ML-KEM-768), HKDF-SHA256, AES-256-GCM
info	<code>sealroom.owner-key-release.v1</code>
AAD	<code>sealroom.key-release.v1</code> , program, genesis hash, model, node, attestation and signed revision, evidence hash, request ID
Recipient	The HPKE key whose hash is committed in <code>enclave_key_hash</code>

Only the CVM holding the matching private key can open it, and the key client refuses an envelope whose context differs from the one it expects. The broker persists its audit record before the ciphertext leaves and zeroes the data key after every attempt. The default broker is run by the owner, so no new party holds keys.

12.8 Attested TLS

The TLS key is generated in the CVM and is part of the committed transcript; the API exposes its hash as `tlsPublicKeyHash`. Before sending a prompt, the caller's SDK reads the node's attestation, requires status `valid`, and requires the presented TLS key to hash to `tlsPublicKeyHash` and the transcript to `enclave_key_hash`. The session ends inside the measured CVM, and the operator's host sees only ciphertext. Node endpoints are HTTPS URLs validated on chain.

12.9 Usage vouchers signed by the enclave key

A metered charge is set by a voucher only the enclave can sign. The voucher (domain `sealroom:usage:v1`) carries lane, channel, request hash, nonce, cumulative amount, expiry, attestation and signed revision, and signer; the signed message also binds the genesis hash. The facilitator accepts it only when the signer is the voucher key in the node's current attestation and every field matches the session. The amount comes from the model's on-chain rates and is capped by `max_per_call` and the caller's ceiling; altered or replayed vouchers fail.

12.10 Bonds and arbitration

Each operator's USDC bond sits in a vault owned by its `Operator` PDA and backs the Operator Agreement. Deposits must be positive and are refused while frozen or unbonding. Unbonding takes 30 days and is refused while frozen or under an open dispute, and no attestation may expire after the unbond deadline, so no attested node outlives the locked bond. A model owner with a paid lane opens a dispute with an evidence hash, a deposit and admission credit (14.6), within per-claimant and per-operator caps. The arbitration multisig rules once: for the claimant it can slash up to the bond principal, freeze the operator and advance its epoch; a dismissal cannot slash, freeze or revoke. `arbitrate_operator` lets arbitration freeze or revoke at any time without a dispute. Slashes are reserved at ruling time and moved by a permissionless `settle_slash`; security rulings make no token transfer, so a frozen token account cannot block them.

12.11 Timelocked governance

Admin and arbitration are separate Squads multisigs and must differ, which the program enforces at initialization and on every arbitration change. Changes to who is paid or which software receives keys wait before applying: 48 hours for a runtime, the fee rate and the dispute configuration; seven days for a split; 30 days for a bond withdrawal. The protocol fee's destination has no change path at all: it is the buyback vault, derived from the program id and the USDC mint. The buyback mint is selected once and never changes. A pending arbitrator has no authority. Owners' runtime lists, allowlists and minimums apply on top of protocol defaults; an owner replaces them with `update_policy`, and the price with `update_price`, effective at once and without moving funds.

12.12 Program account checks

- **Ownership and type.** Typed accounts are checked for program ownership and discriminator; hand-written accounts (GPU pins, Claims) must be program-owned, writable and at their re-derived PDA and bump.
- **PDA seeds.** Every account loads at its canonical address and stored bump, e.g. `["op", authority]`, `["node", platform, index]`, `["att", node]`, `["model", owner, id]`, `["lane", model, node]`, `["claim", lane, recipient_ata]`, with parent links enforced by `has_one`.
- **Signers.** Admin and arbitration must equal the keys in `Config`; operators and owners are bound through their PDA seeds; claimants and beneficiaries must equal the recorded beneficiary. `initialize` must be signed by the upgrade authority recorded in `ProgramData`.
- **Tokens.** The mint must be `Config.usdc_mint` with 6 decimals, transfers use `TransferChecked`, vaults are canonical ATAs of their PDA, split recipients and the forfeiture recipient must be canonical ATAs of the declared wallet, and the buyback vault and output account are the protocol PDA's canonical ATAs.
- **Splits.** 1 to 8 positive entries summing to 10,000 bps, unique recipients, exactly one unlockable Operator entry routed to the lane operator; reductions of locked entries need the beneficiary's signature.

- **Distribution.** Recipient and Claim accounts must match the split, be writable and differ from the vault; the fee account must be the buyback vault; every Claim is authenticated before the first transfer; arithmetic is checked, and fee plus paid plus deferred must equal the available balance.
- **Buyback.** Only `Config.buyback_authority` starts it; the swap program must be Jupiter v6 and the route an exact-input route with zero platform fee, the buyback vault as source, the protocol output account as destination and no signer but the protocol PDA; the vault must fall by exactly `amount`, the output must reach `min_out` and is burned in full, and owners, mints, delegates, close authorities and supply are checked around the swap.

Off chain, every chain-reading service pins the expected genesis hash and program ID, and unset security configuration (challenge secret, vendor adapter, wrapping provider, keeper budgets) fails closed.

13. Parameters and governance

13.1 Protocol parameters

Durations and size limits are program constants; economic values live in `Config` and change only through the instructions listed. `/v1/parameters` reports each value with its source, and a value not yet set as `unset` with the operation it blocks.

Parameter	Value	Where set	Who can
Protocol fee	500 bps at launch; at most 10,000	<code>Config.protocol_fee_bps</code>	Admin p anyone 48 h
Fee destination	Buyback vault: canonical USDC ATA of the protocol PDA <code>["protocol"]</code>	Derived from the program id and USDC mint	Fixed
Buyback authority	The keeper key that signs <code>buyback_swap_and_burn</code>	<code>Config.buyback_authority</code>	Admin <code>set_buy</code>
Buyback mint	No mint authority, positive supply of at most 1,000,000,000 tokens, SPL Token or Token-2022 with metadata extensions only	<code>Config.buyback_mint</code>	Admin <code>set_buy</code>
Swap program	Jupiter v6, <code>JUP6LkbZbjS1jKKwapdHNy74zcZ3tLUZoi5QNyVTaV4</code>	Constant <code>JUPITER_PROGRAM</code>	Program
Attestation TTL	24 h (86,400 s)	Constant <code>ATTESTATION_TTL</code>	Program
Committee	5 distinct members, threshold 3	<code>VerifierSet</code> , written by <code>initialize</code>	Fixed pe epoch
Verifier keys per set	Up to 7	Constant <code>MAX_VERIFIERS</code>	Program
Unbonding period	30 days (2,592,000 s)	Constant <code>UNBONDING_PERIOD</code>	Program
Split delay	7 days (604,800 s)	Constant <code>SPLIT_DELAY</code>	Program
Runtime allowlist timelock	48 h (172,800 s)	Constant <code>RUNTIME_TIMELOCK</code>	Program
Fee and dispute-config timelock	48 h	Same constant	Program

Parameter	Value	Where set	Who can set
Split entries	1 to 8	<code>MAX_SPLIT_ENTRIES</code>	Programmer
Allowed runtimes per model	1 to 8	<code>MAX_ALLOWED_RUNTIMES</code>	Programmer
Operator allowlist per model	0 to 16 (empty admits any eligible operator)	<code>MAX_ALLOWED_OPERATORS</code>	Programmer
Node endpoint, broker URL	HTTPS, at most 128 bytes each	<code>MAX_ENDPOINT_BYTES</code> , <code>MAX_KEY_BROKER_URL_BYTES</code>	Programmer
GPUs per platform and node	1 to 8; H100 or H200	<code>MAX_GPUS</code> , <code>GpuModel</code>	Programmer
Runtime measurement	48 bytes	<code>Runtime.measurement</code>	Programmer
Runtime image metadata	Name at most 64 bytes, version at most 32, commit 40 hex	Program constants	Programmer
Tiers and bond minimums	Tiers 1 to 3, one USDC minimum each	<code>Config.tier_bond_minimums</code>	Written by admin, set by tiers
Model runtimes, allowlist, minimum tier and bond	1 to 8 runtimes; 0 to 16 operators; tier 1 to 3; any USDC amount	<code>Model.allowed_runtimes</code> , <code>Model.operator_allowlist</code> , <code>Model.min_tier</code> , <code>Model.min_bond</code>	Owner, if owner updates immediately
Model price	Exact { <code>per_call</code> } or Upto { <code>input_per_million_tokens</code> , <code>output_per_million_tokens</code> , <code>max_per_call</code> }	<code>Model.price</code>	Owner, if owner updates immediately
Derivative royalty	0 to 10,000 bps	<code>Model.derivative_royalty_bps</code>	Owner, if owner updates immediately
Operator share	Weight of the split's Operator entry	<code>Model.split</code>	Owner, if anyone updates after 7 days
Default operator share	Protocol default reported by the API	<code>Config.default_operator_share_bps</code>	Reserved
Dispute deposit	Positive USDC amount	<code>Config.dispute_deposit</code>	Admin, if anyone proposes anyone

Parameter	Value	Where set	Who can
Open disputes per claimant and operator	Positive integer	<code>Config.max_open_per_claimant_operator</code>	Same pa
Open disputes per operator	Default 8	<code>Config.max_open_per_operator</code>	Same pa
Dismissed deposit	<code>ForfeitOnDismissal</code> to a canonical ATA whose owner is not arbitration	<code>Config.dispute_forfeiture_ata</code>	Same pa
Arbitration	Squads multisig, not the admin	<code>Config.arbitration</code>	initial dispute-
Admin	Squads multisig	<code>Config.admin</code>	initial
Claim policy, claim destination	<code>BeneficiaryOnlyNoExpiry</code> ; <code>CanonicalAta</code> (default) or <code>AnyOwnedUsdc</code>	<code>Config</code>	Admin set_set
Split settlement	<code>LaneBoundary</code>	<code>Config.settlement_mechanism</code>	Fixed at
Pause	true or false	<code>Config.paused</code>	Admin s
USDC mint	Six-decimal SPL mint	<code>Config.usdc_mint</code>	initial
Mainnet beta caps	Total bonds and total vaults; the brief's example is \$250k	Reserved <code>Config</code> fields	Admin o
Data key, weight cipher	256-bit key, AES-256-GCM	Broker and runtime formats	Runtime
Enclave keys	HPKE (X25519 + ML-KEM-768), TLS, Ed25519 voucher	Runtime	Runtime
Committee message	200 bytes; commitment 194 bytes	Program	Program
Legacy transaction size	1,232 bytes	Solana	Network

13.2 Service parameters

Parameter	Value
Index freshness: maximum age; maximum lag	30,000 ms; 150 slots
Indexer poll; backfill bound	2,000 ms; 10,000 signatures

Parameter	Value
API page size; public write rate	1 to 100, default 20; 20 per minute per address
Keeper poll; jobs per tick	15,000 ms in production, 5,000 ms locally (<code>KEEPER_POLL_MS</code>); 32
Keeper lease default and maximum; RPC timeout	60,000 ms and 180,000 ms; 10,000 ms
Keeper retry backoff; blocked re-alert	<code>min(300 s, 1 s * 2^min(attempts, 8))</code> ; 900 s
Keeper global, owner and model caps; window	Set per deployment, no defaults
Buyback poll; minimum input; slippage	60,000 ms by default; <code>BUYBACK_MIN_USDC</code> and <code>BUYBACK_SLIPPAGE_BPS</code> set per deployment, no defaults
Coordinator verifier timeout	30 s
Payment session timeout; compute limit	120 s; 100,000 CU at price 0
Evidence size	At most 1 MiB

13.3 Authorities

Authority	Holder	Powers
Upgrade authority	Sealroom	Signs <code>initialize</code> ; deploys upgrades
Admin	Squads multisig	Operator admission, tier and freeze; co-signs <code>register_platform</code> ; runtime allowlist; <code>revoke_attestation</code> , <code>clear_attestation_revocation</code> ; pause; settlement policy; proposes and cancels fee and dispute-config changes; <code>set_buyback_authority</code> ; <code>set_buyback_mint</code> , once
Arbitration	Separate Squads multisig	<code>rule_dispute</code> , <code>arbitrate_operator</code> , <code>clear_attestation_revocation</code>
Buyback authority	The keeper's key	<code>buyback_swap_and_burn</code> ; holds no protocol funds
Verifier committee	Five members, three needed	Signs attestations; any member may <code>revoke_attestation</code>
Model owner	Owner wallet	<code>create_model</code> , <code>update_price</code> , <code>update_policy</code> , <code>propose_split</code> , <code>cancel_split</code> , <code>open_dispute</code> ; runs the key broker
Operator	Operator authority	Bond lifecycle, co-signs <code>register_platform</code> , <code>register_node</code>

Authority	Holder	Powers
Beneficiaries	Recipient wallets	Consent to locked-entry reductions; claims
Anyone	Any wallet	<code>record_attestation</code> with a quorum, <code>open_lane</code> , <code>prepare_claim</code> , <code>distribute</code> , <code>claim_protocol_fee</code> , <code>apply_split</code> , <code>apply_fee</code> , <code>apply_dispute_config</code> , <code>settle_slash</code>

13.4 The multisig and timelock model

The admin admits operators, curates runtimes and proposes economic changes; the arbitration panel rules on disputes and takes security actions. Both sign through the Squads multisig program, and their keys must differ, so no single authority both changes dispute rules and rules on disputes.

Change	Propose	Wait	Apply	Cancel
Allow a runtime	Admin <code>allow_runtime</code>	48 h	At <code>effective_at</code>	Admin <code>revoke_runtime</code>
Fee rate	Admin <code>propose_fee</code>	48 h	Anyone <code>apply_fee</code>	Admin <code>cancel_fee</code>
Arbitration, deposit, caps, forfeiture	Admin <code>propose_dispute_config</code>	48 h	Anyone <code>apply_dispute_config</code>	Admin <code>cancel_dispute_config</code>
Model split	Owner <code>propose_split</code>	7 days	Anyone <code>apply_split</code> , then per-lane activation	Owner <code>cancel_split</code>
Bond withdrawal	Operator <code>request_unbond</code>	30 days	Operator <code>withdraw_bond</code>	Operator <code>cancel_unbond</code>

1. One proposal of each kind may be pending.
2. A fee proposal changes the rate only; every fee goes to the buyback vault, whose address no proposal can change.
3. A dispute-config proposal requires arbitration distinct from the admin and from the forfeiture owner, positive deposit and caps, and `ForfeitOnDismissal`; applying re-checks the separation.
4. A pending arbitrator has no authority, and the direct setter `set_dispute_config` always returns `GovernanceTimelockRequired`.
5. `apply_split` is refused while paused; cancellation stays available.

Protective actions apply immediately: `revoke_runtime`, `revoke_attestation`, `freeze_operator`, `set_paused` and rulings. So do an owner's `update_price` and `update_policy`, and the admin's `set_buyback_authority` and one-time `set_buyback_mint`. Claim beneficiaries, forfeiture recipients and ruling authorities are snapshotted into accounts and events, so later governance cannot reassign entitlements or re-attribute past actions.

14. Economics

All money in Sealroom is USDC, in base units with 6 decimals: callers pay per call into lane vaults, `distribute` pays the protocol fee, the operator and the split, and bonds and dispute deposits are posted in USDC. The protocol fee is paid into the buyback vault, and `buyback_swap_and_burn` spends it on \$SEALROOM and burns every token it buys (14.7).

14.1 Fee flow

1. **Price.** The owner sets a fixed `per_call` price (`exact`) or per-million-token input and output rates with a `max_per_call` ceiling (`upto`), and changes it with `update_price` , effective at once.
2. **Payment.** Each call pays over x402 into the vault of its (model, node) lane: the price for `exact` , or the voucher amount from an escrowed ceiling for `upto` .
3. **Accrual.** The vault accumulates payments; nothing is split per call.
4. **Distribution.** Anyone calls `distribute(lane)` . One all-or-nothing transaction pays the whole fee into the buyback vault, the USDC account of the program PDA `["protocol"]` , and every split entry, with the Operator entry routed to the lane operator's payout account.
5. **Deferral.** A share whose account is closed, frozen or owner-changed is reserved as a Claim for its beneficiary; other payouts proceed. A fee the buyback vault cannot receive is reserved the same way, and `claim_protocol_fee` , open to anyone, later delivers it into the vault.
6. **Buyback.** The keeper, as the buyback authority, calls `buyback_swap_and_burn` : in one instruction the program swaps the vault's USDC for \$SEALROOM through Jupiter, signing for its own PDA, and burns the entire output (14.7).

14.2 Distribution arithmetic

```
G    = vault.amount - reserved_amount
F    = floor(G * protocol_fee_bps / 10000)
N    = G - F
s_i  = floor(N * bps_i / 10000)          for entries i = 1 .. k-1
s_k  = N - (s_1 + ... + s_(k-1))        the final entry receives the
rounding dust
F + s_1 + ... + s_k = G
```

Worked example at the launch fee of 500 bps with an eight-entry split of 1,250 bps each:

Available G (base units)	Fee F	Net N	Entries 1 to 7, each	Entry 8	Check
1,000,001	50,000	950,001	118,750	118,751	$50,000 + 7 \times 118,750 + 118,751 = 1,000,001$
10,003	500	9,503	1,187	1,194	$500 + 7 \times 1,187 + 1,194 = 10,003$

The fee and eight recipients fit in one 308-byte v0 transaction with a lookup table and a compute-budget instruction.

14.3 Per-call pricing

```
charge = ceil((input_tokens * input_per_million_tokens + output_tokens *
output_per_million_tokens) / 1000000)
charge <= max_per_call
charge <= caller_ceiling
```

At the brief's assumed specialist price of \$3 to \$10 per million output tokens (3,000,000 to 10,000,000 base units per million), one second of generation at the assumed 1,500 output tokens per second costs:

```
charge_low  = ceil(1500 * 3000000 / 1000000) = 4500 base units =
0.0045 USDC
charge_high = ceil(1500 * 10000000 / 1000000) = 15000 base units = 0.015
USDC
fee_low     = floor(4500 * 500 / 10000)       = 225 base units
fee_high    = floor(15000 * 500 / 10000)      = 750 base units
```

14.4 Illustrative unit economics

The brief's illustrative economics use the listed price of a TEE H100 on Phala, \$3.08 per GPU-hour, and assume an 8B to 32B model sustains 1,500 output tokens per second on one H100.

```
tokens_per_gpu_hour      = 1500 * 3600 = 5400000
compute_per_million_full = 3.08 / 5.4 = 0.57 USD      (full use)
compute_per_million_20   = 3.08 / 1.08 = 2.85 USD     (20% use)
fee_per_million           = 0.05 * price_per_million
```

Quantity	At \$3 per million tokens	At \$10 per million tokens
Compute cost per million tokens, full use / 20% use	\$0.57 / \$2.85	\$0.57 / \$2.85
Protocol fee per million tokens (5%)	\$0.15	\$0.50
Caller payments per GPU-hour, full use (5.4M tokens)	\$16.20	\$54.00
Protocol fee per GPU-hour, full use	\$0.81	\$2.70
Paid to the split per GPU-hour, full use	\$15.39	\$51.30
Caller payments per GPU-hour, 20% use (1.08M tokens)	\$3.24	\$10.80

The GPU-hour costs \$3.08 whether it serves 5.4 million tokens or 1.08 million, so utilization sets the operator's margin.

14.5 Protocol revenue

```
protocol_revenue = caller_payments * 500 / 10000
200000000 USD * 0.05 = 10000000 USD
```

Annual caller payments	Protocol revenue at 5%	Inference tokens at \$3 per million	Inference tokens at \$10 per million
\$20M	\$1M	6.67 trillion	2 trillion
\$50M	\$2.5M	16.7 trillion	5 trillion

All protocol revenue lands in the buyback vault and is spent only by `buyback_swap_and_burn` on \$SEALROOM, which is burned (14.7). The 5% fee is comparable to OpenRouter's 5 to 5.5% fee on credit purchases.

14.6 Operator compensation, royalties and admission credit

Owners and operators choose how the operator is paid. Under **revenue share**, the split's single Operator entry carries a weight set by the owner and pays each lane's own operator, so one model served by several operators pays each from its own lanes; the entry cannot be locked, because a model cannot consent for every operator that may serve it. Under a **reservation**, the owner pays the operator for reserved capacity off-chain and the split's Operator weight reflects that arrangement.

Contributors are paid through the same split. A derivative of a base model whose `derivative_royalty_bps` is positive must carry a locked BaseModel entry of at least that weight paid to the base owner's canonical ATA, and the base owner's broker can release the base key only

to lanes whose split carries it. A locked entry changes only with its beneficiary's signature, and every split change waits seven days and applies per lane at an empty boundary.

Each tier has a protocol bond minimum and each model may require a higher `min_bond`. Dispute admission is priced in protocol fees actually delivered into the buyback vault, directly by `distribute` or later by `claim_protocol_fee`; owner and operator shares never count.

```
credit_available = eligible_protocol_fees_paid -
admission_credit_consumed
admit            = credit_available >= dispute_deposit
gross_for_credit = ceil(dispute_deposit * 10000 / protocol_fee_bps)
at 500 bps:      gross_for_credit = 20 * dispute_deposit
```

`open_dispute` consumes credit equal to the deposit and transfers the deposit. A ruling for the claimant returns the deposit and pays the settled slash; a dismissal forfeits it. Consumed credit is never restored.

14.7 \$SEALROOM

\$SEALROOM is the protocol's token. Its design is deliberately small: a fixed supply, one fair launch, and a single source of demand that comes from paid inference.

Property	Value
Ticker	\$SEALROOM
Supply	1,000,000,000, fixed
Launch	100% to one public launch on equal terms
Allocations	None: no presale, team, investor, airdrop or reserved allocation
Vesting	None
Protocol fee share	All of it buys \$SEALROOM on the market and burns it
Buyback vault	USDC account of the program PDA <code>["protocol"]</code> , spent only by <code>buyback_swap_and_burn</code>
Mint setting	<code>Config.buyback_mint</code> , configured once with <code>set_buyback_mint</code>

Fair launch. The entire supply entered circulation through one public launch, on the same terms for every participant. Nothing is held back for a team, investors or a reserve, and nothing unlocks later, so the supply that exists after the launch is the most that will ever exist; from then on it only falls as tokens are burned.

Buyback and burn. The protocol fee is taken at `distribute` (14.1 and 14.2), in USDC, before any split entry is paid, and lands in the buyback vault: the canonical USDC account of the program PDA `["protocol"]`. The PDA has no private key, and no instruction withdraws from the vault. Its USDC

has one use, `buyback_swap_and_burn(amount, min_out, swap_data)`. The keeper, the key in `Config.buyback_authority`, quotes an exact-input Jupiter route from USDC to \$SEALROOM for the vault balance and submits the instruction with that route; the program then carries out the whole buyback inside it:

1. It accepts only Jupiter v6 and only a `route` or `shared_accounts_route` whose input equals `amount`, whose platform fee is zero, whose source is the buyback vault, whose destination is the protocol PDA's \$SEALROOM account, and in which no account but the protocol PDA signs.
2. It invokes Jupiter, signing for the protocol PDA itself.
3. It requires the vault to fall by exactly `amount`, the output account to rise by at least `min_out`, the \$SEALROOM supply to be unchanged by the swap, and both accounts to keep their owner, mint and empty delegate and close authority.
4. It burns the entire output increase through the mint's own token program, signed by the protocol PDA, and requires the output account to return to its prior balance and the supply to fall by exactly the burned amount.
5. It adds `amount` to `Config.usdc_spent_total` and the burned amount to `Config.burned_total`, and emits `BuybackBurned { usdc_in, burned, slot }`.

A failure at any step reverts the swap and the burn together. The keeper starts the buyback and pays its network fee, and the USDC and the bought \$SEALROOM never sit in its account or any other wallet: they move only between the protocol PDA's accounts and the route, inside the one instruction that ends with the burn. Fees the vault could not receive at distribution reach it through `claim_protocol_fee`, which anyone can submit and which pays nowhere else.

The program takes the \$SEALROOM mint from `Config.buyback_mint`, which the admin configures once with `set_buyback_mint`. That instruction accepts only a mint without a mint authority, with a positive supply of at most 1,000,000,000 tokens, on SPL Token or on Token-2022 with metadata extensions alone, and the selection never changes afterwards.

```
fee(d)          = floor(available(d) * protocol_fee_bps / 10000)
buyback_usdc    = sum of fee(d) over every distribution d
supply(t)       = 1000000000 - burned(t)
at 500 bps:     buyback_usdc = 0.05 * distributed caller payments
```

At the launch fee, every 20 USDC of caller payments that reaches a distribution sends 1 USDC to the buyback. The buyback is funded by paid inference and by nothing else: \$SEALROOM pays no emissions, staking rewards or incentives, and operators earn only from caller revenue through the split (14.6). The fee itself is governed like every other `Config` value: a change is proposed by the admin and applied after 48 hours (Chapter 13).

What stays in USDC. \$SEALROOM sits on the fee side of the protocol. Callers and agents pay over x402 in USDC, so a call never needs a conversion. Bonds, dispute deposits and slashes are USDC, so a ruling pays the claimant in the same unit its calls were priced in. Key release depends only on the six release checks, and operator admission remains a vetting decision. No token, \$SEALROOM or a model's, is accepted as bond collateral or used to gate key release, and Sealroom issues no tokenized model revenue shares.

Appendix A. Instruction reference

Fee and rent payers are also written; they are listed only when they are the signer of record. "Remaining" marks accounts passed after the named ones.

Instruction	Signers	Accounts written	Effect
<code>initialize</code>	admin, as upgrade authority	<code>config</code> , <code>verifier_set</code>	Creates <code>Config</code> (genesis hash, six-decimal USDC mint, fee, treasury binding, three tier minimums, optional arbitration distinct from admin, fixed durations, no buyback mint or authority) and the epoch-0 set of five verifiers at threshold three
<code>set_paused</code>	admin	<code>config</code>	Sets or clears <code>paused</code>
<code>set_buyback_authority</code>	admin	<code>config</code>	Names the non-default key that may sign <code>buyback_swap_and_burn</code>
<code>set_buyback_mint</code>	admin	<code>config</code>	Selects the buyback mint once: no mint authority, positive supply of at most 1,000,000,000 tokens, SPL Token or Token-2022 with metadata extensions only
<code>buyback_swap_and_burn</code>	buyback authority	<code>config</code> , <code>buyback_vault</code> , <code>buyback_mint</code> , <code>output_vault</code> , Jupiter route accounts (remaining)	Swaps <code>amount</code> of the buyback vault's USDC through a checked Jupiter v6 route signed by the protocol PDA, requires at least <code>min_out</code> , burns the entire output and adds to <code>usdc_spent_total</code> and <code>burned_total</code>
<code>register_operator</code>	admin	<code>operator</code> , <code>bond_vault</code>	Admits an operator with agreement hash and tier; creates its bond vault; records its payout ATA
<code>set_tier</code>	admin	<code>operator</code>	Assigns a tier that has a bond minimum
<code>freeze_operator</code>	admin	<code>operator</code>	Sets or clears <code>frozen</code>
<code>deposit_bond</code>	operator	<code>operator</code> , <code>source</code> , <code>bond_vault</code>	Adds a positive USDC amount to principal
<code>request_unbond</code>	operator	<code>operator</code>	Sets <code>unbond_at</code> 30 days ahead; no freeze or open dispute
<code>cancel_unbond</code>	operator	<code>operator</code>	Clears the request
<code>withdraw_bond</code>	operator	<code>operator</code> , <code>bond_vault</code> , <code>destination</code>	After <code>unbond_at</code> , pays principal to the operator

Instruction	Signers	Accounts written	Effect
register_platform	operator, admin	platform, platform_pin, GPU pins (remaining)	Registers a platform ID hash and 1 to 8 GPU UEID hashes; creates global pins
register_node	operator	node, GPU pins (remaining)	Creates a node with key, HTTPS endpoint and GPU model; claims named GPUs exclusively
allow_runtime	admin	runtime	Allows a measurement, effective after 48 h
revoke_runtime	admin	runtime	Revokes a measurement immediately
record_attestation	payer (any)	attestation	Records an attestation signed by three distinct members, after the registry, freshness, expiry, sequence and epoch checks
revoke_attestation	admin or current verifier	attestation	Latches revocation and consumes the next sequence
clear_attestation_revocation	admin or arbitration	attestation	Clears the latch and expires the record
create_model	owner	model	Creates a model with manifest hash, broker, policy, price, royalty, base model and split at revision 1
update_price	owner	model	Replaces the price at once; positive per_call or max_per_call
update_policy	owner	model	Replaces allowed runtimes, min_tier, min_bond and operator allowlist at once
propose_split	owner; beneficiaries of reduced locked entries	model	Stores a pending split, effective after 7 days
cancel_split	owner	model	Removes the pending split
apply_split	none	model	Makes a mature pending split the model default
open_lane	payer (any)	lane, vault	Creates the (model, node) lane and vault; snapshots the split
prepare_claim	payer (any)	claim	Creates the Claim PDA for a lane recipient or the buyback vault

Instruction	Signers	Accounts written	Effect
distribute	none	lane, vault, fee_treasury (the buyback vault), fee_claim, recipient and Claim pairs (remaining)	Pays the fee into the buyback vault and the split, defers unpayable shares, activates a newer split at an empty boundary
claim_deferred	beneficiary	lane, claim, vault, destination	Pays a Claim under the destination policy
claim_protocol_fee	payer (any)	lane, claim, vault, buyback vault	Pays a deferred protocol fee into the buyback vault and credits its eligible part to the lane
set_settlement_policy	admin	config	Sets claim and claim-destination policies
propose_fee	admin	config	Records a pending fee rate with its treasury binding
cancel_fee	admin	config	Removes the pending fee
apply_fee	none	config	Applies the fee after 48 h
propose_dispute_config	admin	config	Records pending arbitration, deposit, caps and forfeiture recipient
cancel_dispute_config	admin	config	Removes the pending configuration
apply_dispute_config	none	config	Applies it after 48 h
set_dispute_config	admin	none	Always returns GovernanceTimeLockRequired
open_dispute	claimant (model owner)	operator, lane, claimant_disputes, dispute, source, deposit_vault	Admits a dispute on paid-fee credit, sequence and caps; takes the deposit
rule_dispute	arbitration	operator, dispute, claimant_disputes, bond_vault, deposit_vault	Rules once: slash, freeze and epoch advance for the claimant, or dismissal
arbitrate_operator	arbitration	operator, claimant_disputes, dispute, deposit_vault	Records a ruled security action; a positive slash needs a paid lane
settle_slash	any signer	operator, dispute, claimant_disputes, bond_vault, deposit_vault	Moves a reserved slash into the dispute vault

Appendix B. Glossary

\$SEALROOM. The protocol's token: 1,000,000,000 fixed supply, launched entirely in one public launch; the whole protocol fee buys it on the market through `buyback_swap_and_burn` and burns it (14.7).

Admission credit. Protocol fees from a lane that reached the buyback vault, directly or through `claim_protocol_fee`, minus credit already used to open disputes.

Arbitration panel. A Squads multisig, separate from the admin, that rules on disputes and can freeze operators, revoke attestations and slash bonds.

Attestation. The on-chain record that a verifier quorum checked a node's evidence: measurement, RIM hash, evidence hash, enclave key hash, TCB status, slot, expiry and signers.

Attestation epoch. A per-operator counter committed into every attestation; advancing it invalidates all of that operator's attestations.

Attested TLS. TLS whose server key is committed in the node's attestation, so the session ends inside the CVM.

Base model. A model referenced by a derivative, whose royalty the derivative's split must carry as a locked entry.

Bond. USDC an operator posts as collateral, slashable by arbitration and withdrawable 30 days after an unbond request.

Buyback vault. The canonical USDC account of the program PDA `["protocol"]`, which receives every protocol fee; its only outflow is `buyback_swap_and_burn`, which swaps the USDC for \$SEALROOM through Jupiter and burns the output in the same instruction.

Caller. An app, enterprise or agent that pays per call over x402.

Claim. A per-lane, per-recipient account holding a share that could not be paid directly, payable only to its beneficiary; a protocol-fee Claim is payable only into the buyback vault.

Committee message. The 200-byte message verifiers sign for one attestation.

CVM. A confidential virtual machine: an Intel TDX guest, with GPUs in confidential-computing mode.

Data key. The random 256-bit key that encrypts a model's weights.

Dispute. A model owner's complaint against an operator, ruled once by arbitration.

Distribute. The permissionless instruction that pays a lane's balance to the buyback vault as the fee, the operator and the split.

Enclave key hash. SHA-256 of a node's HPKE, TLS and voucher public keys.

Evidence. A node's TDX quote, GPU reports, public keys and recent slot hash.

Exact. The fixed-price x402 scheme.

Facilitator. The service that validates, sponsors and settles x402 payments.

HPKE. Hybrid Public Key Encryption; Sealroom wraps data keys with X-Wing (X25519 + ML-KEM-768).

Keeper. The service that sponsors time-driven permissionless transactions such as split activation and protocol-fee delivery, and starts `buyback_swap_and_burn` as the buyback authority.

Key broker. The owner-run service that wraps a model's data key to an attested enclave.

Lane. The account for one (model, node) pair, with its own USDC vault and active split.

Locked entry. A split entry that changes only with its beneficiary's signature.

Measurement. The 48-byte TDX measurement of the runtime image.

Model. The on-chain listing of an encrypted model: manifest hash, broker, policy, price and split.

Node. One CVM with 1 to 8 GPUs running the measured runtime.

Operator. A vetted datacenter company that posts a bond and registers platforms and nodes.

Platform. A physical server identified by its Intel PPID/PCK hash and its GPU UEIDs.

Pin. A global account tying a platform to one operator, or a GPU to one platform and one node.

Protocol fee. The basis points of each distributed amount that `distribute` pays into the buyback vault; 500 bps at launch.

REPORTDATA. The TDX quote field carrying the key-and-slot binding.

Runtime. The reproducibly built CVM image that serves inference.

Sequence. An attestation's replay counter; each signed revision uses the next value.

Slot hash. A recent Solana slot's hash, used to prove evidence freshness.

Split. Up to eight weighted recipients of a model's net revenue, summing to 10,000 bps.

TCB status. The vendor's verdict on firmware and microcode; only `UpToDate` is accepted.

Tier. An operator class from 1 to 3 with a bond minimum.

UEID. A GPU's device identity, pinned to one platform and node.

Upto. The metered x402 scheme, settled by an enclave-signed voucher.

Vault. A program-owned USDC account for lane payments, bonds, dispute funds or protocol fees.

Verifier. A committee member that checks evidence and signs the committee message.

Voucher. An enclave-signed statement of metered usage.

x402. The HTTP 402 protocol through which callers pay per call.