

Bonded whitepaper

Policy vaults and slashable USDC bonds on Solana: the mechanism, its formulas and parameters, the program, the services around it, security design and deployment.

Version 1.1 · 23 October 2026 237 pages, A4

Contents

1. Introduction

- 1.1 How agents pay today
- 1.2 Why metered services bill in arrears
- 1.3 What Bonded is
- 1.4 What this paper covers
- 1.5 Notation

2. System overview

- 2.1 Actors and keys
- 2.2 The three lanes
- 2.3 Accounts at a glance
- 2.4 The life of a tab
- 2.5 Where each piece runs

3. Policy vaults

- 3.1 The vault account and its policy
- 3.2 Creating a vault and setting policy
- 3.3 Window accounting
- 3.4 Merchant rules and the checks in pay
- 3.5 Approvals and the co-signer
- 3.6 Session key rotation
- 3.7 The guardian and freezing
- 3.8 Withdrawals from the vault

4. The float

- 4.1 Why the float exists
- 4.2 The allowance
- 4.3 A float payment
- 4.4 refillfloat
- 4.5 Every instruction that touches the float
- 4.6 The float as a loss bound
- 4.7 Worked example

5. Tabs and vouchers

- 5.1 The bonded-tab scheme
- 5.2 The 96-byte voucher
- 5.3 The v2 domain tag
- 5.4 Cumulative amounts and nonces
- 5.5 Merchant-side verification
- 5.6 settletab on chain
- 5.7 Transaction size and compute
- 5.8 Batching and settlement cadence
- 5.9 Voucher attestations

6. Bonds and credit lines

- 6.1 The Bond account and postbond
- 6.2 openline and the CreditLine account
- 6.3 The coverage ratio
- 6.4 resizeline
- 6.5 Bond withdrawals
- 6.6 Public coverage as the merchant's credit signal

7. Defaults and claims

- 7.1 When settletab finds the vault short
- 7.2 The claim window and fileclaim
- 7.3 executedefaultpayout
- 7.4 Worked examples
- 7.5 The keeper's part
- 7.6 What the operator's record shows afterwards

8. Escrow

- 8.1 The merchant-signed quote
- 8.2 payescrow and the Escrow account
- 8.3 The state machine
- 8.4 Deadlines and windows
- 8.5 The arbiter and resolve
- 8.6 Co-signing above the threshold
- 8.7 Escrow fees

9. Program architecture

- 9.1 The program
- 9.2 Accounts
- 9.3 Protocol configuration
- 9.4 The instruction set
- 9.5 Token accounts and authorities
- 9.6 Events
- 9.7 Compute and size budgets
- 9.8 Upgrade authority and program versioning

10. Fees and settlement

- 10.1 What the protocol charges
- 10.2 The tab settlement fee
- 10.3 The escrow fee
- 10.4 Ceilings, the fee treasury and governance
- 10.5 Solana costs
- 10.6 Worked example: settling a 100 USDC tab
- 10.7 \$BONDED

11. Off-chain services

- 11.1 The API
- 11.2 The indexer and projections
- 11.3 The keeper
- 11.4 The guardian worker
- 11.5 The merchant middleware
- 11.6 The agent SDK and the MCP server

12. Security design

- 12.1 Key separation
- 12.2 What the program enforces on every payment
- 12.3 Replay protection
- 12.4 Transaction integrity between server and wallet
- 12.5 The SDK's independent pins
- 12.6 Bond safety by construction
- 12.7 The freeze path
- 12.8 Service security
- 12.9 Upgrade authority and verified builds

13. Infrastructure and deployment

- 13.1 What runs where
- 13.2 The services
- 13.3 Deploying the program
- 13.4 Configuration
- 13.5 Checking a deployment
- 13.6 Upgrades

14. Parameters

- 14.1 Program constants
- 14.2 Protocol configuration
- 14.3 Operator and vault policy
- 14.4 Service configuration
- 14.5 Derived quantities

Appendix A. Instruction reference

- A.1 Conventions and summary
- A.2 Protocol
- A.3 Operator, bond and lines
- A.4 Settlement and defaults
- A.5 Vault and payments
- A.6 Escrow
- A.7 Freeze and guardians

Appendix B. Account layouts

- B.1 Conventions
- B.2 Protocol
- B.3 Operator and bond
- B.4 Vault and policy accounts
- B.5 Lines and defaults
- B.6 Freeze
- B.7 Escrow

Appendix C. Error codes

- C.1 Program errors
- C.2 API error codes

Appendix D. Glossary

Appendix E. References

1. Introduction

Bonded is a Solana program, `bonded`, and a set of off-chain services that let an AI agent buy metered services on credit and pay for delayed jobs through escrow. Each agent spends from a policy vault whose limits the program enforces. Each operator, the company that runs the agent, posts a USDC bond that a merchant can claim against when the agent's vault cannot pay what the agent signed for. This chapter describes how agents pay today, why that leaves metered merchants unable to bill an agent in arrears, and what Bonded adds. It ends with a map of the paper and the notation used throughout.

1.1 How agents pay today

On stablecoin rails, merchants that sell to agents take payment in advance for each request, which is how x402 works [6][7]. x402 turns the HTTP status code 402 Payment Required into a payment handshake. The client requests a resource. The server answers 402 with its payment requirements: the scheme, the network, the asset, the amount and the recipient. The client builds a payment that meets them and retries the request with the payment attached. The server, usually through a facilitator, verifies and settles the payment, then serves the response. Payment comes first, and each request carries its own payment.

On Solana the `exact` scheme carries a signed transaction whose payment is a top-level SPL

`TransferChecked` inside a fixed layout of three to six instructions. The facilitator checks that layout, pays the network fee and submits the transaction [7].

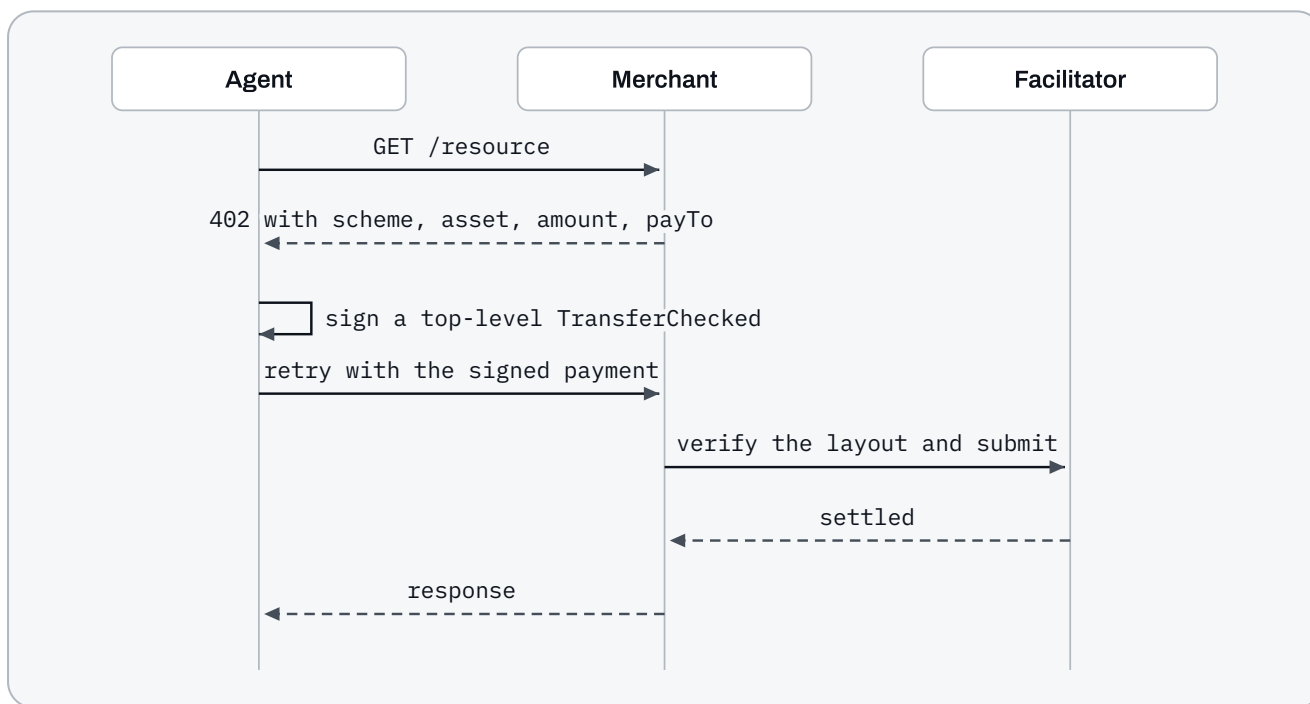


Figure 1. A prepaid x402 request: the agent pays before the merchant serves.

Solana carried about 70% of monthly x402 volume in mid-August 2026, in a 30-day window of roughly \$24.24M across all chains [8]. In January 2026 the all-chain daily volume was \$69k, an average of about \$0.07 per payment [9]. Two facilitators, Dexter and PayAI, carried most Solana x402 transactions in early 2026 [10].

A USDC transfer is final. No network reverses it, so a merchant paid in advance carries no payment risk, and the payer carries all of it. Prepayment is how a merchant protects itself from a client it cannot identify, and per-request prepayment is the form x402 standardizes.

The payer's losses come from the agent's own spending. An attacker who reached an operator's dashboard queued transfer prompts that moved 55.5 ETH, about \$106k [21]. A probable decimal error on a 4 SOL tip sent 52.4M LOBSTAR [20]. A gifted NFT unlocked a permission tier, and a hidden instruction relayed to the agent moved about 3B DRB, \$155k to \$174k [19]. In each case a recipient allowlist or a per-transaction cap denominated in USDC would have bounded the loss, and those are checks a program can run before money moves.

1.2 Why metered services bill in arrears

The first merchants Bonded serves sell metered services to agents: inference, data and compute. Their price is a function of what a request consumed, such as tokens generated, rows returned or seconds of GPU time, and that quantity is known only after the work is done. Prepaying each request forces the merchant either to charge a flat price per call or to take a maximum up front and refund the difference. It also means one on-chain transfer per call, each with its own signature fee [27] and facilitator round trip.

A merchant that bills in arrears serves first, meters what was used, and settles a batch of usage in one transfer. There are signs of demand for this on agent rails: Cloudflare proposed a deferred x402 scheme for batch settlement [13], and a Dexter-linked author published a Solana payment-channel draft [35].

Billing in arrears means extending credit to the client until settlement. A human customer comes with an identity, a contract and a card network's dispute rules. An agent comes with a key. When a merchant serves an agent on credit it holds a promise, and when it comes to collect, the agent's wallet may be empty, its key may have been replaced, and the operator behind it may be a name the merchant cannot reach. A message signed by the agent's key proves the amount owed, but on its own it gives the merchant no funds to collect from.

To bill an agent in arrears, a merchant needs the following before it serves:

1. Proof of the amount owed that it can present without the agent's cooperation. A signature by the agent's key over a running total serves, because the merchant can submit it alone.
2. A ceiling on its exposure, fixed in advance, so that it knows the most it can lose before it extends any credit.
3. Collateral it can reach under rules that do not depend on the operator's cooperation, sized and visible before the credit is extended.

The operator needs the converse: the agent's key must not be able to commit the operator to unbounded spending, whether through a stolen key, an injected instruction or an arithmetic slip.

1.3 What Bonded is

Bonded is one Anchor program plus the services around it. Two kinds of account carry the design.

The **policy vault** (`Vault`) holds the USDC of one agent. It is a program-derived account whose associated token account holds the funds, so only the program can move them. The agent holds a *session key*, and the operator holds the *admin key*. The vault's policy sets a float cap, a spending window and its budget, an optional co-signer with a co-sign threshold, and a guardian that can freeze the vault. Per-merchant rules (`MerchantRule`) add a per-transaction cap, a per-merchant window budget and a quote requirement. USDC on Solana is a classic SPL Token mint, so Token-2022 transfer hooks are not available for it [32]; every check lives in the program that owns the vault.

The **bond** (`Bond`) holds USDC the operator posts. Against it the operator opens credit lines (`CreditLine`) for merchants it chooses. Each line has a limit, the sum of open limits is the bond's `reserved_lines`, and the bond's balance divided by that sum is its *coverage ratio*. Coverage is public, and the program refuses any line,

limit increase or withdrawal that would take it below the operator's minimum, which is at least 3,300 bps (33%). The bond is slashable. When a merchant settles a voucher and the vault cannot pay, the program opens a default event. Every line holder may file its latest voucher within 72 hours, and the bond then pays each claim in full, or pro rata when the claims exceed the bond.

Agents pay through three lanes, each protected by a different fact:

- **Float.** Open-world x402 `exact` calls. Because the `exact` scheme requires a top-level transfer [7] and a program-owned vault can only sign through the program, the vault delegates a bounded SPL allowance, at most the float cap, to the session key, which signs the transfer as delegate. `refill_float` restores the allowance and books what was spent against the window budget. The loss bound is the allowance outstanding.
- **Tab.** Metered services billed in arrears. The operator opens a line for a merchant. With each request the agent signs a 96-byte voucher carrying the cumulative amount owed, and the merchant verifies it locally. Later the merchant calls `settle_tab` with the latest voucher and the vault pays the difference since the last settlement. If the vault is short, the bond pays.
- **Escrow.** Jobs with delayed delivery. The agent funds an escrow with `pay_escrow` against a merchant-signed quote that names the amount, a specification hash, a delivery deadline and an optional arbiter. The merchant marks delivery at or before the deadline; the payer accepts, or a 48-hour dispute window lapses and the funds release. Without a delivery mark by the deadline, anyone can call `refund_on_timeout`.

The program stands behind two obligations, because each rests on a fact it can check without judgment: a voucher the agent's key signed, and the absence of a delivery mark by a deadline. Every other spending decision stays inside the policy, which bounds it in USDC base units before the money moves.

$$\text{unsettled}(\text{line}) = \text{voucher.cumulative} - \text{settled_floor} \leq \text{line.limit}$$
$$\text{eligible} = \min(\max(\text{voucher.cumulative} - \text{settled_floor}, 0), \text{line_limit_snapshot})$$
$$\text{payout} = \text{eligible} \text{ when } \Sigma \text{ eligible} \leq \text{pool}$$
$$\text{payout} = \lfloor \text{pool} \times \text{eligible} \div \Sigma \text{ eligible} \rfloor \text{ when } \Sigma \text{ eligible} > \text{pool}$$

<code>voucher.cumulative</code>	the amount in the merchant's latest signed voucher
<code>settled_floor</code>	the larger of the line's settled and retired cumulative amounts
<code>line_limit_snapshot</code>	the line limit the claim records when first filed
<code>pool</code>	the bond balance recorded at the first payout of a default event

Example. An operator posts 1,000 USDC to its bond and opens a 250 USDC line for an inference merchant. Reserved lines are 250 USDC and coverage is 40,000 bps (400%). Before serving a single request, the merchant knows it can be owed at most 250 USDC beyond what has settled, and that a 1,000 USDC bond stands behind that amount, shared with any other line the operator opens. If the operator opens three more 250 USDC lines, coverage falls to 10,000 bps. With a minimum of 10,000 bps, the program refuses a fifth.

Bonded's token is \$BONDED, with a fixed supply of 1,000,000,000, all of it issued in one public launch on equal terms; every fee the protocol keeps is for buying \$BONDED on the market and burning it (\$10.7). Bonds, payments and claims stay in USDC. The two fees are paid in USDC: a tab settlement fee in basis points of each settled amount, deducted from what the merchant receives, and an escrow fee reserved on top of the job's principal and earned only when the escrow releases to the merchant. The devnet parameter file sets them at 10 bps, waived, and 25 bps.

1.4 What this paper covers

Part	What it covers
§2 System overview	The actors and what each key can do, the three lanes compared, the accounts and how they relate, the life of a tab, and where each component runs.
§3 Policy vaults	The vault and its policy, window accounting, merchant rules, payment intents, approvals by Ed25519 and passkey co-signers, the guardian and freeze.
§4 The float	The delegated allowance, <code>refill_float</code> , and how the float shares the window budget with program payments.
§5 Tabs and vouchers	The <code>bonded-tab</code> scheme, the 96-byte voucher, merchant-side verification, <code>settle_tab</code> , and voucher-key rotation on a line.
§6 Bonds and credit lines	Posting collateral, opening, resizing and renewing lines, coverage, and bond withdrawals with their cooldown.
§7 Defaults and claims	How a default event opens, filing and refiling claims, the payout snapshot, the pro-rata rule, and freeze recourse.
§8 Escrow	Quotes, funding, delivery marks, the dispute window, arbiters, and every terminal path.
§9 Program architecture	The account model, instruction groups, signature verification through the instructions sysvar, and committed events.
§10 Fees and settlement	The tab and escrow fees, the fee treasury, the size and compute cost of settlement transactions, and \$BONDED: its supply, launch and the fees that buy and burn it.
§11 Off-chain services	The API, indexer, keeper, guardian worker, merchant middleware, agent SDK and MCP server.
§12 Security design	What each key can do, which checks run in each instruction, and the invariants the program keeps.
§13 Infrastructure and deployment	The services on Railway, the program's deploy and initialization, configuration, and how a deployment is checked.
§14 Parameters	Every program constant and every value set at initialization.
Appendix A Instruction reference	Arguments, accounts and signers of every instruction.
Appendix B Account layouts	Field order, types and sizes of every account.
Appendix C Error codes	Every program error with its number and the check that raises it.
Appendix D Glossary	Exact definitions of the product's terms.
Appendix E References	The sources cited in brackets.

1.5 Notation

The paper uses the program's own units, encodings and names.

Notation	Meaning
Amounts	USDC base units, stored as <code>u64</code> . USDC has 6 decimals, so 1 USDC = 1,000,000 base units. Amounts appear in base units where the program uses them, with the USDC value alongside: 25,000,000,000 base units (25,000 USDC).
bps	Basis points. 1 bps = 1/10,000, so 10,000 bps = 100%. A rate applied to an amount is $[\text{amount} \times \text{bps} \div 10,000]$.
Arithmetic	Checked <code>u64</code> arithmetic. Products are taken in <code>u128</code> before division, and division rounds down ($\lfloor \rfloor$).
Time	Unix seconds. Account timestamps are <code>i64</code> ; expiries inside signed messages are <code>u64</code> . Durations are given in seconds with hours alongside: 259,200 s (72 h). "Before t" means strictly less than t; "at or after t" means greater than or equal to t.
Byte order	Integers are little-endian in account data, instruction data and signed messages. Byte offsets count from 0, and layout figures give each field's offset and length.
Account data	Borsh encoding. Every program account starts with an 8-byte discriminator, the first 8 bytes of SHA-256(<code>account:<Name></code>), then a 1-byte PDA bump. An <code>Option</code> is a 1-byte tag followed by its payload. Instruction data starts with the first 8 bytes of SHA-256(<code>global:<name></code>).
Hashes and domains	SHA-256 throughout. A signed message starts with an 8-byte domain tag, the first 8 bytes of SHA-256(prefix $\parallel$ program ID $\parallel$ cluster genesis hash), where the prefix is a NUL-terminated string such as <code>Bonded voucher v2\0</code> . $\parallel$ is byte concatenation.
PDA seeds	Written as a list, for example <code>[b"line", bond, merchant]</code> : a byte-string literal followed by 32-byte addresses or raw byte arrays, in order. Integer seeds are 8-byte little-endian, as in <code>[b"default", bond, sequence]</code> . The account stores its canonical bump.
Names	Instructions, accounts, fields and seeds appear in code font exactly as in the program: account types in PascalCase (<code>CreditLine</code>), instructions and fields in snake_case (<code>settle_tab</code> , <code>reserved_lines</code>).
Constants and configuration	A constant is compiled into the program, such as <code>CLAIM_WINDOW = 259,200 s</code> . A configuration value is set by <code>initialize_protocol</code> or a governed instruction and stored in <code>ProtocolConfig</code> ; the paper gives the value from the devnet parameter file <code>backend/config/protocol.devnet.json</code> .
Examples	Worked examples are marked <i>Example</i> and use example values, not parameters.

2. System overview

This chapter describes the whole system before later chapters take each part apart: the keys and what each one can do, the three lanes an agent pays through, the accounts the program keeps and how they relate, the life of a tab from setup to payout, and where each component runs.

2.1 Actors and keys

Bonded divides authority by key. Each key can do a short, enumerable list of things, and every instruction checks its signer against the key stored in the account it touches. A key that leaks exposes only its own list. Table 2.1 gives each actor, the key it holds, and the instructions that key can sign.

Actor	Key	What the key can do	Instructions it signs
Operator	Admin key, stored as <code>Operator.admin</code>	Creates its operator account and bond, its vaults and their policies; sets merchant rules; posts collateral; opens, resizes and renews lines; rotates session, line and guardian keys; withdraws vault funds; requests and finalizes bond withdrawals; unfreezes.	<code>create_operator</code> , <code>create_vault</code> , <code>set_policy</code> , <code>set_merchant_rule</code> , <code>set_passkey_credential</code> , <code>rotate_session_key</code> , <code>withdraw</code> , <code>post_bond</code> , <code>open_line</code> , <code>resize_line</code> , <code>renew_line</code> , <code>rotate_line_key</code> , <code>request_withdrawal</code> , <code>finalize_withdrawal</code> , <code>rotate_operator_guardian</code> , <code>rotate_vault_guardian</code> , <code>fund_freeze_reserve</code> , <code>unfreeze</code> ;also <code>freeze</code> , <code>accept</code> , <code>dispute</code>
Agent	Session key, stored as <code>policy.session_key</code>	Spends the float as SPL delegate; makes program payments and funds escrow within policy; signs vouchers as a line's voucher key; accepts or disputes its escrow jobs.	Top-level <code>TransferChecked</code> , <code>pay</code> , <code>pay_escrow</code> , <code>accept</code> , <code>dispute</code> ; vouchers (off-chain messages)
Co-signer	Ed25519 key, or a <code>PasskeyCredential</code> for a P-256 passkey, stored as <code>policy.cosigner</code>	Approves one exact payment intent above the co-sign threshold.	The 128-byte approval message, verified by <code>approve_payment</code> or <code>approve_payment_passkey</code>
Merchant	Merchant key, the line's <code>merchant</code>	Settles its line, files claims in a default, attests outgoing-key vouchers, marks escrow delivery.	<code>settle_tab</code> , <code>file_claim</code> , <code>attest_line_voucher</code> , <code>mark_delivered</code> , <code>close_claim</code> , <code>close_default_placeholder</code>
Guardian	<code>Operator.guardian</code> or <code>policy.guardian</code>	Freezes a vault.	<code>freeze</code>
Keeper	Any key	Runs permissionless instructions whose outcome the program fixes from state and time.	<code>refill_float</code> , <code>execute_default_payout</code> , <code>finalize_empty_default</code> , <code>refund_on_timeout</code> , <code>release_after_window</code> , <code>refund_after_resolution_deadline</code> , <code>close_approval</code> , <code>close_freeze_checkpoint</code>

Actor	Key	What the key can do	Instructions it signs
Arbiter	Key named in a merchant-signed quote	Resolves a disputed escrow, to release or refund, before its resolution deadline.	<code>resolve</code>
Protocol authority	The program's upgrade authority	Initializes the protocol; sets the tab fee within its fixed maximum; configures the escrow fee and resolution duration; sets the \$BONDED mint, the buyback crank and its minimum.	<code>initialize_protocol</code> , <code>update_protocol</code> , <code>configure_escrow_fee</code> , <code>configure_escrow_resolution</code> , <code>configure_buyback</code>

Operator

The operator is the company running the agent. Its admin key creates the `Operator` account at `[b"operator", admin]` , and the same instruction, `create_operator` , creates the operator's one `Bond` at `[b"bond", operator]` with its USDC token account and the operator's `minimum_coverage_bps` , which must be at least 3,300. The admin key is any Solana signer; the program checks only its signature, so a multisig that signs as one address can hold it.

Every admin power that could let an operator leave ahead of its obligations is delayed. A line decrease takes effect 72 hours after it is scheduled (`DECREASE_DELAY` = 259,200 s). A guardian replacement takes effect after the same 72 hours. A bond withdrawal unlocks only after the longest line expiry, plus the 72-hour claim window, plus a configured margin. The operator's `tabs_settled` , `defaults_count` and `created_at` fields are public, and anyone can read them.

Agent

The agent holds one key, the session key. It is the only key in the agent's runtime, and the policy bounds everything it can do: the float allowance, the window budget, the merchant rules and the co-sign threshold. When the operator opens a line, the program copies the vault's current session key into the line's `voucher_session_key` ; that key signs the line's vouchers until a delayed `rotate_line_key` replaces it. The session key cannot change policy, open lines, withdraw funds or unfreeze a vault.

Co-signer

The co-signer is a human key that approves large payments. `policy.cosigner` is either zero (no co-signer), an Ed25519 public key, or the address of a `PasskeyCredential` that holds a P-256 passkey. When a program payment exceeds `cosign_threshold` , `pay` and `pay_escrow` require an unused, unexpired `Approval` that binds the exact payment intent, merchant and amount to the vault's current policy revision. The approval is created in its own transaction, so a `WebAuthn` assertion, which carries authenticator data and client JSON, never has to share a transaction with a merchant quote and the payment.

Merchant

The merchant holds its own key and signs only for its own lines and jobs. Opening a line does not need the merchant's signature, because the line commits only the operator. A merchant decides whether to extend credit by reading the line, the bond and its coverage, and the operator's counters, all of which are public account state.

Guardian

A guardian can freeze a vault and do nothing else. Freezing revokes the float delegate, stops every payment from the vault and records a `FreezeCheckpoint` whose freeze time cannot change; it moves no funds, and only the admin can unfreeze. Each vault answers to two guardians: the operator-level `Operator.guardian` and the vault-level `policy.guardian`. The admin replaces either through `rotate_operator_guardian` or `rotate_vault_guardian` with 72 hours' notice, and `set_policy` cannot change a guardian directly. The operator can pre-fund the vault's `freeze-reserve` account so that the checkpoint's rent comes from the reserve rather than from the guardian.

Keeper

The keeper is anyone. Every keeper instruction has an outcome fixed by account state and the clock, such as a payout computed from a default's snapshot or a refund that falls due after a deadline. The signer's identity changes nothing, so any party can run the cranks and any other party can replace it.

Arbiter

An arbiter exists only for an escrow job whose quote names one. The quote's `arbiter` field is zero when there is none, and the program refuses a quote that names its own merchant. The arbiter can act only on a disputed job and only before `resolve_until`; after that deadline anyone can refund the job.

Protocol authority

The protocol authority is the program's loader-v3 upgrade authority. `initialize_protocol` and every governed update check that the program's `ProgramData` names the signer as its upgrade authority. Fee rates are bounded by maxima that are fixed once set: the tab fee maximum at initialization and the escrow fee maximum at its first configuration. No instruction gives this key access to vault, bond or escrow balances, or to the fee treasury: the fees leave only through the crank's buyback, which burns the \$BONDED it buys (\$10.7).

2.2 The three lanes

An agent pays through three lanes. Each lane exists because a different kind of payment rests on a different checkable fact, and the lane determines which account pays the merchant and what bounds the operator's loss.

	Float	Tab	Escrow
For	Open-world x402 <code>exact</code> calls to any recipient	Metered services billed in arrears by a merchant holding a line	Jobs with delayed delivery
Enforcement	SPL delegate allowance on the vault token account, at most <code>float_cap</code> , refilled within the window budget by <code>refill_float</code>	A <code>CreditLine</code> the admin opened; vouchers signed by the line's voucher key; each settlement at most <code>limit</code>	A 169-byte merchant-signed quote bound to vault, merchant, amount, lane and expiry; merchant rule caps and budgets; approval above threshold
What bounds the loss	The outstanding allowance, at most <code>float_cap</code> , and at most <code>window_budget</code> per window	Per merchant, <code>limit</code> beyond the settled floor, only for merchants the admin chose	The quoted principal plus the fee, refunded if no delivery mark arrives by <code>deliver_by</code>

	Float	Tab	Escrow
What pays the merchant	The transfer itself, final when made	The vault at <code>settle_tab</code> ; the bond through claims if the vault is short	The escrow account, on acceptance, after the dispute window, or by the arbiter
When the merchant is paid	Before serving	In arrears, when it settles	On release
Instructions	<code>refill_float</code> ; a top-level <code>TransferChecked</code> signed by the session key	<code>open_line</code> , <code>settle_tab</code> , <code>file_claim</code> , <code>execute_default_payout</code>	<code>pay_escrow</code> , <code>mark_delivered</code> , <code>accept</code> , <code>dispute</code> , <code>resolve</code> , <code>refund_on_timeout</code> , <code>release_after_window</code> , <code>refund_after_resolution_deadline</code>

Float

The Solana `exact` scheme verifies a top-level `TransferChecked` [7]. A vault is a program-derived address, and a PDA signs only through a cross-program invocation from its program, which produces an inner instruction rather than a top-level one. The float resolves this: the vault grants the session key an SPL delegate allowance, and the session key signs the top-level transfer as the delegate. The Bonded program is not part of that transaction, so merchant rules cannot gate it; the allowance itself is the bound.

`refill_float` is permissionless. It reads the token account's real `delegated_amount` , books what the session key consumed since the last approval into the window, and approves a new allowance.

$$\begin{aligned} \text{consumed} &= \text{last_approved} - \text{delegated_amount} \\ \text{window_spent} &\leftarrow \text{window_spent} + \text{consumed} \\ \text{allowance} &= \min(\text{float_cap}, \text{window_budget} - \text{window_spent}) \end{aligned}$$

<code>last_approved</code>	the allowance approved at the previous refill
<code>delegated_amount</code>	the allowance still unspent on the vault token account
<code>window_spent</code>	spending booked in the current window

The window is anchored at `window_started_at` and advances in whole periods of `window_seconds` . Because a refill books consumption exactly once and never approves more than the budget left in the current window, booked float spending never exceeds `window_budget` in a window, and the session key never holds an allowance above `float_cap` , however often anyone refills.

Tab

On a tab the merchant extends credit. With each request the agent signs a voucher that carries the cumulative amount owed on the line, and each voucher replaces the previous one; settlement needs only the latest.

`settle_tab` requires the Ed25519 verification instruction immediately before it, checks the voucher's domain tag, line address, signer, nonce, cumulative amount and expiry, and pays the difference between the voucher's

cumulative amount and the line's settled floor. That difference may not exceed the line's limit. When the vault holds less than the difference, `settle_tab` transfers nothing and opens a default event instead, and the bond pays the merchants' claims after the claim window.

Escrow

On an escrow job the protecting fact is time. The merchant signs a quote naming `deliver_by`; the agent funds the escrow with principal plus fee; the merchant must call `mark_delivered` at or before `deliver_by`. Without that mark, `refund_on_timeout` returns everything to the vault after the deadline. With it, a 48-hour dispute window opens (`DISPUTE_WINDOW_SECONDS = 172,800 s`), after which the funds release to the merchant. Within the window the payer may dispute, but only when the quote named an arbiter; the arbiter then releases or refunds before the job's resolution deadline, and after that deadline anyone can refund it.

Program payments

Beside the three lanes, `pay` sends USDC from the vault to an allowlisted merchant through the program, final at once. It runs the same checks as escrow funding: the merchant's `MerchantRule`, its per-transaction cap and window budget, the vault's window budget, an optional or required quote, and an approval above the co-sign threshold. The outstanding float allowance counts against the vault's window budget before any program payment, so the float and program payments together never exceed `window_budget` in one window.

$$\text{window_spent} + \text{amount} + \text{delegated_amount} \leq \text{window_budget}$$

$$\text{rule.window_spent} + \text{amount} \leq \text{rule.window_budget}$$

$$\text{amount} \leq \text{rule.per_transaction_cap}$$

`amount` the total debit of the program payment, principal plus any escrow fee

`rule` the vault's `MerchantRule` for this merchant

2.3 Accounts at a glance

All Bonded state lives in program-derived accounts of one program. Each account's address is derived from fixed seeds, so any party can compute where an account must be and the program can reject a substitute. Every program account starts with an 8-byte discriminator and a 1-byte bump.

Account	Seeds	Holds	Opened by	Closed by
ProtocolConfig	<code>[b"protocol"]</code>	Authority, USDC mint, fee treasury, tab fee, waiver and maximum, escrow fee and maximum, resolution duration, withdrawal margin, freeze grace, cluster genesis hash, voucher domain, total bond balance	<code>initialize_protocol</code>	Stays open
Fee treasury (SPL token account)	<code>[b"fees"]</code>	Collected tab and escrow fees, under ProtocolConfig authority	<code>initialize_protocol</code>	Stays open

Account	Seeds	Holds	Opened by	Closed by
Operator	[b"operator", admin]	Admin, guardian, creation time, tabs_settled , defaults_count , frozen vault count, recourse lock	create_oper ator	Stays open
Bond	[b"bond", operator]	Balance, reserved_lines , minimum_coverage_bps , longest line expiry, withdrawal request, default sequence	create_oper ator	Stays open
Vault	[b"vault", operator, agent_id]	Token account, policy, window counters, freeze state, policy revision	create_vaul t	Stays open
MerchantRule	[b"rule", vault, merchant]	Per-transaction cap, window budget and spent, quote requirement	set_merchan t_rule	Stays open
Approval	[b"approval", vault, intent_hash]	Merchant, amount, expiry, used flag, policy revision, rent payer	approve_pay ment , approve_pay ment_passke y	Consumed by pay or pay_escrow ;Or close_approval
PasskeyCred ential	[b"passkey", vault, credential_id_ hash]	Compressed P-256 key, RP ID hash, allowed origin hash, sign count	set_passkey _credential	Stays open
PaymentRec eipt	[b"payment", vault, intent_hash]	Intent and quote hashes, amount, total debit, nonce, lane, outcome	pay , pay_escrow	Stays open as a replay guard
QuoteRecei pt	[b"quote", vault, quote_hash]	Used flag and the payment receipt that used the quote	pay with a quote, pay_escrow	Stays open as a replay guard
CreditLin e	[b"line", bond, merchant]	Vault, voucher keys, limit and pending decrease, settled and retired cumulative, maximum nonce, expiry, tab fee	open_line	Stays open; renewed by renew_line
VoucherAtt estation	[b"voucher- attestation", line, outgoing_key, effective_at]	Acknowledged maximum nonce and cumulative amount of an outgoing voucher key	attest_line _voucher	Stays open
DefaultEve nt	[b"default", bond, sequence]	Opening time, window end, claims total and count, payout snapshot, total paid	settle_tab	A healthy settlement closes its placeholder at once; a real default stays open as history
Claim	[b"claim", default_event, merchant]	Latest voucher amount, nonce and expiry, settled and limit snapshots, eligible and paid amounts	file_claim	execute_default_payo ut , Or close_claim
FreezeChec kpoint	[b"freeze", vault, generation]	Freeze time, unfreeze time, recourse end, episode, defaults count at freeze	freeze	close_freeze_checkpo int

Account	Seeds	Holds	Opened by	Closed by
Freeze reserve (system account)	<code>[b"freeze-reserve", vault]</code>	Lamports for checkpoint rent	<code>fund_freeze_reserve</code>	Stays open
Escrow	<code>[b"escrow", vault, quote_hash]</code>	Principal, fee, specification hash, deadlines, deliverable hash, arbiter, state	<code>pay_escrow</code>	Every terminal escrow instruction

The vault, the bond and each escrow hold their USDC in classic SPL associated token accounts whose authority is the PDA itself. The protocol's mint must have 6 decimals, and every token account must belong to the classic SPL Token program; Token-2022 accounts are refused.

The cardinalities follow from the seeds. One admin key has one `Operator`, and one operator has one `Bond`. An operator has any number of vaults, one per 32-byte `agent_id` (the TypeScript builder uses the SHA-256 of a UTF-8 agent ID). A bond backs any number of lines, at most one per merchant, and each line names the single vault whose USDC pays its tab. Merchant rules gate program payments and escrow from a vault; they do not gate tabs, which a line governs. A bond has at most one active default at a time, recorded in `active_default_sequence`, and each default holds at most one claim per merchant.

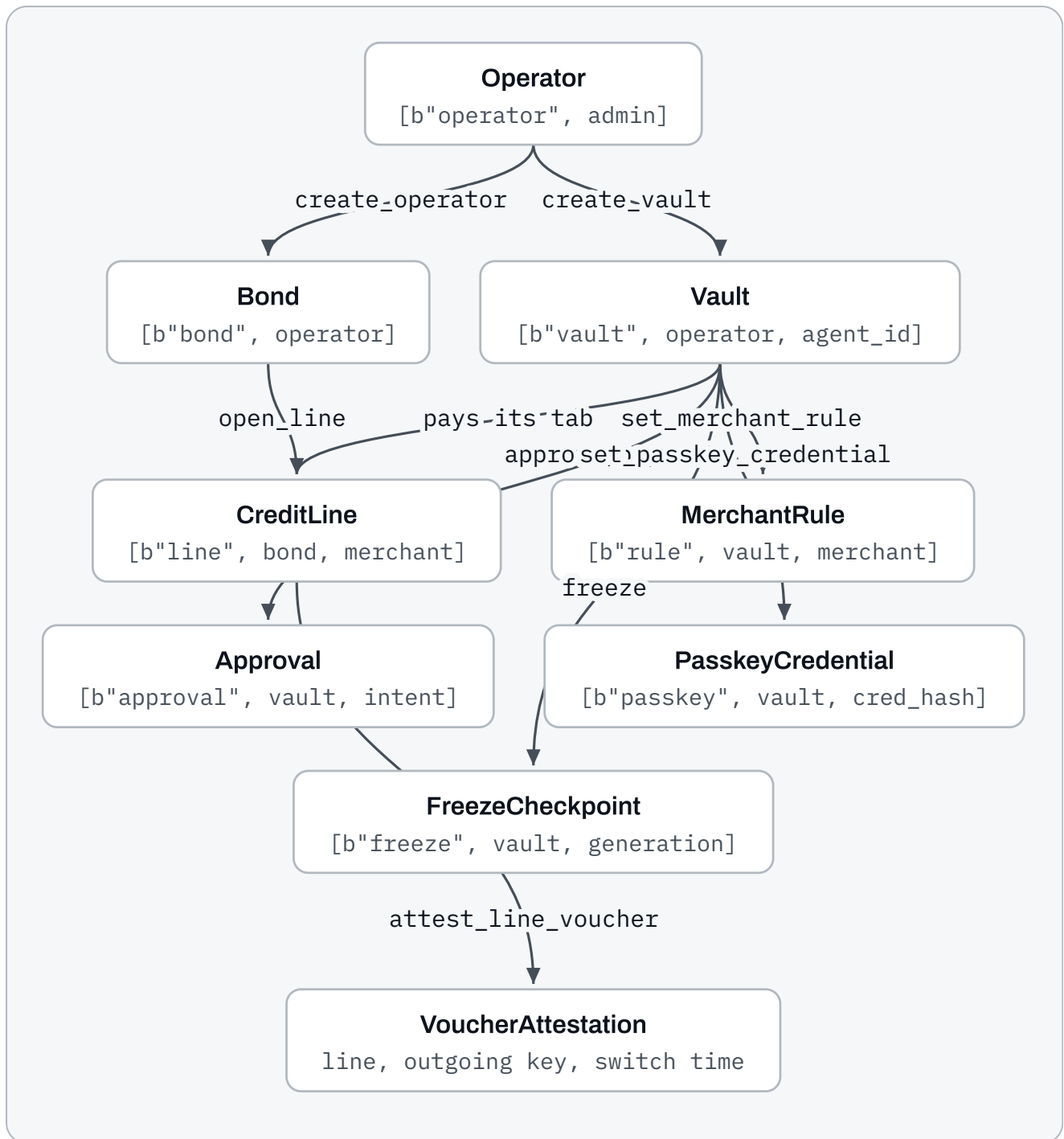


Figure 2. The accounts behind one operator: its bond and lines, and each agent's vault with its policy accounts.

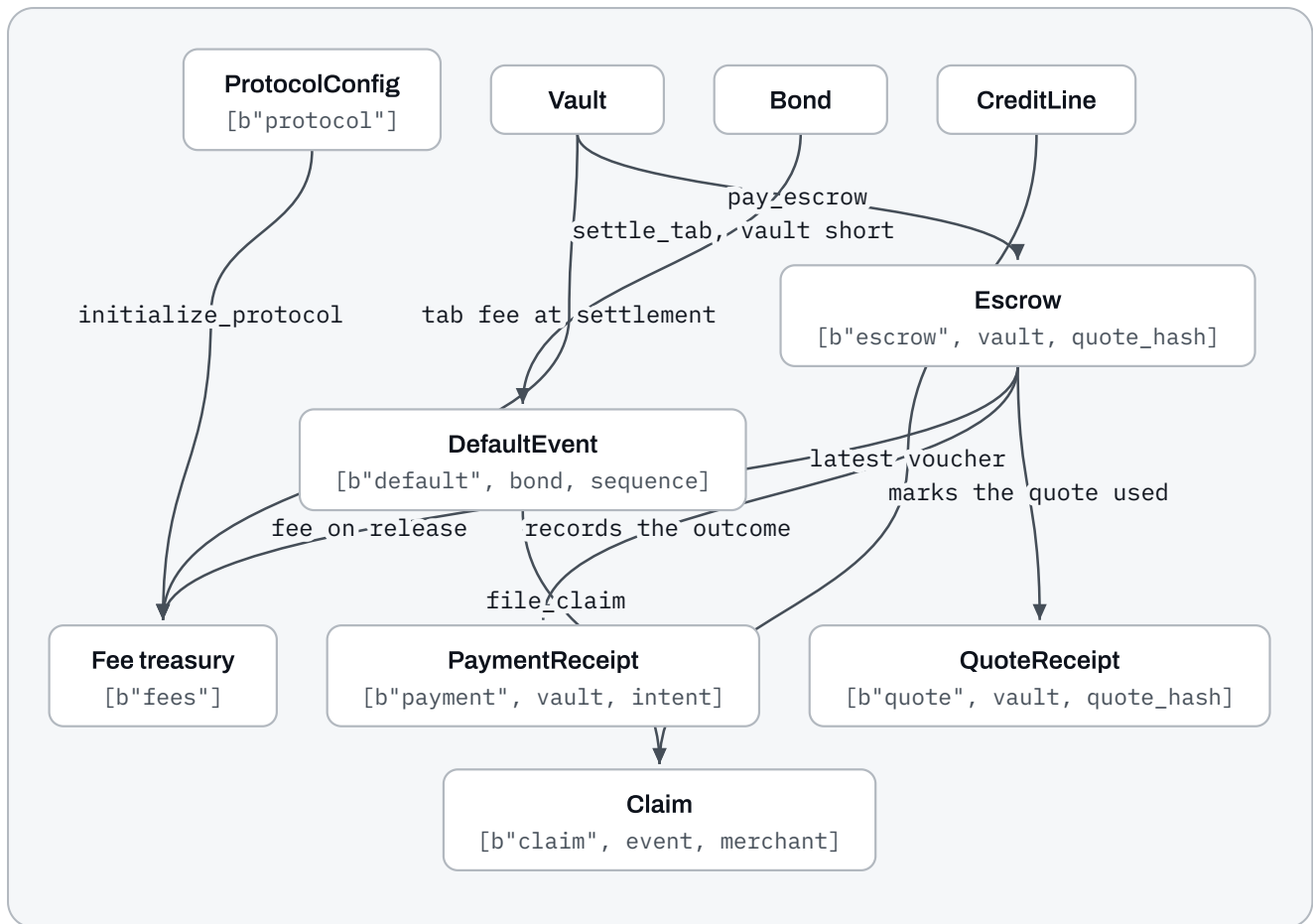


Figure 3. The accounts that settlement, defaults and escrow create, and the instruction that creates or feeds each one.

2.4 The life of a tab

A tab runs in four stages: the operator sets up, the agent buys on credit, the merchant settles, and, if the vault is short, the bond pays the line holders' claims.

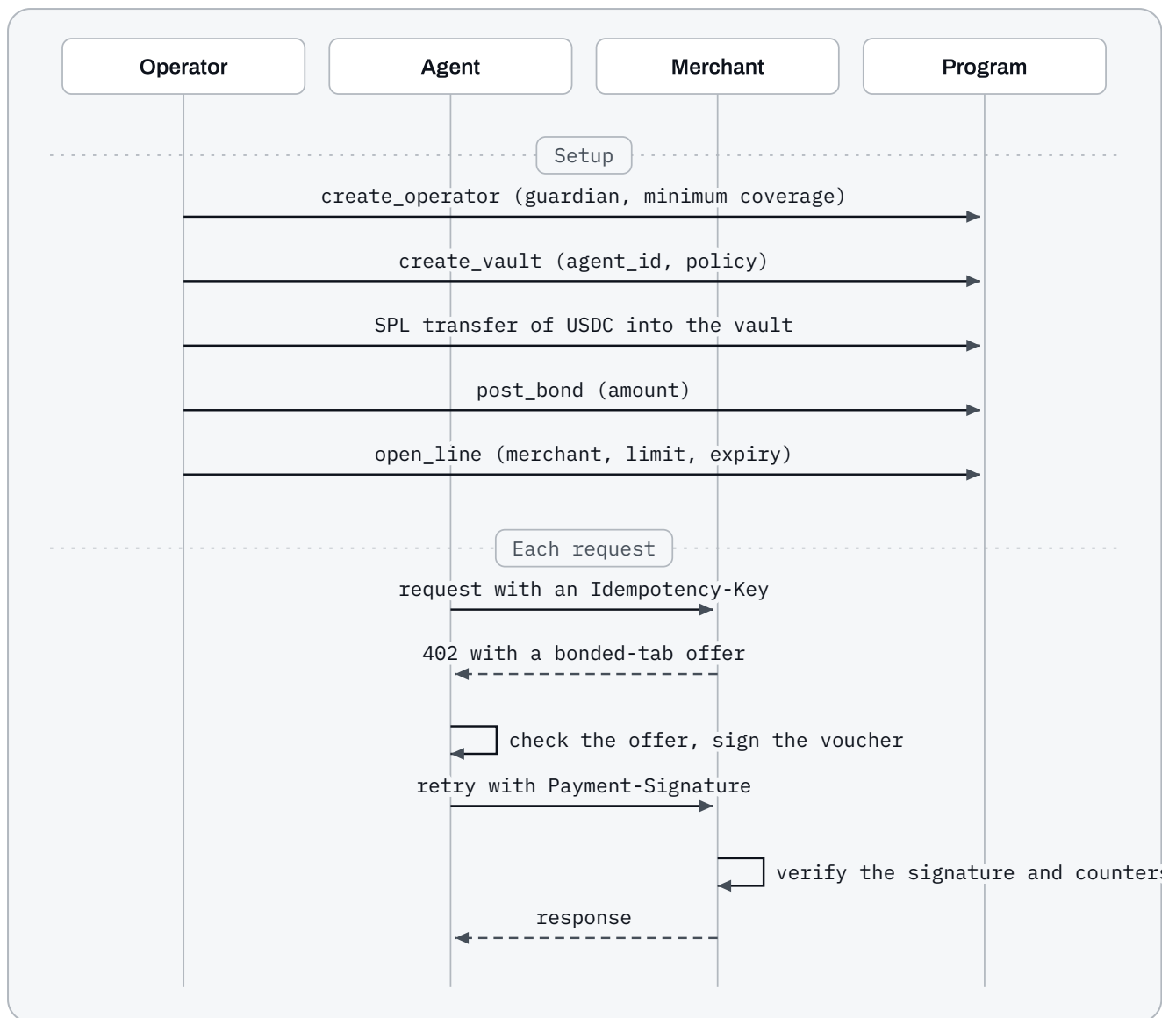


Figure 4. An operator sets up a vault, a bond and a line, and the agent then buys on the tab.

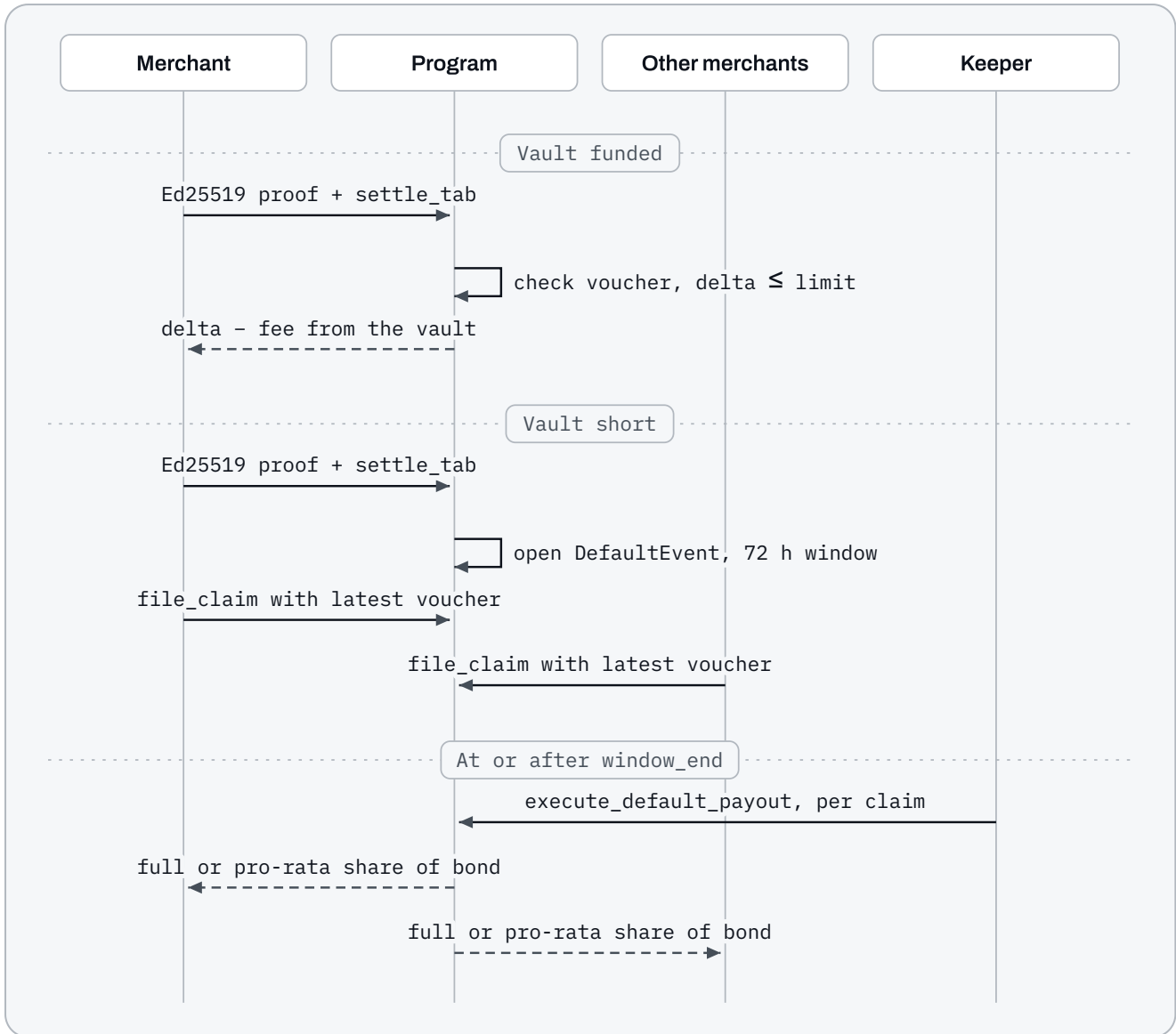


Figure 5. The merchant settles its latest voucher; a short vault opens a default event that the bond pays.

Setup

1. `create_operator(guardian, minimum_coverage_bps)` creates the `Operator`, the `Bond` and the bond's token account. It refuses a minimum below 3,300 bps.
2. `create_vault(agent_id, policy)` creates the vault and its token account. The policy must name a non-zero session key, a positive `window_seconds`, and a `float_cap` no greater than `window_budget`.
3. The operator funds the vault with an ordinary SPL `TransferChecked` from its own USDC account into the vault's token account. The API prepares it as the `fund_vault` action; no program instruction is involved.
4. `post_bond(amount)` moves the operator's USDC into the bond. The program refuses a bond balance above 25,000,000,000 base units (25,000 USDC) and a protocol-wide total above 250,000,000,000 base units (250,000 USDC) (`BOND_CAP`, `TOTAL_CAP`).
5. `open_line(limit, expires_at)` opens a line for one merchant. The vault must not be frozen and the bond must have no active default. The program adds `limit` to `reserved_lines` and checks coverage against the collateral not already promised to a pending withdrawal. It records the line's tab fee, which is zero when the protocol waives fees, and copies the vault's session key as the line's voucher key.

$$(\text{balance} - \text{pending_withdrawal}) \times 10\,000 \geq (\text{reserved_lines} + \text{limit}) \times \text{minimum_coverage_bps}$$

$$\text{coverage_bps} = \lfloor \text{balance} \times 10\,000 \div \text{reserved_lines} \rfloor$$

<code>balance</code>	the bond's tracked USDC, base units
<code>pending_withdrawal</code>	the amount of a requested bond withdrawal, or zero
<code>reserved_lines</code>	the sum of the limits of the lines the bond backs
<code>minimum_coverage_bps</code>	the operator's minimum, at least 3 300

Service

Before serving, the merchant reads the line and the bond: the limit, expiry, voucher key, settled and retired cumulative amounts, maximum nonce, the vault's freeze state. Its middleware caches the line and re-reads it when an outstanding increment reaches a configured threshold; whether the bond's coverage is enough to serve on credit is the merchant's own call, read from the public bond.

On a request, the merchant middleware answers 402 with exactly one `bonded-tab` offer, in the JSON body and in a base64 `Payment-Required` header. The offer names the line, the request key, a request digest, the metered units, and the next voucher's cumulative amount, nonce, expiry and usage digest. The usage digest commits to the request's method, URL, body, request key, charge and units. The agent SDK checks the URL, method, network, mint, merchant, digests, counters and its maximum charge, signs the 96-byte voucher with the session key, and retries the same request with a `Payment-Signature` header. The merchant verifies the Ed25519 signature and checks that the cumulative amount and nonce advance its stored latest voucher and that the unsettled amount stays within the limit. It then serves and stores the response so that a retry with the same key is replayed and not billed twice.

Settlement

The merchant's worker settles one line per legacy transaction: an Ed25519 verification instruction whose data is exactly 208 bytes, followed immediately by `settle_tab(default_sequence, ed25519_ix_index)`. In the local benchmark, a bare single-voucher transaction measures 894 bytes and 61,044 compute units. The program finds the proof through the instructions sysvar and checks its header, signer and message byte for byte. It requires the voucher's nonce to exceed the line's `maximum_nonce`, its cumulative amount to exceed the line's settled floor, and its expiry to lie in the future and no later than the line's expiry. It then pays.

$$\text{settled_floor} = \max(\text{settled_cumulative}, \text{retired_cumulative})$$
$$\text{delta} = \text{voucher.cumulative} - \text{settled_floor}$$
$$\text{delta} \leq \text{limit}$$
$$\text{fee} = \lfloor \text{delta} \times \text{line.tab_fee_bps} \div 10\,000 \rfloor$$
$$\text{merchant_amount} = \text{delta} - \text{fee}$$

<code>settled_floor</code>	the cumulative amount already settled or retired by a default
<code>limit</code>	the line's applied limit
<code>merchant_amount</code>	moved from the vault to the merchant's token account
<code>fee</code>	moved from the vault to the fee treasury

The line's `settled_cumulative` and `maximum_nonce` advance to the voucher's values, and the operator's `tabs_settled` counter increments.

Default, claims and payout

If the vault's token account holds less than `delta`, `settle_tab` transfers nothing. It opens the `DefaultEvent` at `[b"default", bond, default_sequence]`, where `default_sequence` is the bond's previous sequence plus one, sets `window_end` to the opening time plus 259,200 s (`CLAIM_WINDOW`), clears any pending bond withdrawal, and increments the operator's `defaults_count`. A frozen vault opens a default the same way once its frozen time in the current freeze episode reaches the configured freeze grace, which is at least 345,600 s (96 h).

While the window is open, each merchant whose line was opened or renewed since the bond's previous default files its latest voucher with `file_claim`, strictly before `window_end`. A merchant may refile a voucher with both a higher nonce and a higher cumulative amount, which replaces its earlier claim. At or after `window_end`, anyone calls `execute_default_payout` once per claim. The first call snapshots the bond's balance as the pool and the claims total as the denominator, so the order in which claims are paid does not change any amount. Each claim receives its eligible amount, or its floored pro-rata share when the claims exceed the pool, and closes. Its line's `retired_cumulative` and `maximum_nonce` advance to the claimed voucher's cumulative amount and nonce, so the same debt cannot be paid twice. The last payout clears the bond's active default. Only the bond pays; whatever the vault still holds stays in the vault.

Example. An operator posts 1,000 USDC and opens two lines with limits of 250 USDC each, one for an inference merchant A and one for a data merchant B. Coverage is $\lfloor 1,000 \times 10,000 \div 500 \rfloor = 20,000$ bps. The vault holds 300 USDC.

- A settles a voucher of 180 USDC cumulative. The delta is 180 USDC, the vault pays it, and 120 USDC remains. With the fee waived, A receives all of it; at 10 bps, A would receive 179.82 USDC and the fee treasury 0.18 USDC.
- A's vouchers reach 400 USDC cumulative, a delta of 220 USDC, within the 250 USDC limit. The vault holds 120 USDC, so `settle_tab` opens a default.

- A files 400 USDC: eligible = $\min(400 - 180, 250) = 220$ USDC. B, which never settled, files 240 USDC: eligible = $\min(240 - 0, 250) = 240$ USDC. Claims total 460 USDC, below the 1,000 USDC pool, so after the window both are paid in full and the bond falls to 540 USDC.
- Had the bond held 400 USDC (8,000 bps coverage, allowed under a 3,300 bps minimum), the pool would be 400,000,000 base units against 460,000,000 eligible. A would receive $[400,000,000 \times 220,000,000 \div 460,000,000] = 191,304,347$ base units and B $[400,000,000 \times 240,000,000 \div 460,000,000] = 208,695,652$. The remaining 1 base unit stays in the bond.

2.5 Where each piece runs

Bonded's on-chain part is one program. Everything else is off-chain software that prepares, signs, watches or cranks transactions, and each piece runs with the party whose key it uses.

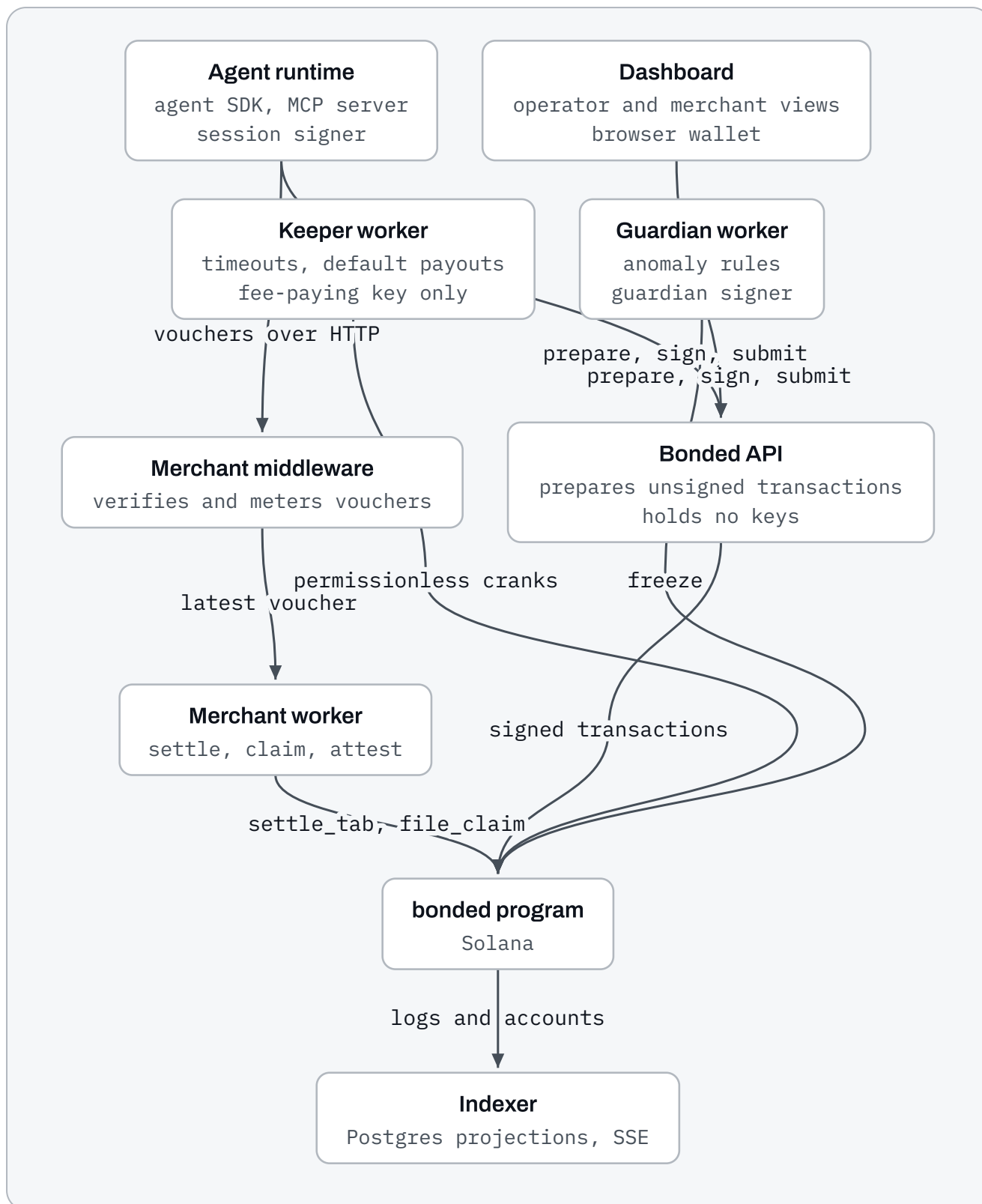


Figure 6. Where each component runs and what crosses between them.

Component	Runs at	Key it uses	What it does
bonded program	Solana	None; it verifies signers	Holds every rule in this paper and moves USDC between vault, bond, escrow, merchant and fee accounts

Component	Runs at	Key it uses	What it does
Bonded API	Backend service	None	Prepares unsigned legacy transactions with a simulation, checks signed bytes before broadcast, serves reads and server-sent events; never loads an admin, guardian, merchant, sponsor or session key
Indexer	Backend service, with Postgres	None	Reads successful program transactions and accounts over RPC, writes projections, deduplicated events and a cursor in one database commit, and publishes committed events
Keeper worker	Backend worker	A fee-paying key	Discovers due escrow timeouts, dispute-window releases, resolution-deadline refunds and default payouts, and submits them with durable per-job retries
Guardian worker	Operator	The operator's guardian signer and a fee sponsor	Evaluates burst and spend-spike rules over committed payments and freezes a vault when one fires
Merchant middleware	Merchant's server (Workers, Hono or Express)	None	Answers 402 with <code>bonded-tab</code> offers, verifies vouchers against cached line state, meters usage and replays responses by idempotency key
Merchant worker	Merchant	Merchant key	Settles one line per transaction, files claims in open defaults and submits <code>attest_line_voucher</code> during a line-key rotation notice
Agent SDK	Agent runtime	Session key, local or through a remote-signer transport the host supplies (tagged Turnkey, Crossmint or Coinbase)	Signs vouchers, refills the float, pays and funds escrow; pins the program ID and cluster genesis hash and rebuilds every instruction locally before signing
MCP server	Agent runtime, over stdio	Session key through the SDK	Exposes the SDK as tools with operator-set per-call and per-window ceilings that the model cannot raise
Dashboard	Web app at <code>/app</code>	The user's browser wallet	Shows operators their vaults, bonds and lines, and merchants their lines, coverage and claims; signs prepared transactions

The API prepares and the wallet signs. A prepared transaction contains unsigned legacy bytes, their hash, the blockhash, the signer roles and the result of simulating it. The holder of the right key signs exactly those bytes, and the API checks the canonical bytes, the message, the fee payer, every signature and the expiry before it broadcasts. The SDK does not trust the API's description of a transaction: it decodes the message, rebuilds each permitted instruction from its own inputs and its own RPC reads, and compares every account, privilege, byte and program ID before it signs.

Chain state is the source of truth for every service. The indexer commits projections, events and its cursor atomically, reprocesses idempotently, and reconciles projected accounts against the chain. The keeper judges deadlines by the indexed chain time, and the guardian evaluates only committed payment projections. The merchant's evidence, its stored vouchers and metered usage, stays in the merchant's own database. The chain shows settled amounts, and an unsettled voucher becomes public only when its merchant settles, files or attests it.

3. Policy vaults

The vault is the agent's account. It holds the USDC the agent spends and the policy that decides which of the agent's payments execute. The agent does not own those funds. It holds a session key, and the session key reaches the vault's USDC in three ways only: by signing `pay` or `pay_escrow`, which the program checks against the policy before it moves anything; by spending the float, a capped SPL Token allowance described in §4; and by signing vouchers on credit lines the admin opened, which merchants settle against the vault up to each line's limit (§5). Changing the policy, allowing a merchant, rotating a key and withdrawing all take the operator's admin key. The guardian key can freeze the vault and do nothing else.

The policy lives in the program because nothing else on the payment path can enforce it. USDC on Solana is a classic SPL Token mint, so Token-2022 transfer hooks are not available for it [32]. The only place a rule can run on every program payment is inside the program that owns the token account.

3.1 The vault account and its policy

Address and token account

A vault is a program-derived account at `[b"vault", operator, agent_id]`. `operator` is the address of the operator's `Operator` account, itself at `[b"operator", admin]`, and `agent_id` is a 32-byte seed; the TypeScript builder derives it as the SHA-256 of a UTF-8 agent ID. One admin key owns one `Operator` and any number of vaults, one per agent. The admin may be a Squads smart-account PDA, so an operator's existing multisig controls every vault it runs.

Each vault owns one token account: the classic SPL associated token account of the vault PDA for the protocol's USDC mint, `ProtocolConfig.usdc_mint`, which must have 6 decimals. Token-2022 accounts are refused. The owner of that token account is the vault PDA. A PDA has no private key, so the only signer that can act as owner of the vault's USDC is the program itself, and it signs only inside its own instructions, after their checks pass. Every outflow from the vault is therefore either a program transfer (`pay`, `pay_escrow`, `settle_tab`, `withdraw`) or a transfer by the SPL delegate the program installed, which is the float.

Fields

The vault's fields, in Borsh order:

Field	Type	Meaning
<code>bump</code>	<code>u8</code>	PDA bump
<code>operator</code>	<code>Pubkey</code>	The <code>Operator</code> account that owns the vault
<code>agent_id</code>	<code>[u8; 32]</code>	Seed that identifies the agent
<code>token_account</code>	<code>Pubkey</code>	The vault's USDC associated token account
<code>policy</code>	<code>PolicyArgs</code>	The spending policy (next table)
<code>window_started_at</code>	<code>i64</code>	Start of the current accounting window, Unix seconds
<code>window_spent</code>	<code>u64</code>	Spending booked against the current window, base units
<code>last_approved</code>	<code>u64</code>	The float allowance as of the last reconciliation, base units
<code>frozen</code>	<code>bool</code>	Whether the vault is frozen

Field	Type	Meaning
<code>frozen_at</code>	<code>i64</code>	Start of the current frozen interval
<code>recourse_until</code>	<code>i64</code>	End of the recourse tail after the last unfreeze
<code>freeze_generation</code>	<code>u64</code>	Number of freezes so far; names the latest <code>FreezeCheckpoint</code>
<code>policy_revision</code>	<code>u64</code>	Incremented by every <code>set_policy</code> ; every approval signs it
<code>pending_guardian</code>	<code>Option<PendingGuardian></code>	A scheduled vault-guardian change and the time it may apply
<code>freeze_episode_start_generation</code>	<code>u64</code>	First freeze generation of the current freeze episode
<code>freeze_episode_started_at</code>	<code>i64</code>	When that episode began
<code>episode_frozen_seconds</code>	<code>u64</code>	Completed frozen seconds in the episode

The policy is one embedded struct, `PolicyArgs` . The same struct is the argument of `create_vault` and of `set_policy` , so a policy is always written whole.

Field	Type	Meaning
<code>session_key</code>	<code>Pubkey</code>	The agent's key. Signs <code>pay</code> and <code>pay_escrow</code> , is the float delegate, and becomes the voucher key of each line opened from this vault
<code>float_cap</code>	<code>u64</code>	The largest allowance a refill may approve to the session key
<code>window_seconds</code>	<code>u32</code>	Length of the accounting window
<code>window_budget</code>	<code>u64</code>	The most that session-key spending may book in one window
<code>cosigner</code>	<code>Pubkey</code>	An Ed25519 co-signer key, or a <code>PasskeyCredential</code> address; zero for none
<code>cosign_threshold</code>	<code>u64</code>	Program payments strictly above this need an approval, when a co-signer is set
<code>guardian</code>	<code>Pubkey</code>	The vault-level guardian; zero for none

The policy separates the keys by what each may authorize. The session key acts on every payment but cannot change any limit. The co-signer adds a second signature above a threshold but cannot start a payment. The guardian stops payments but cannot start them or move funds. Only the admin changes limits, and two of its changes are themselves constrained: a guardian change waits 72 hours, and every policy change voids outstanding approvals (§3.2).



287 bytes · one cell per 4 bytes

Offset	Bytes	Field	Meaning
0	8	discriminator	SHA-256("account:Vault")[0..8]
8	1	bump	PDA bump
9	32	operator	Operator account address
41	32	agent_id	32-byte agent seed
73	32	token_account	the vault's USDC associated token account
105	32	policy.session_key	the agent's session key
137	8	policy.float_cap	u64, base units
145	4	policy.window_seconds	u32
149	8	policy.window_budget	u64, base units
157	32	policy.cosigner	Ed25519 key or credential PDA; zero for none
189	8	policy.cosign_threshold	u64, base units
197	32	policy.guardian	vault guardian; zero for none
229	8	window_started_at	i64, Unix seconds
237	8	window_spent	u64, base units
245	8	last_approved	u64, base units
253	1	frozen	bool
254	8	frozen_at	i64
262	8	recourse_until	i64
270	8	freeze_generation	u64
278	8	policy_revision	u64
286	1	pending_guardian	Option tag, 0 = None, 1 = Some

Figure 7. The fixed-offset prefix of the 351-byte Vault account, up to the pending_guardian option tag.

The account is allocated at its largest encoding, $8 + 343 = 351$ bytes. Borsh writes an `Option` compactly: when `pending_guardian` is `None` it takes one byte, and the three episode fields that follow it start 40 bytes earlier than when it is `Some` (a 32-byte key and an 8-byte time). The figure therefore stops at the option tag.

The vault's satellite accounts

A vault works with a small set of accounts derived from its own address. Rent is the rent-exempt minimum, $(\text{size} + 128) \times 3,480 \times 2$ lamports [28], and is returned when an account closes: to its rent payer, or for a checkpoint to the freeze reserve.

Account	Seeds	Allocated bytes	Rent, lamports	Lifetime
Vault	[b"vault", operator, agent_id]	351	3,333,840	Stays open
MerchantRule	[b"rule", vault, merchant]	106	1,628,640	Stays open; updated in place
Approval	[b"approval", vault, intent_hash]	162	2,018,400	Closes when consumed, expired or superseded
PaymentReceipt	[b"payment", vault, intent_hash]	211	2,359,440	Permanent replay guard
QuoteReceipt	[b"quote", vault, quote_hash]	138	1,851,360	Permanent replay guard
PasskeyCredential	[b"passkey", vault, credential_id_hash]	174	2,101,920	Stays open; immutable
FreezeCheckpoint	[b"freeze", vault, generation]	89	1,510,320	Closes to the freeze reserve after its claim windows
Freeze reserve	[b"freeze-reserve", vault]	0	—	System-owned lamport account

`PaymentReceipt` and `QuoteReceipt` are what make each payment intent and each merchant quote usable once. They outlive the approval and escrow accounts that refer to them, so a closed account can never be replayed into existence.

3.2 Creating a vault and setting policy

`create_vault`

`create_vault(agent_id, policy)` is signed by the admin, who pays rent. Its accounts are the admin (writable signer), the Operator at [b"operator", admin] with `has_one = admin`, `ProtocolConfig`, the USDC mint (it must equal `protocol.usdc_mint`), the new vault, and the vault's associated token account, created when absent; a correctly precreated ATA is accepted. The handler validates the policy:

$$\text{policy.session_key} \neq 0$$
$$\text{policy.window_seconds} > 0$$
$$\text{policy.float_cap} \leq \text{policy.window_budget}$$

0 the all-zero public key

A failed condition returns `InvalidConfiguration`. The third condition is what keeps the window invariant of §3.3: the float allowance is reserved against the window budget, and an allowance larger than the budget could never be fully reserved.

On success the vault has `window_started_at = now`, `window_spent = 0`, `last_approved = 0`, `policy_revision = 0` and no delegate. The session key receives its first allowance at the first `refill_float`. The program emits `CoreEvent` kind 2.

Funding needs no Bonded instruction: the operator sends USDC to the vault's token account with an ordinary SPL `TransferChecked`, which the API's `fund_vault` builder prepares and the indexer records as `vault_funded`.

`set_policy`

`set_policy(policy)` replaces the whole policy. The admin signs. The accounts are the admin, the `Operator`, the vault, the vault's token account, the token program and `ProtocolConfig`. The handler runs in this order:

1. Validate the new policy as for `create_vault`.
2. Require `policy.guardian` to equal the current guardian, otherwise `GuardianRotationRequired`.
3. Reconcile the window under the old policy (§3.3), booking float the session key has already spent.
4. Revoke the SPL delegate on the vault's token account and set `last_approved = 0`.
5. Increment `policy_revision`.
6. Install the new policy and emit `CoreEvent` kind 3.

Each step has its reason. Reconciliation comes before revocation so the window books exactly the float that was consumed; after `Revoke` the token account reports a `delegated_amount` of zero, and the unspent rest of the allowance lapses instead of being counted. The revocation exists because the outstanding allowance was granted under the old `float_cap` and possibly the old session key; the new policy starts with no delegate until a refill computes one from the new values. The revision bump voids every outstanding approval, because each approval signs the revision it was issued under (§3.5): a co-signer who approved a payment under one threshold, one session key and one budget has not approved it under another. The guardian is excluded because it is a commitment to whoever monitors the vault. Changing it goes through `rotate_vault_guardian` and its 72-hour notice (§3.7), and `set_policy` cannot shorten that.

`set_policy` keeps `window_started_at` and `window_spent` as reconciled. A new `window_seconds` takes effect at the next reconciliation, which counts whole periods of the new length from the stored start. A lower `window_budget` applies at once: if the current window has already booked more than the new budget, every payment and refill in that window returns `WindowBudgetExceeded` until the window rolls. `set_policy` has no frozen check, so an admin can tighten a frozen vault's policy before unfreezing it.

Who signs what

Every instruction that acts on a vault, its signer, and the `CoreEvent` kind it emits (the event codec the indexer reads, §9):

Instruction	Signer	Effect on the vault	Event
<code>create_vault</code>	admin	Creates the vault and its USDC ATA; the first window starts	kind 2

Instruction	Signer	Effect on the vault	Event
set_policy	admin	Reconcile, revoke the float, revision + 1, new policy	kind 3
set_merchant_rule	admin	Creates or updates a MerchantRule	kind 21
set_passkey_credential	admin	Registers a PasskeyCredential	—
refill_float	anyone; only a fee payer signs	Reconcile, Approve a new allowance	kind 20
pay	session key; any payer for receipt rent	Policy checks, TransferChecked to the merchant	kind 23, PaymentCommitted
pay_escrow	session key	Policy checks, principal plus fee into escrow (\$8)	PaymentCommitted , EscrowEvent
approve_payment	any payer; the co-signer's Ed25519 signature rides in the proof	Creates an Approval	kind 22, ApprovalCreated
approve_payment_passkey	any payer; the passkey assertion rides in the proof	Creates an Approval , advances the sign counter	kind 22, ApprovalCreated
close_approval	anyone	Closes an expired or superseded approval to its rent payer	ApprovalClosed
rotate_session_key	admin	Reconcile, revoke the float, new session key	kind 24
withdraw	admin	Reconcile, revoke the float, transfer out	kind 25
freeze	admin, operator guardian or vault guardian	Reconcile, revoke the float, write a checkpoint	kind 15
unfreeze	admin	Close the frozen interval, start the recourse tail	kind 16
rotate_vault_guardian	admin	Schedule or apply a guardian change	GuardianRotated
fund_freeze_reserve	admin	Move lamports into the freeze reserve	—
close_freeze_checkpoint	anyone	Return a spent checkpoint's rent to the reserve	—

3.3 Window accounting

The window

Session-key spending (the float, `pay` and `pay_escrow`) is counted against one budget per window. The window is fixed and anchored: its boundaries sit at the vault's creation time plus whole multiples of `window_seconds` , and traffic never moves them. A sliding window would need the history of recent payments

in the account; the anchored window needs two fields, `window_started_at` and `window_spent`, and constant compute per payment. Tab settlement does not draw on the window. A merchant decides when to call `settle_tab`, and tab exposure is bounded per line by its limit instead (§5, §6).

The program cannot see a float payment when it happens, because a delegate transfer is an SPL Token instruction that never invokes `Bonded`. It sees the payment afterwards, as the drop in the token account's `delegated_amount` since the allowance was set. `last_approved` holds the allowance as of the last observation, and the difference is what the session key has spent since.

Reconciliation

`reconcile_window` runs first in every instruction that books spending or touches the delegate: `pay`, `pay_escrow`, `refill_float`, `set_policy`, `rotate_session_key`, `withdraw` and `freeze`.

$$\begin{aligned} \text{periods} &= \lfloor (\text{now} - \text{window_started_at}) \div \text{window_seconds} \rfloor \\ \text{window_started_at} &\leftarrow \text{window_started_at} + \text{periods} \times \text{window_seconds} \\ \text{window_spent} &\leftarrow 0, \text{ when } \text{periods} > 0 \\ \text{consumed} &= \text{last_approved} - \text{delegated_amount} \\ \text{window_spent} &\leftarrow \text{window_spent} + \text{consumed} \\ \text{last_approved} &\leftarrow \text{delegated_amount} \end{aligned}$$

<code>now</code>	the instruction's <code>Clock</code> <code>unix_timestamp</code>
<code>delegated_amount</code>	SPL <code>delegated_amount</code> of the vault's token account, read in the same instruction
<code>consumed</code>	float spent by the session key since the previous reconciliation

It requires `now ≥ window_started_at`, `window_seconds > 0` and `delegated_amount ≤ last_approved`, and returns `InvalidConfiguration` otherwise. The last condition holds by construction: only the vault PDA can `Approve` on its token account, so between two reconciliations `delegated_amount` can only fall.

Two details are deliberate. The rollover happens before the booking, so float spent before a boundary but observed after it is charged to the new window. The program cannot date an SPL transfer, and charging it late means no window ever books less than it spent. And the window advances by whole periods from the stored start, so a vault idle for three windows moves its start forward three windows in one step, and its boundaries stay where creation put them.

Booking a payment

After reconciliation, `book_spend` tests the payment against the merchant's rule and the vault's budget. The full order of checks is in §3.4; the budget arithmetic is:

$$\begin{aligned} \text{rule.window_spent} + \text{amount} &\leq \text{rule.window_budget} \\ \text{window_spent} + \text{amount} + \text{delegated_amount} &\leq \text{policy.window_budget} \\ \text{rule.window_spent} &\leftarrow \text{rule.window_spent} + \text{amount} \\ \text{window_spent} &\leftarrow \text{window_spent} + \text{amount} \end{aligned}$$

`amount` the total debit; for `pay_escrow`, principal plus the escrow fee
`delegated_amount` the float allowance still outstanding after reconciliation

The vault-level test includes `delegated_amount`. Outstanding float reserves its share of the budget before any program payment, because the session key can spend it at any moment without asking the program; a program payment that ignored it could be followed by a float spend that pushed the window past its budget.

A merchant rule's counter shares the vault's window and resets lazily. When the rule's saved `window_started_at` differs from the vault's after reconciliation, the program copies the vault's start into the rule and zeroes `rule.window_spent` before the test. Readers render a rule counter from a stale window as zero.

Refunds never rewind either counter. An escrow refund returns USDC to the vault, but the window keeps the debit it booked: a refund is not a reason for the session key to spend again in the same window.

The window invariant

With the refill rule `allowance = min(float_cap, window_budget - window_spent)` (§4.4), every instruction leaves the vault inside these bounds for as long as its policy is unchanged:

$$\begin{aligned} \text{delegated_amount} &\leq \text{last_approved} \leq \text{float_cap} \leq \text{window_budget} \\ \text{window_spent} + \text{last_approved} &\leq \text{window_budget} \end{aligned}$$

A refill sets `last_approved` to at most the unspent budget. A payment books only when `window_spent + amount + delegated_amount` fits. A rollover zeroes `window_spent` and then books at most `last_approved`, which is at most `float_cap` and so at most `window_budget`.

The invariant gives the bound the window exists for. Under one policy, the USDC that leaves the vault through the float, `pay` and `pay_escrow` during any single window is at most `window_budget`. Every unit that leaves during the window is either booked in that window's `window_spent`, or is float spent after the window's last reconciliation, and the invariant already counts that in `last_approved`.

The shared domain model

`@bonded/contract` carries the same accounting as pure functions. `initializeWindowState(policy, at, initialFloatAllowance)` starts a window at `at` with no spending and an allowance no larger than `floatCap` (zero for a frozen policy). `evaluateWindow` applies one event, `float_spend`, `refill_float` or

`pay ; a float_spend` is checked only against the remaining allowance, as the SPL Token program checks it. `replayWindowAccounting` folds a chronological list of events and keeps the previous state whenever an event is refused, as a failed transaction leaves the chain unchanged. Vectors generated from these functions run against the compiled program, which must reproduce the same counters and remaining delegation.

One day of payments

Example. A vault has `float_cap` = 20,000,000 (20 USDC), `window_seconds` = 86,400 s (24 h), `window_budget` = 200,000,000 (200 USDC), an Ed25519 co-signer and `cosign_threshold` = 100,000,000 (100 USDC). Its window starts at 00:00. Two merchants have rules: an inference API, M1 (`per_transaction_cap` 50 USDC, `window_budget` 120 USDC, no quote required), and a GPU seller, M2 (cap 150 USDC, budget 150 USDC, quote required). The table gives amounts in USDC; the program holds base units.

Time	Event	delegated_ amount read	Booked as consumed	window_sp ent after	last_appro ved after	Outcome
00:05	<code>refill_float</code>	0	0	0	20	Allowance $\min(20, 200 - 0) = 20$
00:05– 03:00	150 x402 calls of 0.05 on the float	—	—	0	20	Delegated falls to 12.50
03:15	<code>pay</code> M1 30	12.50	7.50	37.50	12.50	$7.50 + 30 + 12.50 = 50 \leq 200$
07:00– 11:00	Float calls totalling 10.50	—	—	37.50	12.50	Delegated falls to 2
11:30	<code>pay</code> M1 60	—	—	37.50	12.50	<code>PerTransactionCapExceeded</code> ; nothing changes
11:31	<code>pay</code> M1 45	2	10.50	93	2	$M1\ 30 + 45 = 75 \leq 120$; $48 + 45 + 2 = 95 \leq 200$
12:00	<code>refill_float</code>	2	0	93	20	$\min(20, 107) = 20$; <code>Approve</code> replaces 2 with 20
14:00	<code>pay</code> M2 120, quoted, with approval	—	—	93	20	$93 + 120 + 20 = 233 > 200$: <code>WindowBudgetExceeded</code> ; the approval stays unconsumed
14:02	<code>pay</code> M2 85, quoted	20	0	178	20	$93 + 85 + 20 = 198 \leq 200$; $85 \leq 100$ needs no approval
18:00	Float calls totalling 20	—	—	178	20	Delegated falls to 0
18:10	<code>refill_float</code>	0	20	198	2	$\min(20, 2) = 2$
23:58	Float call of 2	—	—	198	2	Delegated falls to 0
00:03 next day	<code>refill_float</code>	0	2	2	20	One period passes: start + 86,400, spend reset, then 2 booked; $\min(20, 198) = 20$

On the first day the vault paid 40 USDC on the float and 160 USDC through `pay`, exactly its 200 USDC budget. The first window booked 198 of it; the last 2 USDC, observed after midnight, is charged to the second window. The refused payment at 11:30 shows the transaction rule: its reconciliation would have booked 10.50 USDC, but the refusal rolled the whole instruction back, and the next successful instruction booked it instead.

3.4 Merchant rules and the checks in `pay`

The rule

A `MerchantRule` is one allowlist entry with its own limits:

Field	Type	Meaning
<code>bump</code>	<code>u8</code>	PDA bump
<code>vault</code>	<code>Pubkey</code>	The vault the rule belongs to
<code>merchant</code>	<code>Pubkey</code>	The merchant wallet; it must own the destination token account
<code>per_transaction_cap</code>	<code>u64</code>	The largest single debit to this merchant
<code>window_budget</code>	<code>u64</code>	The most this merchant may receive in one vault window
<code>window_spent</code>	<code>u64</code>	Booked to this merchant in the rule's saved window
<code>quote_required</code>	<code>bool</code>	Every payment must carry a merchant-signed quote
<code>window_started_at</code>	<code>i64</code>	The vault window the counter belongs to

`set_merchant_rule(per_transaction_cap, window_budget, quote_required)` is signed by the admin, who pays rent the first time. The rule lives at `[b"rule", vault, merchant]` and is created when absent; at creation it records the vault, the merchant and the vault's current `window_started_at`. Every call overwrites the three limits and leaves `window_spent` alone, so a budget lowered mid-window applies against what the merchant has already received in it. The call emits `CoreEvent` kind 21 carrying the cap.

The rules are the vault's allowlist for program payments. A rule is never deleted. Setting `per_transaction_cap` to zero refuses every payment to that merchant, since every payment must be positive. The rules govern `pay` and `pay_escrow`; they cannot govern the float, where the session key is the top-level authority of an SPL transfer to any address. That lane is bounded by its allowance instead (§4.6).

The quote requirement is for merchants whose prices should bind the agent. With `quote_required` set, the agent does not choose the amount: the merchant signs a 169-byte quote naming the vault, the merchant, the amount, an expiry and a nonce (layout in §8), and `pay` accepts that amount once. Together with the per-transaction cap in base units, this is the guard against amount and decimal errors: the agent pays what the merchant priced, below a ceiling the admin set.

The order of checks

`pay(amount, nonce, quote_hash, intent_hash, quote_ix_index)` is signed by the session key. A separate payer signer pays the rent of the receipts and may be a relay. Anchor's account constraints run first, then the handler. The first failing check decides the error, and because a Solana transaction is atomic, a refusal leaves every account as it was, reconciliation included.

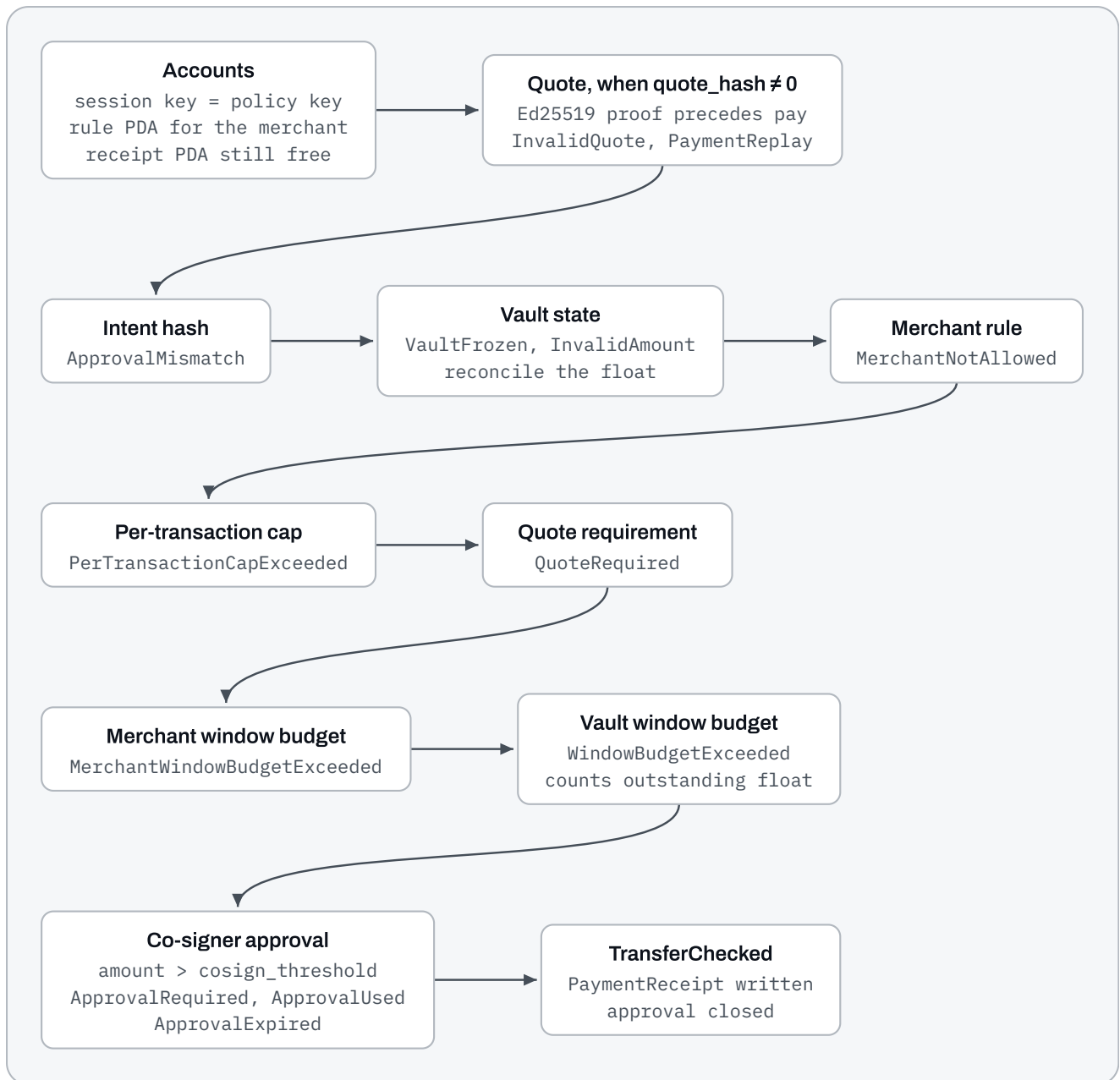


Figure 8. pay runs its checks in this order; the first failure decides the error and rolls back the whole instruction.

Each step, its condition and its refusal:

Step	Condition	Refusal
Accounts	The <code>session signer</code> equals <code>policy.session_key</code>	Unauthorized
Accounts	A <code>MerchantRule</code> exists at <code>[b"rule", vault, rule.merchant]</code>	Account validation fails: no rule, no payment
Accounts	The destination token account has the USDC mint and is owned by <code>rule.merchant</code>	Unauthorized
Accounts	No <code>PaymentReceipt</code> exists yet at <code>[b"payment", vault, intent_hash]</code>	Account initialization fails: the intent was already paid

Step	Condition	Refusal
Quote	With <code>quote_hash</code> nonzero: an Ed25519 proof immediately precedes <code>pay</code> ; the 169-byte quote names this vault, this merchant, this amount and lane 0, is unexpired, and was signed by the merchant	<code>InvalidPrecompile</code> , <code>InvalidQuote</code>
Quote	The quote's hash and nonce equal the arguments; its <code>QuoteReceipt</code> is unused	<code>InvalidQuote</code> , <code>PaymentReplay</code>
Quote	With <code>quote_hash</code> zero: no proof index and no quote receipt are passed	<code>InvalidQuote</code>
Intent	<code>intent_hash</code> equals the SHA-256 of the intent bytes (§3.5)	<code>ApprovalMismatch</code>
State	The vault is not frozen; <code>amount > 0</code>	<code>VaultFrozen</code> , <code>InvalidAmount</code>
State	Reconciliation preconditions (§3.3)	<code>InvalidConfiguration</code>
List	The rule's merchant is the payment's merchant	<code>MerchantNotAllowed</code>
Cap	<code>amount ≤ per_transaction_cap</code>	<code>PerTransactionCapExceeded</code>
Quote rule	A quote is present when <code>quote_required</code>	<code>QuoteRequired</code>
Merchant budget	<code>rule.window_spent + amount ≤ rule.window_budget</code>	<code>MerchantWindowBudgetExceeded</code>
Vault budget	<code>window_spent + amount + delegated_amount ≤ window_budget</code>	<code>WindowBudgetExceeded</code>
Co-signer	When <code>cosigner ≠ 0</code> and <code>amount > cosign_threshold</code> :an approval is passed, unused, unexpired, of the current revision, for this intent, merchant and amount	<code>ApprovalRequired</code> , <code>ApprovalUsed</code> , <code>ApprovalExpired</code> , <code>ApprovalPolicyChanged</code> , <code>ApprovalMismatch</code>

After the last check the program commits both counters, transfers `amount` with a CPI `TransferChecked` (6 decimals) from the vault's token account to the merchant's, signed by the vault PDA, writes the `PaymentReceipt` (lane 0, outcome 1, merchant paid), emits `PaymentCommitted` and `CoreEvent` kind 23, and closes a consumed approval to its original rent payer.

The order follows meaning. Identity and replay come first, so a payment that could never be valid fails before any state is read. The merchant's own limits come before the vault's. The co-signer check comes last, so an approval is consumed only by a payment every other rule already accepts: in the example of §3.3 the 14:00 payment fails on the vault budget and its approval remains usable.

In `pay` the merchant is read from the rule account itself, so the allowlist is enforced by the rule's existence and by the destination-owner constraint, and the `MerchantNotAllowed` comparison holds trivially. It does its work in `pay_escrow` , where the merchant comes from the quote. `pay_escrow` calls the same `book_spend` with the quote mandatory and `amount` equal to principal plus the escrow fee, so caps, budgets and approvals bind the full debit (§8).

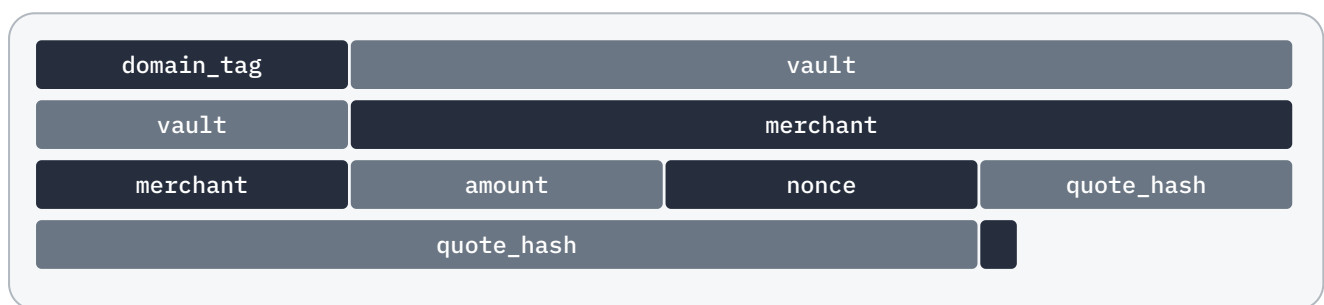
3.5 Approvals and the co-signer

The threshold

With a co-signer in the policy, every program payment whose debit is strictly above `cosign_threshold` needs an `Approval`. With `cosigner` zero the threshold is ignored. The co-signer is a key a person holds, Ed25519 or a passkey, and it adds a second decision above the amount where the operator wants one. It cannot start a payment. It approves one exact payment, which the session key then submits.

The payment intent

What the co-signer approves is a payment intent: the SHA-256 of 121 bytes naming the vault, the merchant, the debit, a nonce, the quote and the lane. `pay` recomputes it from its own arguments and accounts and refuses a mismatch with `ApprovalMismatch`, so an intent cannot be attached to a different payment. The intent hash is also the seed of the payment's `Approval` and of its permanent `PaymentReceipt`.



121 bytes · one cell per byte

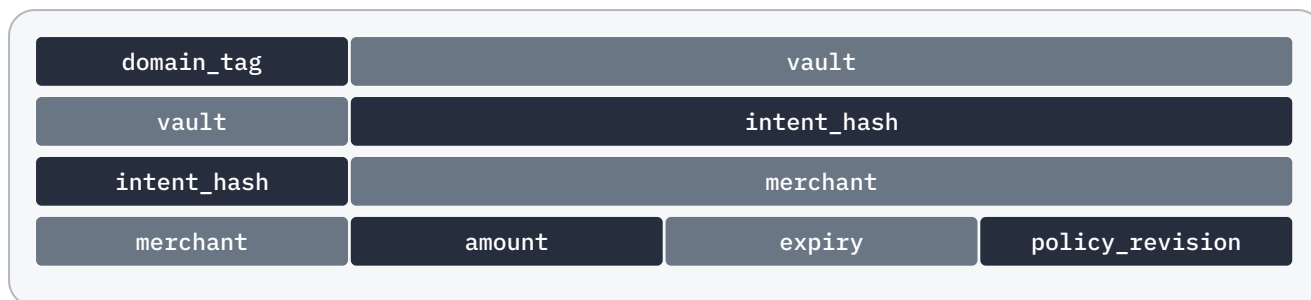
Offset	Bytes	Field	Meaning
0	8	domain_tag	SHA-256("Bonded payment intent v1\0" program ID genesis hash)[0..8]
8	32	vault	vault address
40	32	merchant	merchant wallet
72	8	amount	u64, total debit in base units
80	8	nonce	u64; the quote's nonce when a quote is used
88	32	quote_hash	SHA-256 of the 169-byte quote; zero when absent
120	1	lane	0 = pay, 1 = escrow

Figure 9. The 121-byte payment intent; its SHA-256 is the intent hash a co-signer approves.

The domain tag binds the intent to one program on one cluster (§1.5). For an unquoted payment the agent picks the nonce; distinct nonces make distinct intents and distinct receipts, so two equal payments to one merchant remain two payments.

The approval message

The co-signer signs 128 bytes:



128 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	domain_tag	SHA-256("Bonded approval v2\0" program ID genesis hash)[0..8]
8	32	vault	vault address
40	32	intent_hash	SHA-256 of the payment intent
72	32	merchant	merchant wallet
104	8	amount	u64, base units
112	8	expiry	u64, Unix seconds
120	8	policy_revision	u64, the vault's revision at signing

Figure 10. The 128-byte approval message (version 2) a co-signer signs.

`policy_revision` binds the approval to the policy in force when it was signed. `expiry` is the co-signer's choice and must lie in the future when the approval is recorded.

Ed25519 co-signer: `approve_payment`

`approve_payment(intent_hash, merchant, amount, expiry, ed25519_ix_index, policy_revision)` records an approval from an Ed25519 co-signer. The co-signer does not sign the transaction. Its signature travels in a native Ed25519 precompile instruction placed immediately before the Bonded instruction; any payer submits the transaction and pays the approval's rent. The handler checks, in order:

1. No `PaymentReceipt` exists at `[b"payment", vault, intent_hash]`, otherwise `PaymentReplay`. A committed payment can never be approved again.
2. `policy_revision` equals the vault's, otherwise `ApprovalPolicyChanged`.
3. `amount > 0` and `now < expiry ≤ i64::MAX`, otherwise `InvalidAmount`.
4. The proof is the instruction at index `current - 1`, the current instruction is a top-level Bonded instruction, and the proof's data is exactly 240 bytes with the header `[1, 0, 48, 0, 255, 255, 16, 0, 255, 255, 112, 0, 128, 0, 255, 255]`: one signature, public key at offset 16, signature at 48, a 128-byte message at 112, all inside the same instruction. Otherwise `InvalidPrecompile`.
5. The policy has a co-signer, the proof's key equals it, and the signed message equals the approval message rebuilt from the arguments. Otherwise `ApprovalMismatch`.

The approval is created at `[b"approval", vault, intent_hash]` with `used = false`, the payer as `rent_payer` and the signed revision; `ApprovalCreated` and `CoreEvent` kind 22 record it. Solana's precompile verifies the signature; the program verifies that the verified bytes are the ones it expects, from the key it expects, with no foreign offsets, extra signatures or wrapping program.

Passkey co-signer

A passkey lets a person approve with a phone or laptop authenticator through WebAuthn. Solana's native `secp256r1` precompile verifies the P-256 signature [30][38]; the program verifies everything WebAuthn wraps around it.

The admin first registers the credential with `set_passkey_credential(credential_id_hash, public_key, rp_id_hash, allowed_origin_hash)`, which creates a `PasskeyCredential` at `[b"passkey", vault, credential_id_hash]`:

Field	Type	Meaning
<code>bump</code>	<code>u8</code>	PDA bump
<code>vault</code>	<code>Pubkey</code>	The vault the credential serves
<code>credential_id_hash</code>	<code>[u8; 32]</code>	SHA-256 of the WebAuthn credential ID; the PDA seed
<code>public_key</code>	<code>[u8; 33]</code>	Compressed SEC1 P-256 key
<code>rp_id_hash</code>	<code>[u8; 32]</code>	SHA-256 of the relying-party ID
<code>allowed_origin_hash</code>	<code>[u8; 32]</code>	SHA-256 of the one origin allowed to request assertions
<code>sign_count</code>	<code>u32</code>	Last accepted authenticator counter

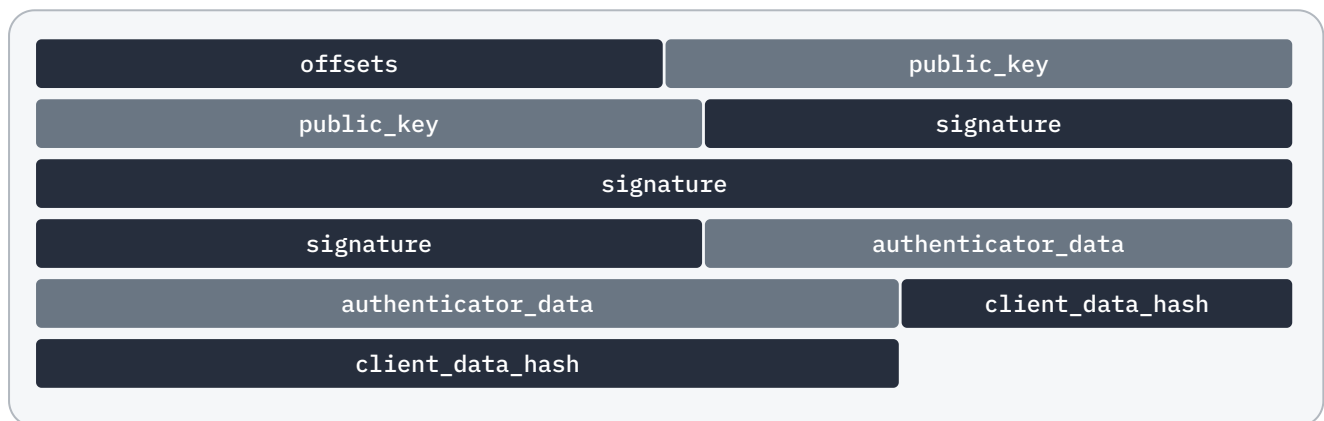
The key must be compressed (first byte 2 or 3), otherwise `InvalidPasskey`. Registration is immutable: repeating it with identical values succeeds without change, any other value is refused, and a different key or domain needs a new credential ID. No relying-party domain is inferred. The admin then selects the credential by writing its PDA address into `policy.cosigner` with `set_policy`. A PDA has no private key, so no Ed25519 proof can ever match it: choosing a passkey turns the Ed25519 path off.

`approve_payment_passkey(intent_hash, merchant, amount, expiry, precompile_ix_index, authenticator_data, client_data_json, policy_revision)` applies the same receipt, revision and amount checks as `approve_payment`, requires `policy.cosigner` to equal the credential's address (`ApprovalMismatch`), rebuilds the 128-byte approval message, and verifies the assertion:

Item	Check	What it establishes
Sizes	<code>authenticator_data</code> is 37 to 128 bytes; <code>client_data_json</code> is at most 512 bytes	Bounded parsing inside a legacy packet
Client data	A JSON object; duplicate keys refused; <code>type</code> is the string <code>"webauthn.get"</code> ; <code>crossOrigin</code> absent or <code>false</code>	An assertion requested by a top-level page
Challenge	Equals the base64url encoding, without padding, of SHA-256 of the approval message	The passkey signed this exact approval
Origin	SHA-256 of <code>origin</code> equals <code>allowed_origin_hash</code>	The request came from the operator's own origin

Item	Check	What it establishes
RP	<code>authenticator_data[0..32]</code> equals <code>rp_id_hash</code>	The credential's relying party
Flags	Byte 32 has UP (0x01) and UV (0x04) set and AT (0x40) clear	The user was present and verified
Counter	Bytes 33..37, big-endian <code>u32</code> : zero while the stored count is zero, otherwise greater than it	A cloned authenticator cannot replay an old count
Proof	The immediately preceding secp256r1 instruction has the canonical header, the credential's key, and the message <code>authenticator_data SHA-256(client_data_json)</code>	The P-256 signature covers exactly these bytes

Failures return `InvalidPasskey` or `InvalidPrecompile`. On success `sign_count` advances to the assertion's counter. Authenticators that always report zero are accepted, because replay is already prevented by the intent's single use; a zero counter can never reset a stored nonzero one.



182 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	16	<code>offsets</code>	[1,0,49,0,255,255,16,0,255,255,113,0,69,0,255,255]
16	33	<code>public_key</code>	compressed P-256 key, equal to the credential's
49	64	<code>signature</code>	compact ECDSA r s, low-S
113	37	<code>authenticator_data</code>	RP hash, flags, counter
150	32	<code>client_data_hash</code>	SHA-256 of <code>client_data_json</code>

Figure 11. The secp256r1 precompile instruction before `approve_payment_passkey`, for 37 bytes of authenticator data.

The approval is a transaction of its own. A WebAuthn assertion adds authenticator data and client data to the proof, and the assertion, a merchant quote and `pay` together would not reliably fit one 1,232-byte legacy packet. Measured on the built program, a passkey approval with 37 bytes of authenticator data is 897 bytes and 32,701 compute units. The `Approval` account carries the decision from that transaction into the payment.

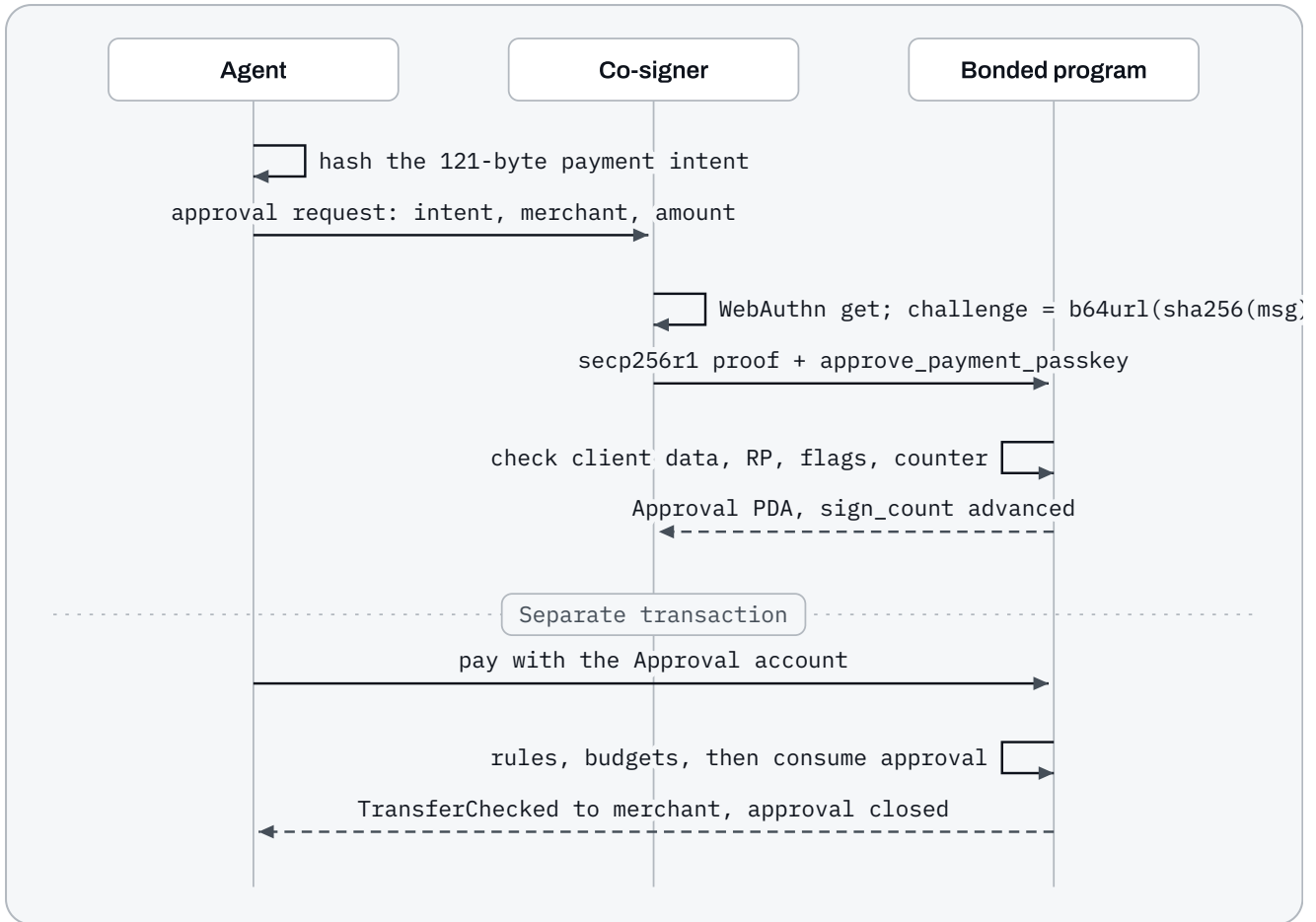


Figure 12. A passkey co-signer approves one payment above the threshold, and a later pay consumes the approval.

Consumption, expiry and single use

`pay` consumes an approval only when one is needed (the co-signer step of §3.4). It requires `used = false`, `now < expiry`, the vault's current revision, and an exact match of intent, merchant and amount. It then sets `used`, moves the USDC and closes the `Approval` in the same instruction, returning its rent to `approval.rent_payer`; the rent receiver passed must be that address, otherwise `ApprovalMismatch`. An approval passed with a payment at or below the threshold is left untouched.

Single use rests on the permanent receipt, not on the closed account. `pay` initializes the `PaymentReceipt` at `[b"payment", vault, intent_hash]`, so the same intent can never be paid twice, and `approve_payment` refuses an intent that already has a receipt, so a consumed approval cannot be recreated from the co-signer's old signature. The signed revision closes the remaining path: an approval voided by `set_policy` cannot be recorded again under the new policy, because its signature names the old revision.

An approval that is never used is closed with `close_approval`. Anyone may call it once `now ≥ expiry` or once the approval's revision differs from the vault's, and the rent goes only to the original payer; an unexpired approval of the current revision returns `ApprovalStillValid`. Once it is closed, a new approval for the same intent may be recorded with a fresh expiry, provided no payment has committed that intent.

Example. On the second day of the §3.3 example, the vault buys a quoted GPU job of 140 USDC from M2. The agent computes the intent from the quote (amount 140,000,000, the quote's nonce and hash, lane 0) and sends it to the co-signer, who signs an approval expiring 15 minutes ahead at `policy_revision 0`. The approval transaction records it. The agent then submits `pay : 140 ≤ 150` on the cap, the quote is present, M2's

counter gives $0 + 140 \leq 150$, the vault gives $2 + 140 + 20 = 162 \leq 200$, and $140 > 100$, so the approval is checked, consumed and closed. Had the admin changed the policy between approval and payment, `pay` would return `ApprovalPolicyChanged`, and anyone could close the void approval to refund its rent.

3.6 Session key rotation

`rotate_session_key(session_key)` replaces the vault's session key. The admin signs, with the same accounts as `set_policy`. The handler requires a nonzero key (`InvalidConfiguration`), reconciles the window, revokes the SPL delegate, sets `last_approved = 0`, writes the new key and emits `CoreEvent` kind 24.

From that instruction on, the old key can do nothing with the vault's USDC: `pay` and `pay_escrow` require the new key as signer, and the float delegate is gone. The new key receives an allowance at the next `refill_float`, whose `session` account must equal the new `policy.session_key`. Rotation leaves `policy_revision` unchanged. An approval binds an intent, a merchant, an amount and a policy, not the key that submits the payment, so a pending approval stays valid for the new key. The instruction has no frozen check, so the admin can rotate the key of a frozen vault before unfreezing it.

Credit lines keep their own key

Tabs follow a separate key schedule. A credit line copies the vault's session key into `voucher_session_key` when it is opened, and the merchant extends credit against vouchers signed by that key. Rotating the vault's key changes no line; lines opened afterwards adopt the new key. Each existing line rotates with `rotate_line_key`, on notice:

1. The admin calls `rotate_line_key(session_key)`. The line records `pending_voucher_key` with `effective_at = now + 259,200 s (72 h)` and emits `CoreEvent` kind 18. Repeating the same key before that time returns `KeyRotationDelayNotReached`; a different key restarts the notice.
2. Verification switches exactly at `effective_at`, with no transaction needed. At time t , a voucher must be signed by the pending key once $t \geq \text{effective_at}$, by the previous key for t before the last switch, and by the current key otherwise.
3. Calling `rotate_line_key` again with the same key at or after `effective_at` materializes the switch: the old key moves to `previous_voucher_key`, `rotated_at` records the switch time, and `CoreEvent` kind 19 marks it applied.

The 72 hours are the merchant's notice. During them the merchant can settle the outgoing key's latest voucher, or record it with `attest_line_voucher` in a `VoucherAttestation` at `[b"voucher-attestation", line, outgoing key, effective_at]`, which holds a maximum nonce and a cumulative ceiling. After the switch, an old-key voucher settles or claims only through that record and only up to both ceilings: the outgoing key cannot sign new debt, and the merchant keeps the debt it already accepted (§5). Scheduling and attestation remain available while the vault is frozen, while unfreeze recourse locks collateral and during an active default, so a rotation is never blocked by the lock it may be needed to get out of.

A replaced key is therefore removed in two stages. `rotate_session_key`, or a guardian `freeze`, ends its reach into the vault at once. `rotate_line_key` moves each line's voucher authority after the merchant's notice, and during the notice each line's limit bounds what any voucher on it can claim.

3.7 The guardian and freezing

Who can freeze

A guardian is a monitoring key whose only power is `freeze`. There are two guardian slots:

`Operator.guardian`, set by `create_operator` and covering every vault of the operator, and `policy.guardian` on each vault. `freeze` accepts the admin, the operator guardian or the vault guardian as signer; anyone else gets `Unauthorized`. `unfreeze` takes the admin alone. A guardian can stop a vault and cannot restart it, and no guardian can move USDC anywhere.

Either guardian is replaced with `rotate_operator_guardian` or `rotate_vault_guardian`, signed by the admin. The first call schedules the replacement 259,200 s (72 h) ahead and emits `GuardianRotated` with `applied = false`. Calling again with the same key at or after that time applies it; before it, the call returns `GuardianRotationDelayNotReached`; naming a different key restarts the notice; naming the current guardian returns `InvalidConfiguration`. The zero key removes the guardian. Rotation stays available while frozen and during a default, so an obsolete guardian cannot block its own replacement.

What `freeze` does

`freeze` takes the actor (writable signer), the `Operator`, the vault, its token account, the token program, the `FreezeCheckpoint` address of the next generation (of the current one when the vault is already frozen), the system program and the vault's freeze reserve. In order:

1. Check the actor.
2. Reconcile the window, so float spent up to the freeze is booked.
3. Revoke the SPL delegate. The session key loses the float in the same instruction.
4. If the vault is not already frozen: increment `freeze_generation` to `g`, create the `FreezeCheckpoint` at `[b"freeze", vault, g]`, start a new freeze episode when this is the vault's first freeze or the previous recourse tail has ended (`now > recourse_until`), set `frozen_at = now`, increment `Operator.frozen_vault_count`, and set `frozen = true`.
5. Set `last_approved = 0` and emit `CoreEvent` kind 15.

Freezing a vault that is already frozen repeats steps 1–3 and 5 against the same checkpoint, time and generation. A measured freeze with a pre-funded reserve is 409 bytes and 54,390 compute units.

Bonded's freeze is program state, separate from SPL Token's own account freeze, which belongs to the mint's freeze authority. The token account stays usable to the SPL Token program, so deposits still arrive and escrow refunds still return; it is the program, and the absence of a delegate, that close every outflow.

The checkpoint is evidence. It records the vault, the generation, `frozen_at`, the operator's `defaults_count` at that moment and the episode's first generation, and `unfreeze` later fills `unfrozen_at` and `recourse_until` once. Credit lines judge voucher and line expiry at a checkpoint's freeze time, so a merchant whose vouchers expire while the vault is frozen keeps the claim it held when the freeze began (\$7).

The freeze reserve

A guardian is usually a hot key run by a monitoring worker, holding no SOL. The vault's freeze reserve, a system-owned PDA at `[b"freeze-reserve", vault]` that the admin funds with

`fund_freeze_reserve(amount)` in lamports, pays the checkpoint's rent instead: 1,510,320 lamports for its 89 bytes. If the reserve would be left below the rent-exempt minimum of an empty account, the program moves all

of it, so no unusable remainder stays behind. When the reserve is short, the freezing signer pays, and a signer that cannot gets `FreezeReserveInsufficient` . A guardian with zero lamports and a separate fee payer can therefore freeze.

`close_freeze_checkpoint` is permissionless and returns a checkpoint's rent to the same reserve once no claim can still use it: for a checkpoint of an older episode, once 259,200 s (72 h) have passed since the current episode began; for the current episode, once the vault is unfrozen and 72 hours have passed after `recourse_until` . Earlier, it returns `FreezeCheckpointInUse` .

What a frozen vault refuses and what it allows

Action	While frozen
<code>pay</code> , <code>pay_escrow</code>	Refused: <code>VaultFrozen</code>
Float transfer	Refused by SPL Token: the delegate was revoked
<code>refill_float</code>	Refused: <code>VaultFrozen</code>
<code>withdraw</code>	Refused: <code>VaultFrozen</code>
<code>open_line</code> , <code>renew_line</code> , a <code>resize_line</code> increase	Refused: <code>VaultFrozen</code>
<code>settle_tab</code>	Moves nothing from the vault. Before grace: <code>VaultFrozen</code> . After grace: opens a default against the bond (§7)
Bond <code>request_withdrawal</code> , <code>finalize_withdrawal</code> ; applying a line decrease	Refused while any vault of the operator is frozen or in its recourse tail: <code>FrozenRecourseActive</code>
<code>unfreeze</code>	Allowed, admin only
<code>freeze</code> again	Allowed; same checkpoint
<code>set_policy</code> , <code>set_merchant_rule</code> , <code>rotate_session_key</code> , <code>set_passkey_credential</code>	Allowed
<code>approve_payment</code> , <code>approve_payment_passkey</code> , <code>close_approval</code>	Allowed; no payment can consume an approval until unfreeze
Guardian rotation, <code>rotate_line_key</code> , <code>attest_line_voucher</code> , scheduling a <code>resize_line</code> decrease	Allowed
<code>file_claim</code> , <code>execute_default_payout</code>	Allowed
Escrow transitions after funding	Allowed: the escrowed USDC already sits in the escrow's own token account (§8)
Deposits to the vault's token account	Allowed: an ordinary SPL transfer

Grace and the recourse tail

A freeze must not become a way to avoid paying tabs. Merchants that served the agent before the freeze hold valid vouchers, and a frozen vault will not pay them. Once the vault has been frozen for `freeze_grace_seconds` within the episode, a merchant may call `settle_tab` and open a default against the operator's bond; no USDC leaves the vault on that path.

$\text{frozen_seconds} = \text{episode_frozen_seconds} + (\text{now} - \text{frozen_at})$, the second term only while frozen

settle_tab on a frozen vault requires $\text{frozen_seconds} \geq \text{freeze_grace_seconds}$

$\text{freeze_grace_seconds} \geq \text{MIN_FREEZE_GRACE_SECONDS} = 345,600 \text{ s (96 h)}$

$\text{recourse_until} = \text{now} + \text{freeze_grace_seconds} + \text{CLAIM_WINDOW}$, at unfreeze

$\text{collateral unlocked} \Leftrightarrow \text{frozen_vault_count} = 0 \wedge \text{now} > \text{recourse_lock_until}$

<code>episode_frozen_seconds</code>	completed frozen intervals of the episode, added at each unfreeze
<code>freeze_grace_seconds</code>	ProtocolConfig value set by <code>initialize_protocol</code> ; 345,600 s in <code>backend/config/protocol.devnet.json</code>
<code>CLAIM_WINDOW</code>	259,200 s (72 h)
<code>recourse_lock_until</code>	Operator field, the latest <code>recourse_until</code> among its vaults

The 96-hour floor is the guardian rotation delay plus a margin. A guardian that freezes in bad faith cannot push the operator into a forced default before the admin's 72-hour replacement completes, and the operator keeps 24 hours to respond. Grace counts only frozen time: an admin who unfreezes promptly stops the clock, and repeated unfreeze and refreeze within one episode cannot restart time already accumulated.

`unfreeze` adds the finished interval to `episode_frozen_seconds`, sets `recourse_until`, writes `unfrozen_at` and `recourse_until` into the current checkpoint (the instruction takes it as its last account and checks its address, owner, vault, generation and that it is still open, `InvalidFreezeCheckpoint` otherwise), raises `Operator.recourse_lock_until` to at least `recourse_until`, decrements `frozen_vault_count` and clears `frozen`. It does not restore the float; the next `refill_float` does. A measured unfreeze is 376 bytes and 28,056 compute units.

Through the tail the bond stays locked. An operator cannot unfreeze, withdraw collateral and leave merchants without recourse for vouchers judged at the freeze. A freeze during the tail continues the same episode with its accumulated frozen seconds; a freeze after the tail starts a new episode with a fresh count.

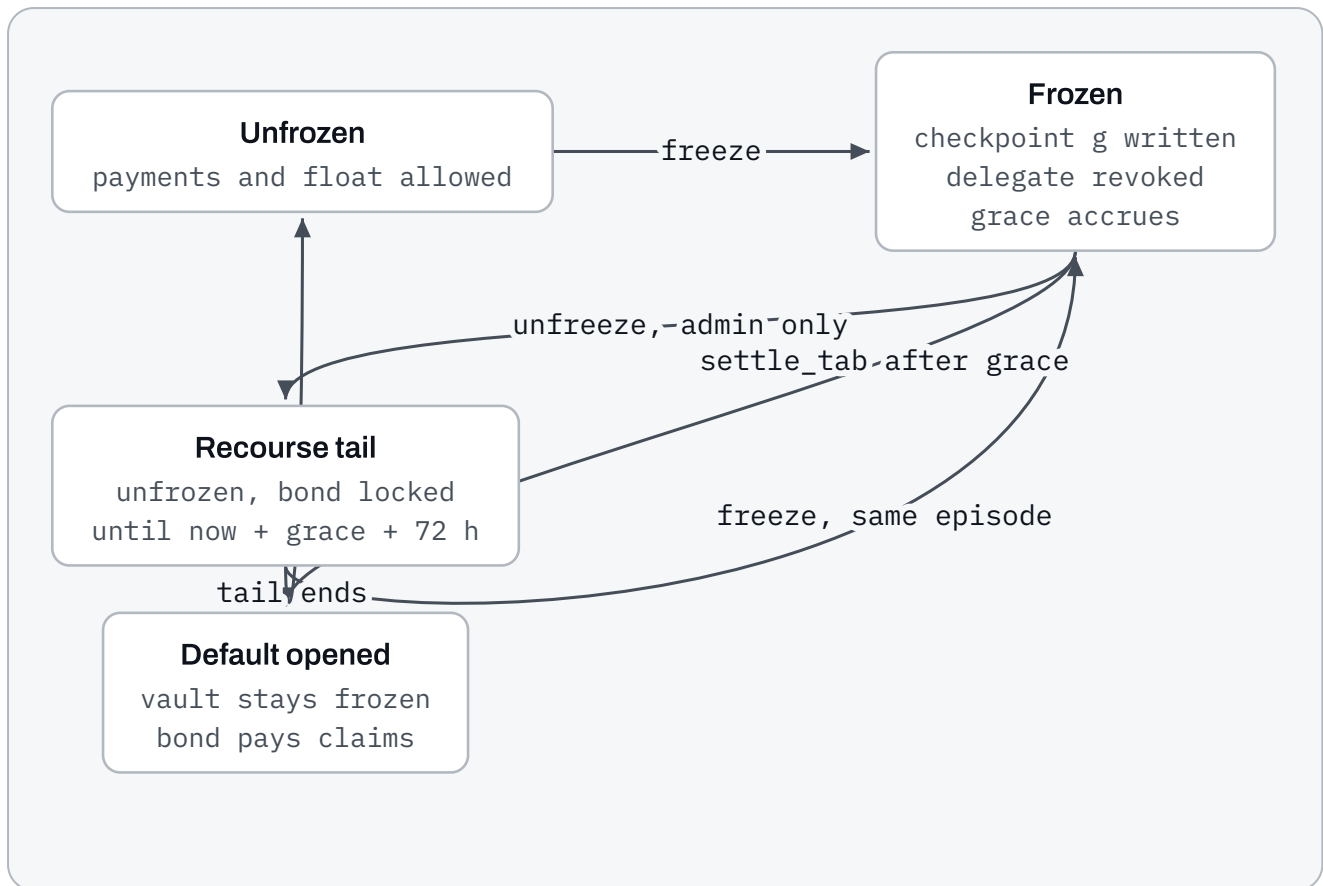


Figure 13. A vault's freeze states and the instruction that moves it between them.

The guardian worker

The program defines what a guardian may do; the guardian worker in `backend/src/guardian.ts` decides when. It reads committed payment projections (float transfers, `pay` and escrow fundings, leaving out `tab` settlements, whose timing the merchant chooses) and raises an alert on a burst of payments to new counterparties, a spend spike against a baseline, or spending above a window cap. Every rule value is operator configuration with no default. An alert becomes a durable freeze job, which a sponsored executor signs with the guardian key and a separate fee payer (§11.4).

3.8 Withdrawals from the vault

`withdraw(amount)` moves USDC from the vault to any classic SPL token account for the configured mint, which may belong to another treasury wallet. The admin signs. The accounts are the admin, the `Operator`, `ProtocolConfig`, the vault, the vault's token account, the destination, the mint and the token program. The handler:

1. Refuses a frozen vault (`VaultFrozen`) and a zero amount (`InvalidAmount`).
2. Reconciles the window.
3. Revokes the float and sets `last_approved = 0` .
4. Transfers `amount` with a CPI `TransferChecked` , signed by the vault PDA; the SPL Token program refuses an amount above the balance.
5. Emits `CoreEvent` kind 25.

A withdrawal is not session-key spending, so it books nothing against the window. Like every admin action on the vault, it ends with no delegate outstanding, and the agent's next allowance is issued by a refill under the vault's state after the withdrawal.

The vault has no cooldown and no safety margin; those belong to the bond. The vault's balance is working capital and carries no promise to merchants. A merchant on a tab relies on the operator's bond, and a bond withdrawal unlocks only at

$$\text{unlock_at} = \max(\text{now}, \text{longest_line_expiry}) + \text{CLAIM_WINDOW} + \text{withdrawal_margin_seconds}$$

<code>longest_line_expiry</code>	Bond high-water mark of every line's expiry
<code>CLAIM_WINDOW</code>	259,200 s (72 h)
<code>withdrawal_margin_seconds</code>	ProtocolConfig value, positive, set by <code>initialize_protocol</code> ; 86,400 s (24 h) in <code>backend/config/protocol.devnet.json</code>

so an operator cannot leave ahead of a default (§6). If the admin withdraws the vault below what a merchant's voucher asks, `settle_tab` finds the vault short and opens a default event, and the line holders' claims are paid from that bond (§7). A vault withdrawal moves the shortfall onto the operator's own collateral.

The frozen refusal completes the freeze. A frozen vault has every outflow closed: program payments, the float, tab settlement from the vault and withdrawal. Only the admin's `unfreeze` opens them again, and that instruction is a public on-chain act that starts the recourse tail and keeps the bond locked for its whole length.

4. The float

The float is the vault's lane for open-web x402 calls. It is a small USDC allowance on the vault's token account, held by the session key as SPL Token delegate, so the agent can pay any x402 endpoint the way an ordinary wallet does: with one top-level `TransferChecked`. The program does not see these payments as they happen. It bounds them in advance by the size of the allowance, and books them afterwards against the vault's window budget (§3.3).

4.1 Why the float exists

An x402 merchant is paid before it serves: its 402 response names an amount and a recipient, the client pays, and the merchant's facilitator verifies the payment. On Solana the `exact` scheme requires a top-level `TransferChecked` in a fixed instruction layout [7]. A vault cannot produce one. Its USDC is owned by a PDA, a PDA signs only through its program, and every program payment from a vault is therefore a `TransferChecked` issued by cross-program invocation from inside `pay`, an inner instruction. A facilitator that checks for a top-level transfer rejects that shape.

The float resolves it with SPL Token's delegate. The vault approves the session key as delegate of its token account for a capped amount. The session key then signs an ordinary top-level `TransferChecked` with itself as authority, which is the shape the `exact` scheme expects, and the facilitator pays the network fee [7]. The merchant needs nothing from Bonded.

The program cannot inspect these payments. A delegate transfer is an SPL Token instruction: Bonded is not invoked, merchant rules do not apply, and the destination can be any token account. The float is therefore a bound rather than a rule. Its cap limits what the session key can move between two program observations, the window budget limits the sum per window, and a freeze removes the delegate in one instruction (§4.6).

4.2 The allowance

SPL Token's delegate mechanics are the whole enforcement, so they are stated exactly:

- `Approve(delegate, amount)` sets the token account's delegate and sets `delegated_amount = amount`. It replaces any previous allowance; it never adds to it.
- A `TransferChecked` signed by the delegate succeeds only when `amount ≤ delegated_amount`, and lowers `delegated_amount` by the amount. At zero the delegate is cleared.
- `Revoke` clears the delegate and sets `delegated_amount = 0`.
- Only the token account's owner can `Approve` or `Revoke`.

The owner is the vault PDA. Only the program can grant or remove the allowance, and it does so in exactly five places: `refill_float` approves, and `set_policy`, `rotate_session_key`, `withdraw` and `freeze` revoke. Between two Bonded instructions the allowance can only shrink, as the session key spends it.

The vault tracks the allowance with two numbers: `policy.float_cap`, the most a refill may approve, and `last_approved`, the allowance as of the last reconciliation. With the token account they keep:

$$\text{delegated_amount} \leq \text{last_approved} \leq \text{float_cap} \leq \text{window_budget}$$

<code>delegated_amount</code>	SPL field of the vault's token account; the session key's current allowance
<code>last_approved</code>	Vault field; the allowance at the last reconciliation
<code>float_cap ≤ window_budget</code>	required by <code>create_vault</code> and <code>set_policy</code>

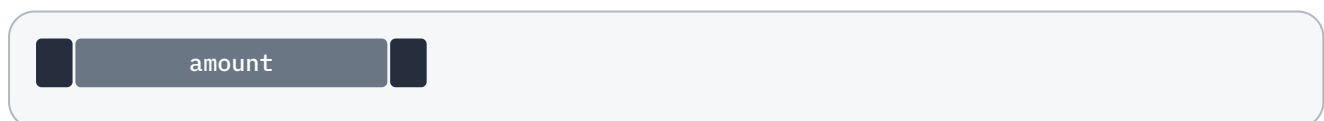
The allowance is a permission, not a reservation of tokens. `Approve` neither checks nor locks the balance, and a float transfer fails at the SPL Token program if the vault holds less than its amount. What the allowance reserves is window capacity: `pay` and `pay_escrow` count the outstanding `delegated_amount` as already spent when they test the vault budget (§3.3).

The program has no default float cap; each policy supplies its own. The brief's \$20 is an example, and the scripted example operator uses `float_cap = 20,000,000` (20 USDC) with a daily `window_budget` of 500,000,000 (500 USDC).

4.3 A float payment

An agent requests a paywalled endpoint, receives HTTP 402 offering the `exact` scheme, and pays with a transaction holding one SPL `TransferChecked` :

Account	Value
Source (writable)	The vault's USDC token account
Mint	The USDC mint, <code>ProtocolConfig.usdc_mint</code>
Destination (writable)	The merchant's token account named by the 402 offer
Authority (signer)	The session key, as delegate



10 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	1	<code>instruction</code>	12, <code>TransferChecked</code>
1	8	<code>amount</code>	u64, USDC base units
9	1	<code>decimals</code>	6, checked against the mint

Figure 14. The 10-byte data of the SPL Token `TransferChecked` instruction the session key signs as delegate.

The session key signs as transfer authority, the facilitator adds its fee-payer signature and submits, and SPL Token lowers `delegated_amount` . The example agent in `backend/examples/agent` makes exactly this payment: it refills, builds one `TransferChecked` from the vault's token account with the session key as

authority, and refills again to book it.

Because Bonded is not invoked, the indexer finds float payments in the vault token account's own transaction history. A USDC transfer with 6 decimals and a positive amount, whose source is a vault's token account owned by that vault, is recorded as a payment in lane `FLOAT`, and the guardian worker counts it with the vault's other session-initiated spending (§3.7, §11.4).

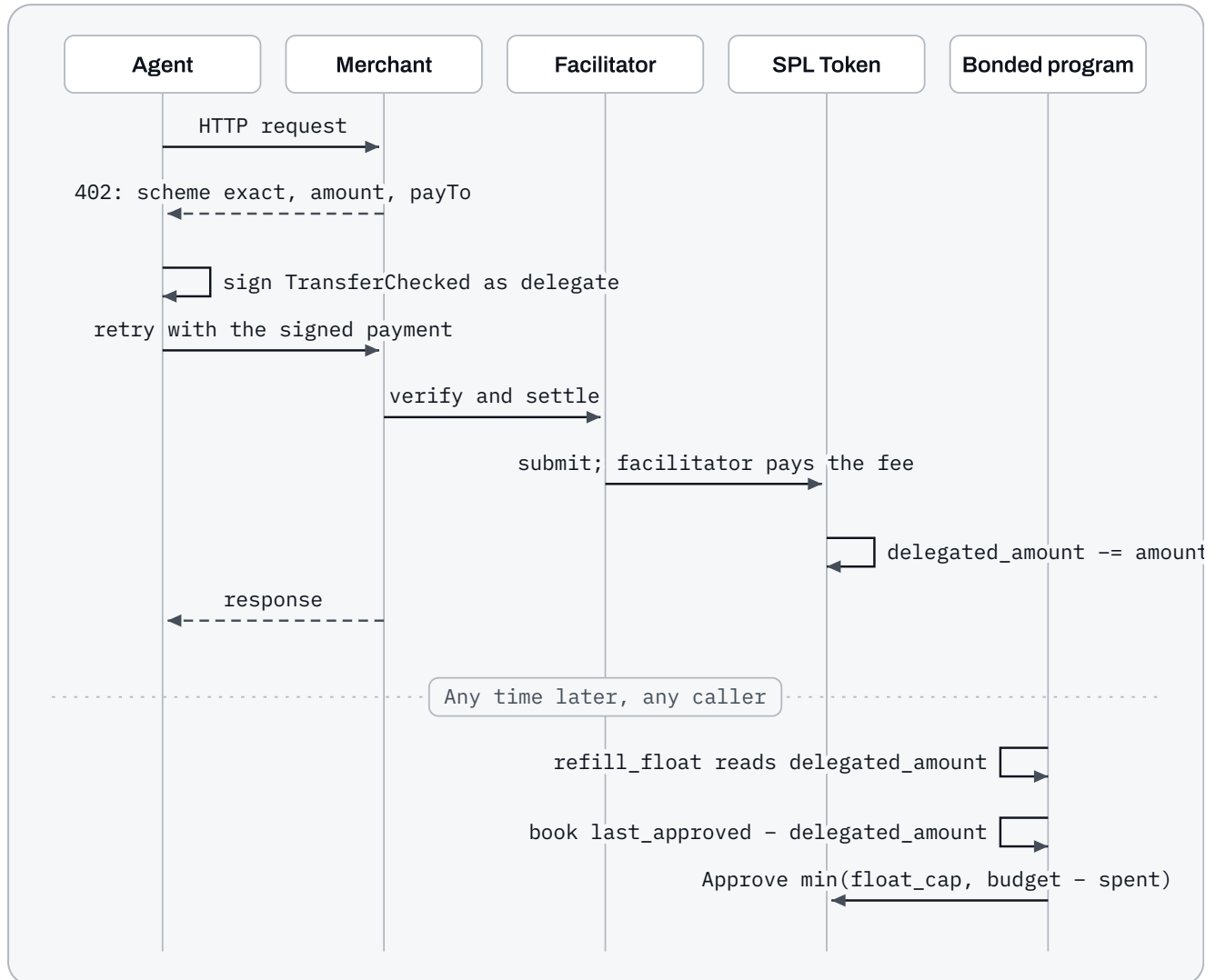


Figure 15. The session key pays an x402 endpoint from the float; a later refill books the payment and restores the allowance.

4.4 refill_float

`refill_float` takes no arguments and four accounts: the vault (writable), its token account (writable; it must be `vault.token_account` with the vault as authority), `session` (it must equal `policy.session_key` and does not sign), and the token program. No Bonded signer is required; anyone who pays the network fee may call it. The handler:

1. Refuses a frozen vault with `VaultFrozen`.
2. Reconciles the window (§3.3): rolls over whole periods, then books what the session key spent since the last reconciliation.
3. Requires `window_spent ≤ window_budget`, otherwise `WindowBudgetExceeded`.

4. Computes the new allowance and calls `SPL Approve` for it, signed by the vault PDA, with the session key as delegate.
5. Sets `last_approved` to the new allowance and emits `CoreEvent` kind 20 carrying it.

```

consumed = last_approved - delegated_amount
window_spent ← window_spent + consumed
allowance = min(float_cap, window_budget - window_spent)
delegated_amount ← allowance
last_approved ← allowance

```

<code>delegated_amount</code>	read from the vault's token account at the start of the instruction
<code>window_spent</code>	after any rollover, so float observed in a new window is charged to it
<code>allowance</code>	replaces the unspent remainder; it is never added to it

Four properties follow from the formula.

It books once. After a refill `last_approved = delegated_amount`, so a second refill books zero and approves the same allowance, unless a window boundary passed in between. No spend is ever counted twice, which is what allows anyone to call the instruction.

Anyone may call it. The outcome is fully determined by the policy, the window state and the token account. No caller chooses the amount, the delegate or a destination; the most any caller can do is bring the allowance up to the value the policy already allows. The agent (`refillFloat` in the agent SDK, `bonded_refill_float` in the MCP server), a keeper or a merchant's service can keep the float topped up without holding any vault key.

The window is its rate limit. Once `window_spent` reaches `window_budget`, the allowance is zero until the window rolls. The program has no separate refill cooldown: a refill may come at any time, and the budget left in the window is the only limit on how much it restores. Step 3 matters only after `set_policy` has lowered the budget below what the window already booked; the vault then cannot refill until the window rolls.

It overwrites. Step 4 replaces the remaining allowance. A session key holding 12.50 USDC of a 20 USDC float is reset to 20, not raised to 32.50, so the outstanding allowance never exceeds `float_cap`.

A refill is how float payments enter the window when no program payment follows them. Every other instruction that reconciles books them too, so the book is current after any Bonded instruction on the vault.

4.5 Every instruction that touches the float

Instruction	Books consumed float	Effect on the delegate
<code>refill_float</code>	Yes	<code>Approve(min(float_cap, window_budget - window_spent))</code>
<code>pay</code> , <code>pay_escrow</code>	Yes	None; the outstanding allowance counts against the vault budget

Instruction	Books consumed float	Effect on the delegate
set_policy	Yes, under the old policy	Revoke ; last_approved = 0 ; the next refill applies the new cap
rotate_session_key	Yes	Revoke ; the old key loses the allowance, the next refill names the new key
withdraw	Yes	Revoke ; last_approved = 0
freeze	Yes	Revoke ; refill_float refuses until unfreeze
unfreeze	No	None; the float stays empty until a refill

Every admin action on the vault, and every freeze, ends with no delegate outstanding. The agent's next allowance is always issued by a refill, under the vault's current policy, key and window state. A session key can never hold an allowance computed under a policy that has since changed.

4.6 The float as a loss bound

The float is the part of the vault the program cannot check payment by payment, so its design question is how much the session key can move without the program. Three quantities answer it:

$$\text{exposure_now} = \text{delegated_amount} \leq \text{float_cap}$$

$$\text{draw per refill} \leq \min(\text{float_cap}, \text{window_budget} - \text{window_spent})$$

$$\text{float spend in one window} \leq \text{window_budget} - \text{program spending booked in that window}$$

exposure_now what the session key can transfer to any address before the program acts again

program spending debits booked by pay and pay_escrow

At any instant the session key can move at most the outstanding allowance, at most `float_cap`, to any address. Every further unit needs a refill, and every refill first books what the previous allowance paid out. The float cap is therefore the bound between two program observations, and the window budget bounds the sum of all draws in a window, shared with the vault's program payments (§3.3). A freeze closes the lane in one instruction: it books the float spent up to that moment, revokes the delegate, and makes `refill_float` refuse until the admin unfreezes.

The two parameters do different work. `float_cap` sets the size of each draw, and with it how much leaves before a guardian can act on what it sees. `window_budget` sets the ceiling when no one acts. Payments to merchants the operator chose run through `pay`, where rules, quotes and approvals apply; open-web calls run on the float, where the protection is the size of the allowance the operator chooses to expose. The bond backs tabs; spending on the float is bounded by the allowance and the window rather than guaranteed.

4.7 Worked example

Example. A vault has `float_cap` = 20,000,000 (20 USDC), `window_budget` = 200,000,000 (200 USDC) and `window_seconds` = 86,400 s (24 h). At 09:00 the window has booked 60 USDC of program payments, and the session key holds a full float: `last_approved` = `delegated_amount` = 20,000,000.

Ordinary use. Between 09:00 and 10:00 the agent makes 140 x402 calls at 0.05 USDC each, 7,000,000 base units in all, and `delegated_amount` falls to 13,000,000. At 10:00 a keeper calls `refill_float` :

$$\text{consumed} = 20,000,000 - 13,000,000 = 7,000,000$$

$$\text{window_spent} = 60,000,000 + 7,000,000 = 67,000,000$$

$$\text{allowance} = \min(20,000,000, 200,000,000 - 67,000,000) = 20,000,000$$

values USDC base units; 1 USDC = 1,000,000

`Approve` sets the allowance back to 20 USDC. A second refill at 10:01 books zero and approves 20 USDC again. The invariant holds: $67 + 20 = 87 \leq 200$.

The bound at its limit. At 11:00 whoever holds the session key moves the whole 20 USDC to an address no merchant rule names. SPL Token accepts the transfer, because the key is the delegate. The holder then calls `refill_float` , which books 20 USDC and approves $\min(20, 200 - 87) = 20$ USDC, moves that too, and repeats:

Refill	<code>window_spent</code> after booking	New allowance	Total moved after the allowance is spent
—	67	— (holds 20)	20
1	87	20	40
2	107	20	60
3	127	20	80
4	147	20	100
5	167	20	120
6	187	13	133
7	200	0	133

Amounts are in USDC. With nobody acting, the outflow stops at 133 USDC, the window budget less the 67 USDC already booked; in the next window the same key could draw again, 20 USDC at a time, up to that window's budget. The key cannot use `pay` to reach the same address, because no merchant rule names it. If the guardian worker freezes the vault after the second draw, the outflow is 40 USDC: the freeze books the second draw, revokes the delegate and makes every further refill fail with `VaultFrozen` . The admin then rotates the session key (§3.6) and unfreezes, and the new key's first refill computes its allowance from the window as booked.

5. Tabs and vouchers

A tab is the lane for metered services billed in arrears. The merchant serves a request before it is paid. With each request the agent's session key signs a voucher that states the total the line owes so far. The merchant checks the voucher locally, keeps the newest one, and later presents it to the program with `settle_tab`, which moves the unpaid difference from the agent's vault to the merchant. When the vault holds less than that difference, the same instruction opens a default event and the operator's bond becomes the source of payment (§7). A tab needs no facilitator and no transaction per request. The only on-chain step is the settlement, and one settlement covers any number of requests.

This chapter follows a tab from the HTTP exchange to the transaction that settles it: the `bonded-tab` scheme (§5.1), the voucher bytes (§5.2), the domain tag (§5.3), cumulative amounts and nonces (§5.4), the merchant's checks (§5.5), `settle_tab` (§5.6), transaction size and compute (§5.7), settlement cadence (§5.8), and voucher attestations during a line-key rotation (§5.9).

5.1 The `bonded-tab` scheme

x402 lets an HTTP server answer a request with status 402 and a description of the payment it accepts; the client pays and retries. The `exact` scheme prepays every request. `bonded-tab` keeps the 402 negotiation and replaces the payment with a signed voucher on a credit line, so the merchant can serve first and collect later. The merchant middleware (`@bonded/merchant-middleware`) issues and verifies the scheme, the agent SDK (`@bonded/agent-sdk`) answers it, and both use the wire types and digest functions in `@bonded/contract/tab`.

The request

The agent sends an ordinary HTTP request with an `Idempotency-Key` header of 8 to 128 characters drawn from `A-Z a-z 0-9 . _ : -`. The key names one billable request: a retry with the same key, URL, method and body is the same request, and the merchant never bills it twice. The middleware calls three hooks the merchant host supplies: `admit` and `lineFor` before it reads any payment data, and `price` once it has decoded the payment and found no stored replay (a replay is answered from the store and never priced again).

`admit(request)` returns the authenticated client identifier (`clientId`), or `false`, which produces 429.

`lineFor(request)` returns the credit line the request draws on. `price(request)` returns the charge in USDC base units and the metered units, both positive `u64` values. On every request the middleware also confirms that its own RPC reports the configured cluster genesis hash.

The challenge

When the request carries no payment, the middleware computes the next voucher for the line and answers 402. The challenge appears twice: as JSON in the body and, base64-encoded, in the `Payment-Required` header, sent with `Cache-Control: no-store`. It contains `x402Version: 2`, the resource (`url`, `method`) and exactly one offer. The merchant proposes the next cumulative amount, nonce and expiry and the agent checks them. The counter stays in one place, the merchant's meter, and the agent still sees every value it needs to refuse a charge it did not agree to.

Offer field	Set by the middleware	Checked by the agent SDK before signing
<code>scheme</code>	<code>bonded-tab</code>	<code>equals bonded-tab</code>
<code>network</code>	<code>solana: + cluster genesis hash (base58)</code>	<code>equals solana: + the pinned genesis hash</code>

Offer field	Set by the middleware	Checked by the agent SDK before signing
asset	the configured USDC mint	equals the protocol's mint
amount	price(request).amount	greater than 0 and at most maxCharge
payTo	the merchant's public key	equals the line's merchant
extra.line	lineFor(request)	line state read fresh, active, vault not frozen
extra.requestKey	the Idempotency-Key	equals the caller's request key
extra.requestDigest	digest of method, URL, key and body (§5.4)	recomputed from the request bytes
extra.units	price(request).units	input to the usage digest
extra.cumulativeAmount	floor + amount (§5.4)	equals max(latest issued, settled, retired) + amount
extra.nonce	max(meter nonce, line maximum_nonce) + 1	above the line's and the latest issued nonce
extra.expiry	min(now + voucherTtlSeconds , line expiry)	after now, no later than the line expiry
extra.usageDigest	SHA-256 of [requestDigest, amount, units]	recomputed

The challenge's `resource.url` and `resource.method` must also equal the request the SDK sent, and the SDK refuses redirects, so a challenge for another resource cannot be answered with this request's voucher.

The payment

`fetchWithTab(input, init, { maxCharge, requestKey, maxRetries, maxRequestBytes })` handles the challenge. It decodes `Payment-Required`, verifies the network against the program ID and genesis hash the operator pinned (both the API's protocol record and the SDK's own RPC must match), and reads the line fresh from a trusted chain adapter or the authenticated API, never from the 402 alone. It signs only if every check above holds, then resends the same request bytes with a `Payment-Signature` header: base64 JSON of `{ x402Version: 2, scheme: "bonded-tab", network, requestKey, voucher }`. Before sending, it records the request authorization (request digest, header, charge, cumulative amount, nonce) under `origin:requestKey`. A retry of that key resends the same header, and the SDK never authorizes a second increment for one request key. A request body needs an explicit `maxRequestBytes` between 1 and 16,777,216 and is buffered once, so every attempt carries identical bytes.

voucher field	Content
line	credit line address, base58
cumulativeAmount, nonce, expiry	decimal strings of the signed u64 values
usageDigest	64 lowercase hex characters
messageBase64	the 96 signed bytes (128 base64 characters)
signatureBase64	the 64-byte Ed25519 signature (88 base64 characters)
signer	the session key, base58
receivedAt	ISO timestamp, recorded by the signer

Both headers are limited to 8,192 bytes of JSON, and the decoder accepts at most 12,000 base64 characters. The MCP server's `bonded_fetch_tab` tool calls the same method with a `maxCharge` taken from operator-configured per-call and window ceilings, so a model cannot raise the charge it authorizes.

Responses

Status	Code	When
402	—	no payment header: the challenge
200 or the service's status	—	voucher accepted; the response is persisted and replayed for the same key
400	INVALID_INPUT	missing or malformed <code>Idempotency-Key</code> ; undecodable payment header; payment for another request key, network or line; voucher that does not bind this request, line, key and expiry
403	FORBIDDEN	admission without a client identifier, or a request key that belongs to another client
409	CONFLICT	the same key is being served, belongs to another request or voucher, or the meter moved since the challenge
409	DEADLINE_NOT_REACHED	credit paused while outstanding vouchers settle (§5.8)
409	VAULT_FROZEN	the line's vault is frozen
409	PER_TRANSACTION_CAP_EXCEEDED	the line is not active, has expired, or cannot extend the charge within its limit
413	INVALID_INPUT	request body above <code>maxRequestBytes</code>
429	RATE_LIMITED	admission refused
503	INTERNAL_ERROR	the merchant could not complete the request, including a service failure; retry with the same key

On a 409 the SDK sends the request again without payment to obtain a fresh challenge, at most `maxRetries` times (one by default, two at most).

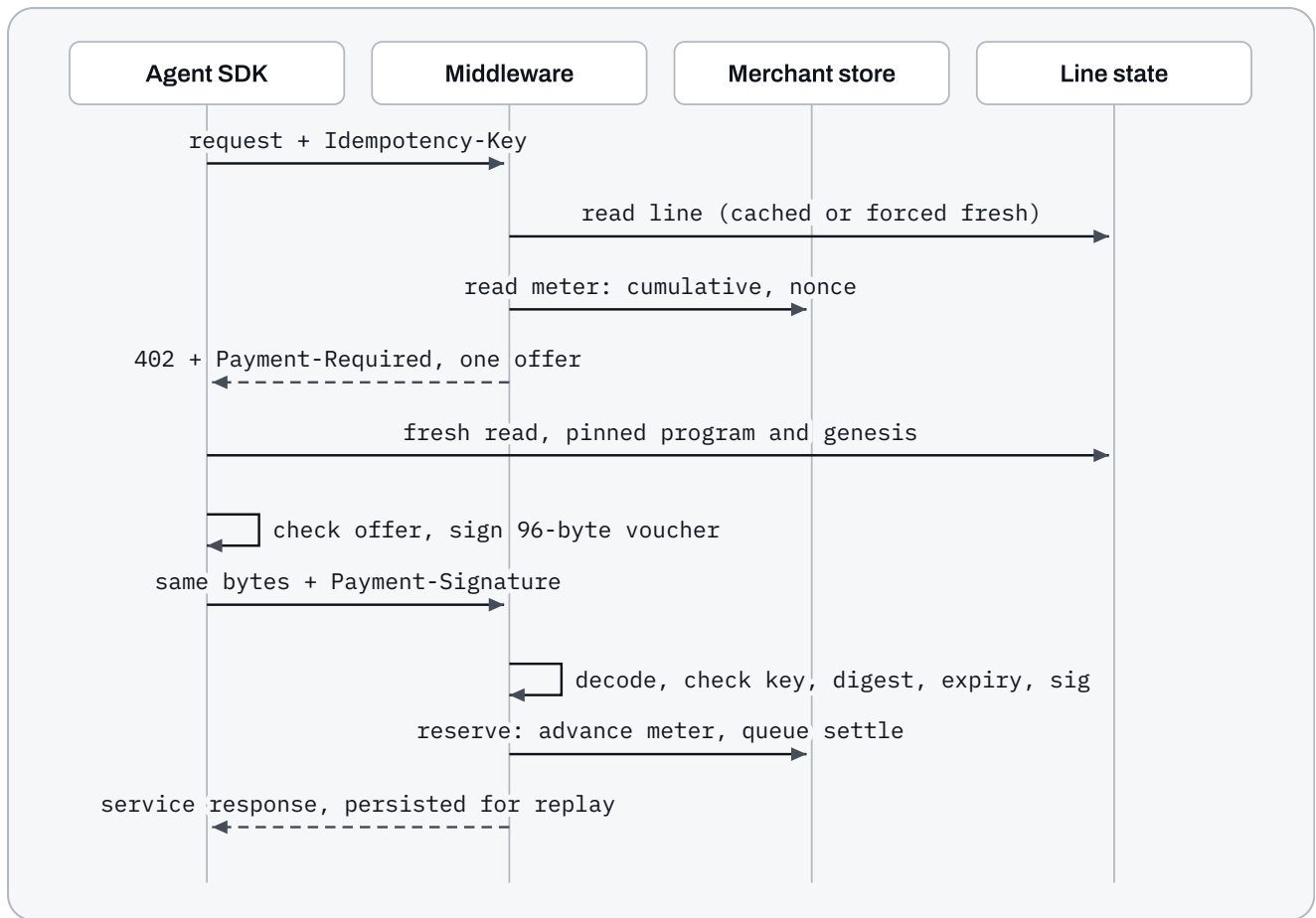
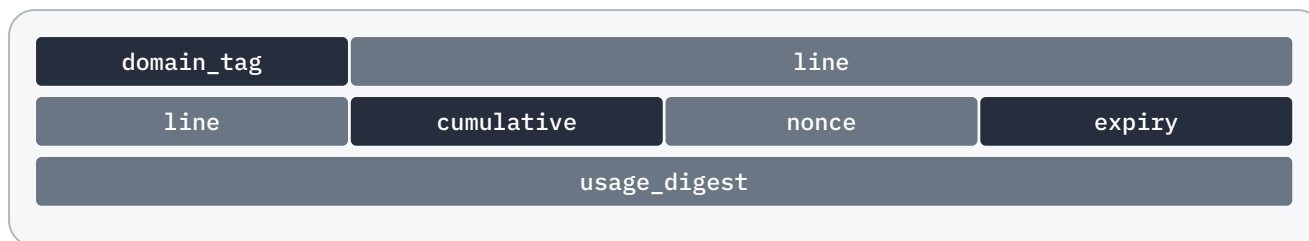


Figure 16. An agent answers a bonded-tab challenge with a signed voucher and receives the metered response.

5.2 The 96-byte voucher

The voucher is the agent's signed acknowledgement of debt. It is fixed-length, with every field at a fixed offset, so the program decodes it by slicing without length prefixes, and the Ed25519 precompile instruction that carries it can pin the message size to 96 (§5.6).



96 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	domain_tag	first 8 bytes of SHA-256(prefix program genesis)
8	32	line	the CreditLine PDA address, raw bytes
40	8	cumulative	u64 LE, total owed on the line, base units
48	8	nonce	u64 LE, strictly increasing per line
56	8	expiry	u64 LE, Unix seconds
64	32	usage_digest	SHA-256 binding request, charge and units

Figure 17. The 96-byte voucher the session key signs for every metered request.

Bytes	Field	Encoding	Checked on chain	Checked by the merchant
0–7	domain_tag	8 raw bytes	equals <code>ProtocolConfig.voucher_domain (InvalidVoucher)</code>	decoder compares with the tag derived from its signature context
8–39	line	32-byte address	equals the <code>line</code> account of the instruction <code>(InvalidVoucher)</code>	equals <code>lineFor(request)</code>
40–47	cumulative	u64 little-endian	above the line's cumulative floor; the delta fits the limit	equals floor + charge, exactly
48–55	nonce	u64 little-endian	above <code>maximum_nonce</code>	equals previous nonce + 1, exactly
56–63	expiry	u64 little-endian, Unix seconds	after the validation time, no later than <code>expires_at</code>	after now + settlement horizon, no later than the offered expiry
64–95	usage_digest	32 raw bytes	decoded, not interpreted	equals the recomputed usage digest

The voucher names no merchant, vault or amount increment because the line address implies them. A line is the PDA `[b"line", bond, merchant]`, and it stores the vault it draws on, so a voucher for one line can be settled only by that line's merchant, against that line's vault, under that line's bond. The increment is computed by the program from the line's stored floor (§5.4), never taken from the signer.

The shared test vector in `backend/test-vectors/core.json` shows the bytes for a program ID of $32 \times 0x11$ and a genesis hash of $32 \times 0x42$ (Example: synthetic identifiers):

17efe2eb836316fc	domain tag
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f	line
ffffffffffffffff	cumulative = 18 446 744 073
709 551 615	
2a00000000000000	nonce = 42
00d2496b00000000	expiry = 1 800 000 000
fffe fd fc fb fa f9 f8 f7 f6 f5 f4 f3 f2 f1 f0 ee fe ee de ce be ae 9e 8e 7e 6e 5e 4e 3e 2e 1e 0	usage digest

The browser codec generates this vector and the program's Rust tests decode it with the program's own `decode_voucher`, so both sides produce and accept the same 96 bytes. The same tests confirm that the bytes fail under a genesis hash of `32 × 0x43`, under another program ID, and when truncated to 95 bytes.

5.3 The v2 domain tag

An Ed25519 signature over 96 bytes is valid wherever those bytes are presented. The first eight bytes of the voucher bind it to one deployment of the program on one cluster, so a signature made for one cannot be spent on another.

$$\text{domain_tag} = \text{SHA-256}(\text{"Bonded voucher v2\0"} \parallel \text{program_id} \parallel \text{cluster_genesis_hash})[0 \dots 7]$$

"Bonded voucher v2\0"	18 bytes of UTF-8, the last one NUL
program_id	the 32 raw bytes of the Bonded program address, not base58 text
cluster_genesis_hash	the 32 raw bytes of the cluster's genesis hash
[0 ... 7]	the first 8 bytes of the 32-byte digest; the hash input is 82 bytes

`initialize_protocol` takes `cluster_genesis_hash` as an explicit argument and derives the tag in the program from its own ID: `voucher_domain(crate::ID, &cluster_genesis_hash)`. Both values are stored in `ProtocolConfig`. Initialization requires the signature of the program's actual upgrade authority, checked against the loader's `ProgramData` account, so no other key can install a different tag. Every instruction that reads a voucher (`settle_tab`, `file_claim`, `attest_line_voucher`) compares bytes 0–7 with the stored `voucher_domain` before it reads any voucher field. A voucher with another tag, including the old literal tag `BONDTAB1`, fails with `InvalidVoucher`.

The two inputs stop two different replays.

- **Across clusters.** PDAs derive from the program ID and seeds only. An operator who uses the same admin key, merchant and program ID on devnet and on mainnet has the same `Operator`, `Bond` and `CreditLine` addresses on both, and the session key may be the same too. Without the genesis hash in the tag, a voucher signed against a devnet line would carry a valid signature and a valid line address on mainnet. The genesis hash makes the two tags differ.
- **Across programs.** Another deployment of the program, or any program that reuses the layout, has a different ID. The line address already differs there, and the tag differs as well, so a session key that signs vouchers for several deployments produces signatures each deployment alone accepts.

The other messages Bonded signs use the same derivation with their own prefixes, so a signature over one kind of message cannot be presented as another.

Message	Prefix	Size
Voucher	Bonded voucher v2\0	96 bytes
Merchant quote	Bonded quote v1\0	169 bytes
Payment intent	Bonded payment intent v1\0	121 bytes
Co-signer approval	Bonded approval v2\0	128 bytes

Sign-in messages carry the distinct prefix `Bonded off-chain authentication v1`, and the SDK refuses to sign arbitrary API messages, voucher bytes included, through its login path. Both off-chain parties pin the tag inputs. The SDK's `verifyNetwork()` requires the API's `voucherDomain.programId` to equal the operator-pinned program ID, and both the API's `clusterGenesisHash` and the SDK's own RPC `getGenesisHash()` to equal the pinned genesis hash; `signatureContextFromProtocol` recomputes the tag and requires it to equal the API's `tagHex`. The middleware refuses to start if its signature context and configured genesis hash differ, and refuses every request while its RPC reports another genesis hash.

5.4 Cumulative amounts and nonces

A voucher states the total owed on the line since it opened, not the amount of one request. The newest voucher therefore includes every earlier one, and a merchant that holds it can collect the whole tab with a single settlement. Settling records the voucher's amount and nonce on the line, and every older voucher falls below those floors.

$$\text{floor} = \max(\text{line.settled_cumulative}, \text{line.retired_cumulative})$$

$$\text{delta} = \text{voucher.cumulative} - \text{floor}$$

$$\text{accepted} \Leftrightarrow \text{voucher.nonce} > \text{line.maximum_nonce} \wedge \text{voucher.cumulative} > \text{floor}$$

$$\wedge \text{delta} \leq \text{line.limit} \wedge \text{validation_at} < \text{voucher.expiry} \leq \text{line.expires_at}$$

<code>line.settled_cumulative</code>	cumulative amount of the last voucher the vault paid
<code>line.retired_cumulative</code>	highest voucher amount resolved by a default payout (\$7)
<code>line.maximum_nonce</code>	nonce of the last settled or paid voucher
<code>validation_at</code>	the current time, or the line's freeze reference for a frozen vault

Both the nonce and the cumulative amount must strictly advance. Requiring both makes the accepted vouchers a single chain: a voucher with a higher amount but an old nonce, or a new nonce for the same amount, is rejected with `VoucherReplay`. The program requires an increase, not a step of exactly one, so a merchant can settle any voucher in its history that is newer than the floor. The retired floor stops a voucher from

being collected twice. A default payout raises `retired_cumulative` to the claimed voucher's amount and `maximum_nonce` to its nonce, so the same debt cannot be settled again from the vault after the bond has resolved it.

The limit bounds the unsettled amount, not the line's lifetime volume. After each settlement the floor rises, and the agent can again owe up to `limit` above it.

Example: line limit 10,000,000 (10 USDC)

	Charge	Cumulative	Nonce	Floor after	On chain
Request 1	1,200,000	1,200,000	1	0	—
Request 2	800,000	2,000,000	2	0	—
Request 3	3,000,000	5,000,000	3	5,000,000	<code>settle_tab(v3)</code> pays 5,000,000
Request 4	2,500,000	7,500,000	4	5,000,000	—
Request 5	500,000	8,000,000	5	8,000,000	<code>settle_tab(v5)</code> pays 3,000,000
Replay of v4	—	7,500,000	4	8,000,000	refused: <code>VoucherReplay</code>
Request 6	9,000,000	17,000,000	6	8,000,000	delta 9,000,000 ≤ 10,000,000

Vouchers v1, v2 and v4 were never settled and need not be: v3 and v5 contain them.

How the merchant issues the next voucher

The middleware derives each offer from its meter and the chain state:

$$\text{chain_floor} = \max(\text{line.settled_cumulative}, \text{line.retired_cumulative})$$

$$\text{cumulative}_n = \max(\text{meter.cumulative}, \text{chain_floor}) + \text{charge}_n$$

$$\text{nonce}_n = \max(\text{meter.nonce}, \text{line.maximum_nonce}) + 1$$

$$\text{expiry}_n = \min(\text{now} + \text{voucherTtlSeconds}, \text{line.expires_at})$$

$$\text{cumulative}_n - \text{chain_floor} \leq \text{effective_limit}$$

`meter.cumulative`, `meter.nonce` the merchant's stored counters for this line

`effective_limit` line.limit, or a lower pending limit whose `effective_at` has passed

The last line is the merchant's limit check. It uses a matured pending decrease before the chain applies it, because `settle_tab` applies a matured decrease before it checks the delta (§5.6, §6.4). The agent SDK runs the same calculation independently. It signs only if the offered cumulative equals the highest of its own latest issued voucher, the line's settled amount and its retired amount, plus the charge it was shown. A merchant cannot add a charge the agent did not see.

The usage digest

The usage digest ties a voucher to the request and charge that produced it:

```
request_digest = hex SHA-256( JSON[ METHOD, url_without_fragment,  
                                request_key, hex SHA-256(body) ] )
```

```
usage_digest = SHA-256( JSON[ request_digest, amount, units ] )
```

METHOD	the HTTP method in upper case
request_key	the Idempotency-Key
amount, units	the charge and metered units as decimal strings
JSON[...]	JSON.stringify of the array, UTF-8

The program carries the digest without interpreting it. It is evidence for the parties: the merchant keeps every request's voucher, so each increment of the cumulative amount can be traced to one request, its body hash, its charge and its units. For off-chain review of metering, the newest voucher establishes the total, and the stored history attributes each part of it to one request.

5.5 Merchant-side verification

The merchant decides at request time whether to extend credit, using its own copy of the line state, and confirms everything the program will check later. A voucher the middleware accepts is one `settle_tab` would accept. The middleware holds no private key; it reads, verifies and records.

For a request with a `Payment-Signature` header the middleware runs these steps in order:

1. **Admission and framing.** `admit` must return a client identifier; the body must fit `maxRequestBytes` ; the `Idempotency-Key` must be well formed; the merchant's RPC must report the configured genesis hash.
2. **Envelope.** The payment's `network` , `requestKey` and `voucher.line` must equal `solana:<genesis>` , the request's key and `lineFor(request)` .
3. **Idempotent replay.** The store looks up `(merchant, line, requestKey)` . A completed request with the same client, request digest and exact voucher proof is answered from the stored response. One that is in service returns 409. One from another client returns 403, and one with different bytes or proof returns 409.
4. **Line state.** Reads are cached for `cacheMilliseconds` . A charge, or an outstanding increment over the settled amount, that reaches `rereadIncrement` forces a fresh read.
5. **Pause conditions.** Credit pauses and settlement is queued when the line expiry, a pending key switch, or the newest outstanding voucher's expiry falls within `settlementHorizonSeconds` (§5.8).
6. **Line checks.** The vault must not be frozen; the line must be active; the charge must fit the effective limit (§5.4).
7. **Voucher bytes.** The 96 bytes are decoded under the signature context (domain tag). Their cumulative amount, nonce, expiry, line and usage digest must equal the DTO's fields and the recomputed usage digest. `signer` must equal the line key effective now, `voucherKeyAt(line, now)` , which switches to a scheduled

key exactly at its `effective_at` . The expiry must lie beyond now + horizon and no later than the offered expiry. The signature must verify under strict RFC 8032 rules (`zip215: false`), which refuse non-canonical encodings.

8. **Reservation.** Under a PostgreSQL advisory transaction lock on `merchant:line` , the voucher's cumulative amount must equal $\max(\text{meter}, \text{chain floor}) + \text{charge}$ and its nonce $\max(\text{meter nonce}, \text{chain nonce}) + 1$. The meter advances, the usage row is created, and the line's settlement job is queued. A concurrent request that lost the race receives 409 and fetches a new challenge.
9. **Service.** The service callback receives the idempotency key, units and voucher. Its response, bounded by `maxResponseBytes` , is stored before it is returned.

Steps 3 and 8 make billing exactly-once per request key. Each service execution holds a lease of `serviceLeaseMilliseconds` , which must exceed `maximumServeMilliseconds` . A failed service releases its lease, and a retry with the same key resumes the same billed request instead of creating a new charge.

The middleware keeps its state in PostgreSQL through one of two adapters with the same three roles. `createBondedMerchant` uses the built-in `pg` store, whose tables `BondedMerchantMeter` , `BondedMerchantUsage` and `BondedMerchantJob` are created by `bonded-merchant migrate` ; its worker scans the lines on each run. The Prisma adapter (`@bonded/merchant-middleware/postgres`) keeps them in these tables:

Table	Key	Holds
Merchant Meter	<code>merchant:line</code>	cumulative amount, nonce, latest accepted voucher
Merchant Usage	SHA-256 of (merchant, line, request key)	client ID, request key and digest, charge, units, the voucher, state (<code>RESERVED</code> , <code>COMPLETED</code>), lease owner and deadline, stored response status, headers and body
Merchant ChainJob	<code>settle:<merchant>:<line></code> , or a hash for claims and attestations	<code>kind(settle_tab , file_claim , attest_line_voucher)</code> , state, prepared ID, signed wire bytes, submit key, voucher and nonce in flight, attempts, next attempt time, lease, confirmed signature

The meter answers "what is owed now". The usage table is the proof history: every request's own voucher, kept for replay and for the worker, which searches it for the newest voucher that can still settle. The job table is the settlement outbox.

5.6 settle_tab on chain

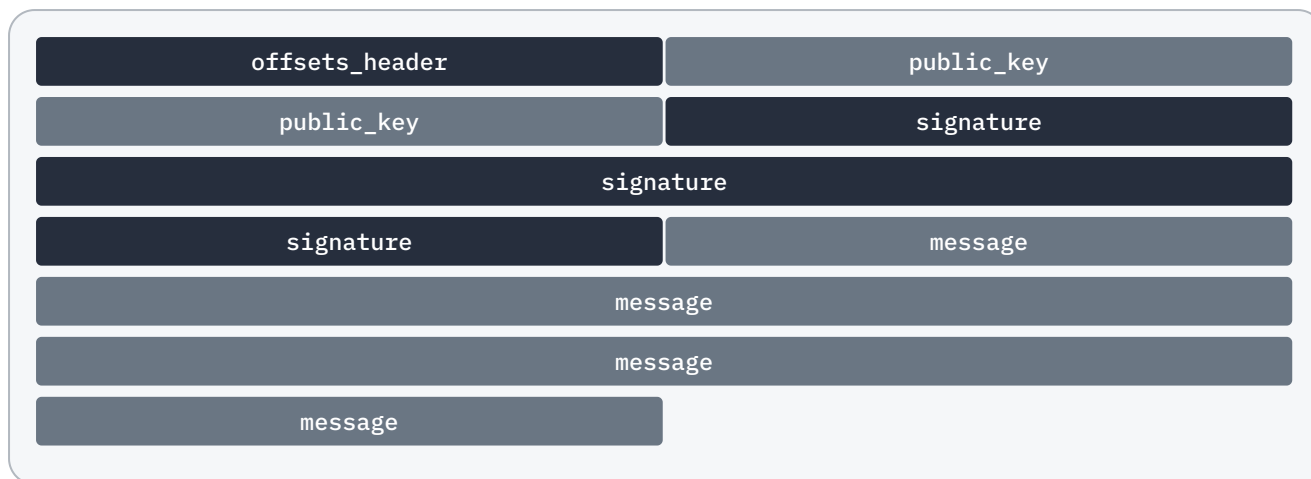
`settle_tab(default_sequence: u64, ed25519_ix_index: u16)` is signed by the merchant that holds the line. Its 18 bytes of instruction data are the 8-byte discriminator (the first 8 bytes of SHA-256(`global:settle_tab`)), the bond's next default sequence and the index of the proof instruction.

#	Account	Writable	Constraint
0	<code>merchant</code>	yes, signer	the line's merchant; pays the placeholder rent
1	<code>protocol</code>		<code>[b"protocol"]</code> : voucher domain, fee treasury, mint
2	<code>operator</code>	yes	<code>[b"operator", operator.admin]</code> : <code>tabs_settled</code> , <code>defaults_count</code>
3	<code>vault</code>		<code>[b"vault", operator, agent_id]</code> , belongs to the operator

#	Account	Writable	Constraint
4	vault_token	yes	vault.token_account , authority the vault, USDC mint
5	bond	yes	[b"bond", operator]
6	line	yes	[b"line", bond, merchant] , with this bond, vault and merchant
7	default_event	yes	[b"default", bond, default_sequence] , init_if_needed
8	merchant_token	yes	a USDC account owned by the merchant
9	fee_treasury	yes	protocol.fee_treasury
10	mint		protocol.usdc_mint
11	instructions		the instructions sysvar address
12	token_program		classic SPL Token
13	system_program		
14	freeze_checkpoint		canonical [b"freeze", vault, generation] PDA for the line's reference
15	voucher attestation		optional first remaining account (§5.9)

The Ed25519 proof and the instructions sysvar

Ed25519 verification runs in Solana's native precompile, not in the program. The transaction carries an Ed25519 program instruction holding the public key, signature and message; the runtime verifies it before any instruction executes, and a bad signature fails the whole transaction. The program then has to establish that the verified message is this voucher, verified under this key. It reads the transaction's instructions from the instructions sysvar and accepts exactly one form:



208 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	16	offsets_header	one signature; every instruction index is 0xFFFF
16	32	public_key	the key that signed the voucher
48	64	signature	Ed25519 over bytes 112–207
112	96	message	the voucher, byte for byte

Figure 18. The 208-byte Ed25519 precompile instruction that carries one voucher proof.

Header bytes	Field	Required value
0	number of signatures	1
1	padding	0
2–3	signature offset	48
4–5	signature instruction index	0xFFFF (this instruction)
6–7	public key offset	16
8–9	public key instruction index	0xFFFF
10–11	message offset	112
12–13	message size	96
14–15	message instruction index	0xFFFF

`verified_voucher` performs these checks, in order:

1. `load_current_index_checked` gives the index of the executing instruction; it must be greater than 0, and `ed25519_ix_index` must be exactly one less. The proof immediately precedes the settlement.
2. The instruction at the current index must belong to the Bonded program itself. When Bonded is reached through a CPI, the top-level instruction belongs to the calling program, so CPI wrappers are refused.
3. The proof instruction must belong to the Ed25519 program, list no accounts and carry exactly 208 bytes of data.

4. Its first 16 bytes must equal `[1, 0, 48, 0, 255, 255, 16, 0, 255, 255, 112, 0, 96, 0, 255, 255]` .
5. Bytes 112–207 are decoded as the voucher, with the domain tag check, and the voucher's line must equal the `line` account.
6. Bytes 16–47 are returned as the signer.

The header pins every instruction index to `0xFFFF`, which tells the precompile to read key, signature and message from its own data. Without that, a proof could verify a signature over bytes in some other instruction while the program decoded different bytes here. Pinning the offsets and the count to one signature means the bytes the runtime verified are the bytes the program reads. Any other layout, a second signature, a proof elsewhere in the transaction, or a reference into another instruction fails with `InvalidPrecompile` .

Checks

Order	Check	Error
1	the freeze checkpoint account is the line's canonical reference PDA, or a valid checkpoint of the current episode	<code>InvalidFreezeCheckpoint</code>
2	the proof is canonical and the voucher's tag and line match	<code>InvalidPrecompile</code> , <code>InvalidVoucher</code>
3	the signer is the line key effective now, activated at or before the validation time, or an attested outgoing key within its ceilings (§5.9); for a frozen vault, judged at the freeze reference	<code>InvalidPrecompile</code> , <code>VaultFrozen</code>
4	a frozen vault has accumulated <code>freeze_grace_seconds</code> frozen in its episode	<code>VaultFrozen</code>
5	no default is active on the bond	<code>DefaultActive</code>
6	<code>default_sequence = bond.default_sequence + 1</code>	<code>InvalidDefault</code>
7	the line is in the bond's current default epoch and the validation time is before <code>line.expires_at</code>	<code>LineExpired</code>
8	<code>validation time < voucher.expiry ≤ line.expires_at</code>	<code>VoucherExpired</code>
9	<code>voucher.nonce > maximum_nonce</code> and <code>voucher.cumulative > floor</code>	<code>VoucherReplay</code>
—	a matured pending decrease is applied, if the vault is not frozen and the operator's collateral is not locked	—
10	<code>delta ≤ line.limit</code>	<code>LineLimitExceeded</code>

The freeze rows exist because a frozen vault cannot transfer and its lines keep a recourse path through the bond. §3 describes freeze and §7 the forced default. On an unfrozen vault with no recourse tail, the validation time is simply the current block time.

What moves

When the vault holds at least the delta and is not frozen, the program pays the delta from the vault, signing as the vault PDA with seeds `[b"vault", operator, agent_id, bump]` :

```

delta = voucher.cumulative - max(settled_cumulative, retired_cumulative)
fee = [ delta × line.tab_fee_bps ÷ 10 000 ]
merchant_token += delta - fee
fee_treasury += fee
vault_token -= delta

```

`line.tab_fee_bps` the rate copied to the line at `open_line`, 0 when the protocol waived fees then

Two `TransferChecked` CPIs (decimals 6) move the amounts; the second is skipped when the fee is zero. The line then records `settled_cumulative = voucher.cumulative` and `maximum_nonce = voucher.nonce`, the operator's `tabs_settled` increases by one, the program emits `CoreEvent kind 8 ( tab_settled )` with the delta as `amount` and the fee as `fee`, and it closes the `DefaultEvent` placeholder, returning its rent to the merchant in the same instruction.

The merchant pays the fee out of the settled amount. The rate is fixed per line at opening, so a governed change to `tab_fee_bps` applies to lines opened afterwards and never reprices a voucher already accepted on an existing line. `update_protocol` cannot raise the rate above `maximum_tab_fee_bps`, which is fixed at initialization and at most 10,000. The devnet parameter file sets `tabFeeBps 10`, `tabFeeWaived true` and `maximumTabFeeBps 100`, so lines opened under it carry `tab_fee_bps = 0` (\$10).

Example. A line opened with `tab_fee_bps = 10` settles a delta of 12,500,000 (12.5 USDC): the fee is $[12,500,000 \times 10 \div 10,000] = 12,500$, the merchant receives 12,487,500 and the fee treasury 12,500. A delta of 999 base units at the same rate carries a fee of 0, because the rate rounds down.

When the vault holds less than the delta, or the vault is frozen and its grace has run, no USDC moves and the voucher is not booked as settled. The program opens the `DefaultEvent` at sequence `default_sequence`, sets `active_default_sequence`, clears any pending bond withdrawal, sets the 259,200-second (72 h) claim window, increments `defaults_count` and emits kind 9 (`default_opened`). There is no partial payment. §7.1 describes this branch.

A full tab

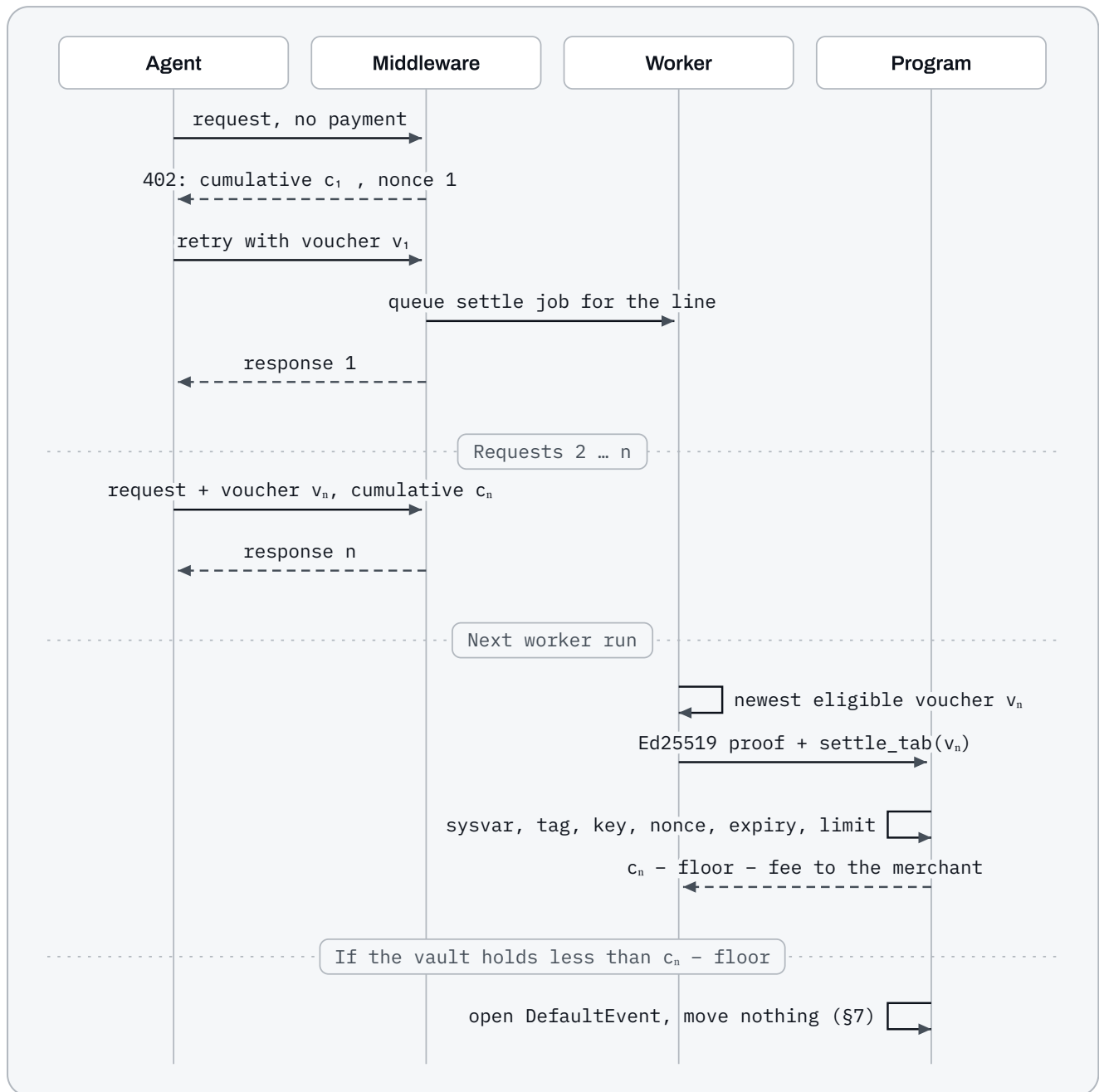


Figure 19. A tab from the first request to a confirmed settlement.

5.7 Transaction size and compute

A settlement is a legacy Solana transaction, limited to 1,232 serialized bytes. The program's own measurement run executes each signed transaction in LiteSVM 0.7.1 (Agave 2.3 runtime) against the compiled program and records its full serialized size, including the signature section, and the compute it consumed.

Signed legacy transaction	Bytes	Compute units	Signatures	Account keys	Fits 1,232 bytes
One voucher, bare; placeholder closes	894	61,044	1	17	yes
One voucher, SDK token-account guard	936	65,382	1	18	yes

Signed legacy transaction	Bytes	Compute units	Signatures	Account keys	Fits 1,232 bytes
One voucher, API token-account guard and preparation memo	1,014	82,363	1	19	yes
One voucher, archived outgoing-key debt (§5.9)	927	68,692	1	18	yes
Voucher attestation during notice (§5.9)	589	17,652	1	8	yes
Two merchants, bare	1,302	122,088	2	20	no
Two merchants, SDK guards	1,354	132,264	2	21	no
One merchant, two operators, bare	1,366	114,588	1	24	no

The compute figures describe this reproducible benchmark. Other actor keys change the cost of PDA searches, so they are measurements of the implementation, not a protocol constant. Against the planning estimates, the bare single-voucher transaction is 894 bytes where roughly 700 were estimated, and 61,044 units where the estimate was well under 100,000.

The 894 bytes break down exactly into the legacy wire format:

Part	Bytes
Signature count and one 64-byte signature	65
Message header	3
Account key count and 17 keys × 32 bytes	545
Recent blockhash	32
Instruction count	1
Ed25519 instruction: program index, account count, 2-byte data length, 208 data bytes	212
<code>settle_tab</code> : program index, account count, 15 account indices, data length, 18 data bytes	36
Total	894

The 17 keys are the 15 accounts of §5.6, the Ed25519 program and the Bonded program. The SDK path adds the associated token program's `CreateIdempotent` instruction for the merchant's token account: 10 instruction bytes plus the program's 32-byte key, 936 in all. The API path also appends an SPL Memo `bonded: <preparation UUID>`, which gives separate preparations distinct messages: a 46-byte instruction plus the memo program's key, 1,014 in all. The merchant worker submits API-prepared transactions: the 1,014-byte, 82,363-unit row plus the two `ComputeBudget` instructions every API preparation puts at the front (a unit limit and a unit price, 52 bytes), 1,066 bytes in all.

A second line costs at least 248 instruction bytes (212 + 36). A second line of a different merchant on the same bond and vault also needs a second signature (64 bytes) and three keys (its merchant, line and token account; the placeholder PDA is shared), 408 bytes in all. A second line of the same merchant must belong to another bond, because one merchant has one line per bond, and that adds seven keys (472 bytes). The bare single-voucher transaction leaves 338 bytes of headroom, so no second line fits, and the measured two-line candidates are 1,302 and 1,366 bytes. Settling two vouchers of the same line is never needed, because the

newer one contains the older. The production planner therefore puts exactly one line in each legacy transaction (`settlementsPerLegacyTransaction: 1`) and checks the fully serialized size before signing and before broadcasting.

One settlement transaction carries one signature, so its network fee is the 5,000-lamport signature fee [27] plus the priority fee its compute budget sets (1,000 micro-lamports per unit by default, about 100 lamports for a settlement), however many requests its voucher covers.

5.8 Batching and settlement cadence

Batching in a tab happens over time, not across lines. Every request on a line folds into the next voucher, and one `settle_tab` collects all of them. A merchant that settles once per minute pays one transaction per line per minute, however many requests the line served in that minute.

The worker

`createMerchantWorker(db, merchant, executor, { leaseMilliseconds, retryMilliseconds, settlementHorizonSeconds })` runs the settlement outbox. The host calls `runOnce()` on its own schedule; no daemon starts on import. Each run has two passes.

- **Discovery.** For every meter of this merchant, the worker reads chain time, the line and the line's defaults. A pending line-key switch in the future queues an `attest_line_voucher` job. A default whose claim window is open queues a `file_claim` job. A line with no unresolved default queues its `settle:<merchant>:<line>` job. A finished job is queued again when a stored voucher with a higher nonce exists.
- **Processing.** Due jobs run in order of kind and then ID, so attestations run before claims and claims before settlements. Each job takes a lease. The worker re-reads the line, picks the newest eligible voucher from the usage table (above the line's nonce and floor, unexpired at its judgment time, signed by an authorized key), prepares the action through the API, and requires a live simulation that succeeds for exactly that action. It checks the 1,232-byte limit and persists the prepared ID. The host's `sign` callback must authorize the original action independently, against the pinned program, genesis hash and trusted account state; the example merchant rebuilds the instructions itself and compares the complete message. The worker then checks the size again, persists the signed bytes, and submits them.

Persisting the signed bytes before submission makes a crash or an ambiguous RPC response safe. The next run resubmits the identical transaction under the same submit key and cannot produce a second, different settlement. A settlement job is closed without action while any default on the line's bond is unresolved, because `settle_tab` would fail with `DefaultActive`. While the default's claim window is open, the line's `file_claim` job runs instead, and settlement resumes after the payout, once the operator has renewed the line into the bond's new epoch (§6.2).

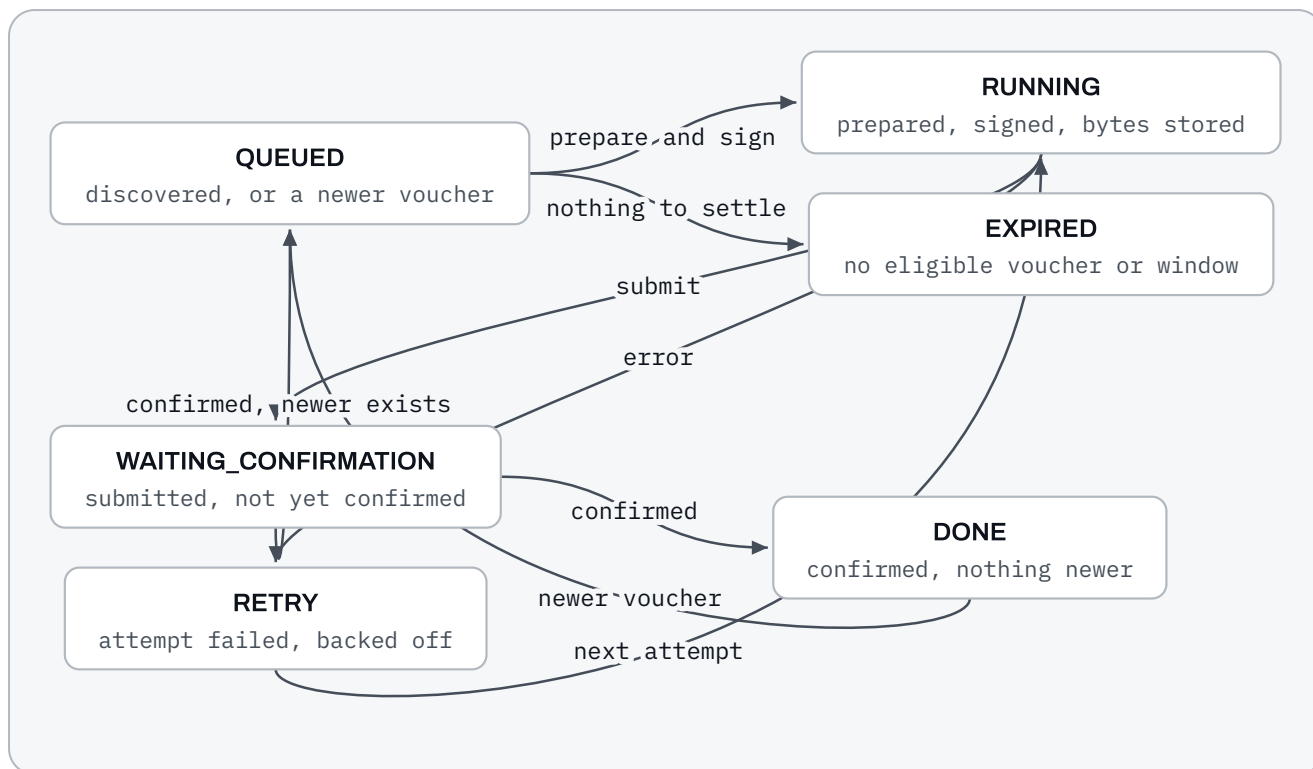


Figure 20. A merchant chain job's states in the settlement worker.

The settlement horizon

The cadence is bounded by one parameter, `settlementHorizonSeconds` : the time the merchant needs to get a voucher on chain, covering the scheduler interval, confirmation and every retry. The middleware and worker require explicit values and reject inconsistent ones at construction.

Parameter	Constraint	Example merchant
<code>settlementHorizonSeconds</code>	positive	30 s
<code>voucherTtlSeconds</code>	greater than $2 \times \text{horizon}$	3,600 s
<code>cacheMilliseconds</code>	at most the horizon	100 ms
<code>rereadIncrement</code>	positive, base units	10,000
<code>serviceLeaseMilliseconds</code>	greater than <code>maximumServeMilliseconds</code>	10,000 ms
<code>maximumServeMilliseconds</code>	positive	2,000 ms
worker <code>retryMilliseconds</code>	below the horizon	1,000 ms
worker <code>leaseMilliseconds</code>	below the horizon	10,000 ms
worker run interval	chosen by the host	1,000 ms

The horizon protects the latest voucher. Earlier vouchers need no protection, because the latest one contains them. The latest voucher must reach the chain before its expiry, since afterwards `settle_tab` refuses it with `VoucherExpired`. The middleware therefore accepts a voucher only if it expires more than one horizon from now, and it pauses credit on the line, answering 409 `DEADLINE_NOT_REACHED` and queuing settlement, as soon as any of three deadlines comes within the horizon:

- the line's `expires_at` ,
- a pending line-key switch, after which the outgoing key's vouchers need an attestation (§5.9),
- the expiry of the newest voucher that is still above the chain floor.

Requiring `voucherTtlSeconds` to exceed twice the horizon gives every accepted voucher more than one horizon of service time before the pause, and one full horizon of settlement time after it. Once the settlement confirms, the floor rises, the outstanding voucher no longer counts, and the next request receives a fresh expiry.

With the Prisma adapter every accepted request re-queues its line's settlement job; the `pg` store's worker finds the line on its next scan. Either way the line settles at the worker's next run. The example merchant runs the worker every second. A merchant that settles less often runs it less often, within the same horizon rules, and the line limit bounds how much it has outstanding in between. The guardian worker leaves tab settlements out of its spend and counterparty rules for this reason: the merchant, not the agent, chooses when a settlement executes.

5.9 Voucher attestations

A line's voucher key is copied from the vault's session key when the line opens, and it changes only through `rotate_line_key` . The admin signs it, and the new key becomes `pending_voucher_key` with `effective_at` = `now` + 259,200 seconds (72 h). Verification switches at exactly that timestamp, with no transaction to apply it. A compromised or retired session key therefore stops creating new debt at a time every party can read on chain, and the merchant has 72 hours of notice. Rotating the vault's session key (`rotate_session_key`) revokes the float delegation and leaves each line's own scheduled rotation unchanged.

The switch raises one question: vouchers the outgoing key signed before the switch and the merchant had not settled yet. After the switch the outgoing key is no longer `voucher_key_at(now)` , so its signatures carry no authority on their own. `attest_line_voucher(ed25519_ix_index)` lets the line's merchant record, during the notice, the newest outgoing-key voucher it holds. That record caps what the outgoing key can be held to afterwards.

The merchant signs, and the voucher arrives in the same canonical Ed25519 proof as for `settle_tab` . The program requires:

- a pending key switch whose `effective_at` is still in the future (`AttestationNoticeClosed` otherwise);
- a signer equal to the line's current, outgoing `voucher_session_key` ;
- $0 < \text{voucher.expiry} \leq \text{line.expires_at}$;
- nonce and cumulative amount above the line's floors, and the delta within the limit;
- if an attestation for this epoch already exists, a nonce and amount at least as high as the recorded ones, so the ceilings only rise.

It writes a `VoucherAttestation` account at `[b"voucher-attestation", line, outgoing_key, effective_at]` (the switch time as `i64` little-endian), funded by the merchant, holding `bump` , `line` , `merchant` , `voucher_key` , `valid_from` (the line's `rotated_at` , the start of the outgoing key's epoch), `effective_at` , `maximum_nonce` , `cumulative_amount` and `attested_at` , and emits `CoreEvent` kind 26 with the attested cumulative amount.

After the switch, `settle_tab` and `file_claim` accept an outgoing-key voucher only when the attestation is passed as their first remaining account and

$$\text{signer} = \text{record.voucher_key} \wedge \text{now} \geq \text{record.effective_at} \wedge \text{reference} \geq \text{record.valid_from}$$

$$\text{voucher.nonce} \leq \text{record.maximum_nonce} \wedge \text{voucher.cumulative} \leq \text{record.cumulative_amount}$$

record the VoucherAttestation PDA, checked for owner, seeds, line and merchant

reference the time the voucher is judged at, now or the line's freeze reference

A fresh signature from the outgoing key, made after the switch, cannot raise the acknowledged debt, because it cannot exceed either ceiling. An unexpired, acknowledged voucher stays payable from the vault after the switch, freeze or no freeze. The attestation moves no funds and revives nothing: settlement still applies every expiry and replay check of §5.6 at its own judgment time. Each rotation's attestation is its own PDA, so later rotations never discard the evidence of earlier epochs.

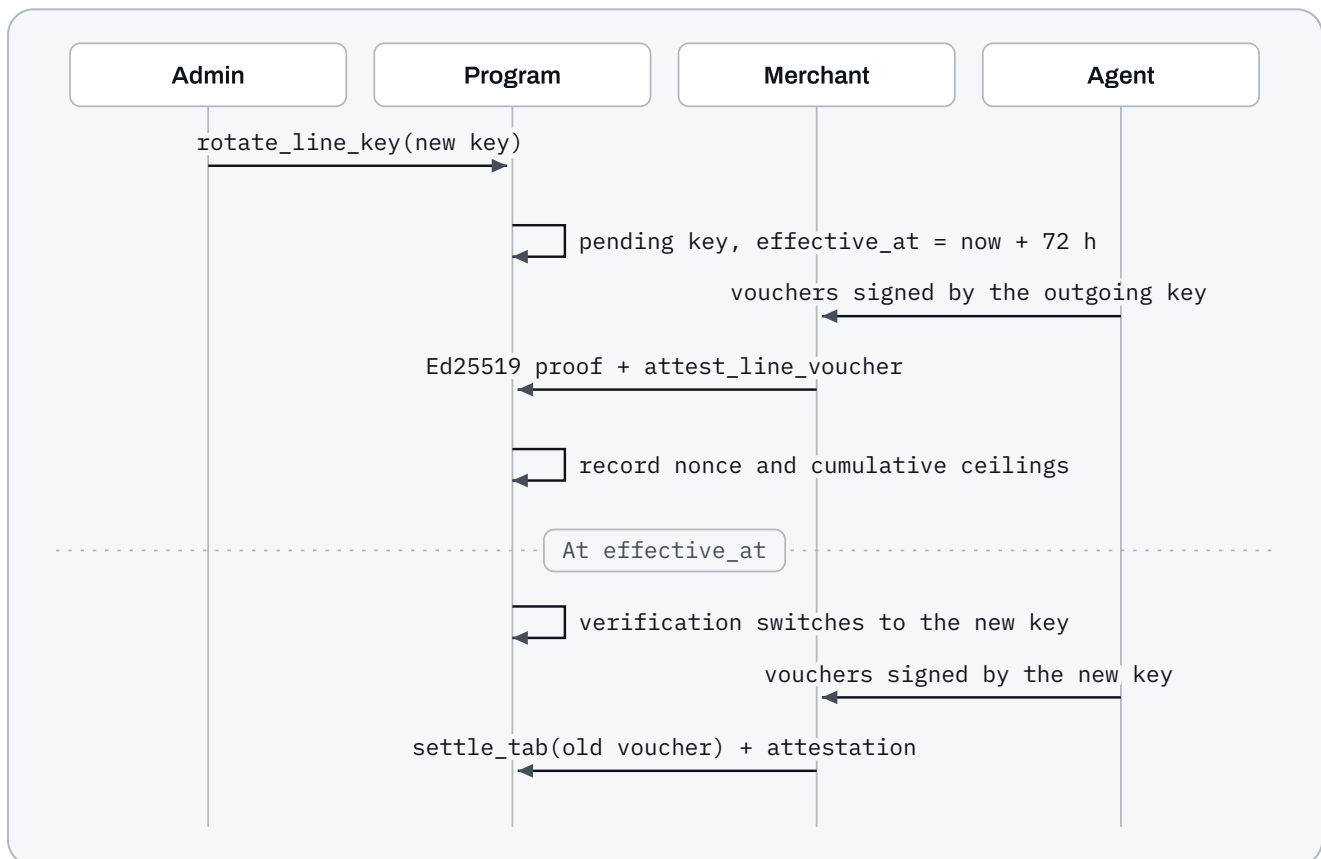


Figure 21. A merchant attests the outgoing key's latest voucher during a line-key rotation notice.

The merchant worker automates this. While a switch is pending, every discovery pass queues an attestation job for the line and re-queues it when a newer outgoing-key voucher arrives. Attestations run first in the processing order, and the middleware pauses credit when the switch comes within the horizon. An attestation transaction is 589 bytes and 17,652 compute units. A settlement that pays archived outgoing-key debt, with the attestation as an extra account, is 927 bytes and 68,692 compute units (§5.7).

6. Bonds and credit lines

The bond is the operator's USDC collateral, held by a program-owned account. Credit lines are the merchant-specific limits that collateral backs. A line gives one merchant the right to accept vouchers from one vault up to its limit until its expiry, with the bond standing behind the vault if the vault cannot pay. The bond pays nothing while the vault settles tabs. It pays only through a default event (§7), and then to every line holder under one formula. The rules in this chapter keep that promise checkable in advance: how much collateral the bond books, how much exposure it may back, how fast either can change, and when collateral may leave.

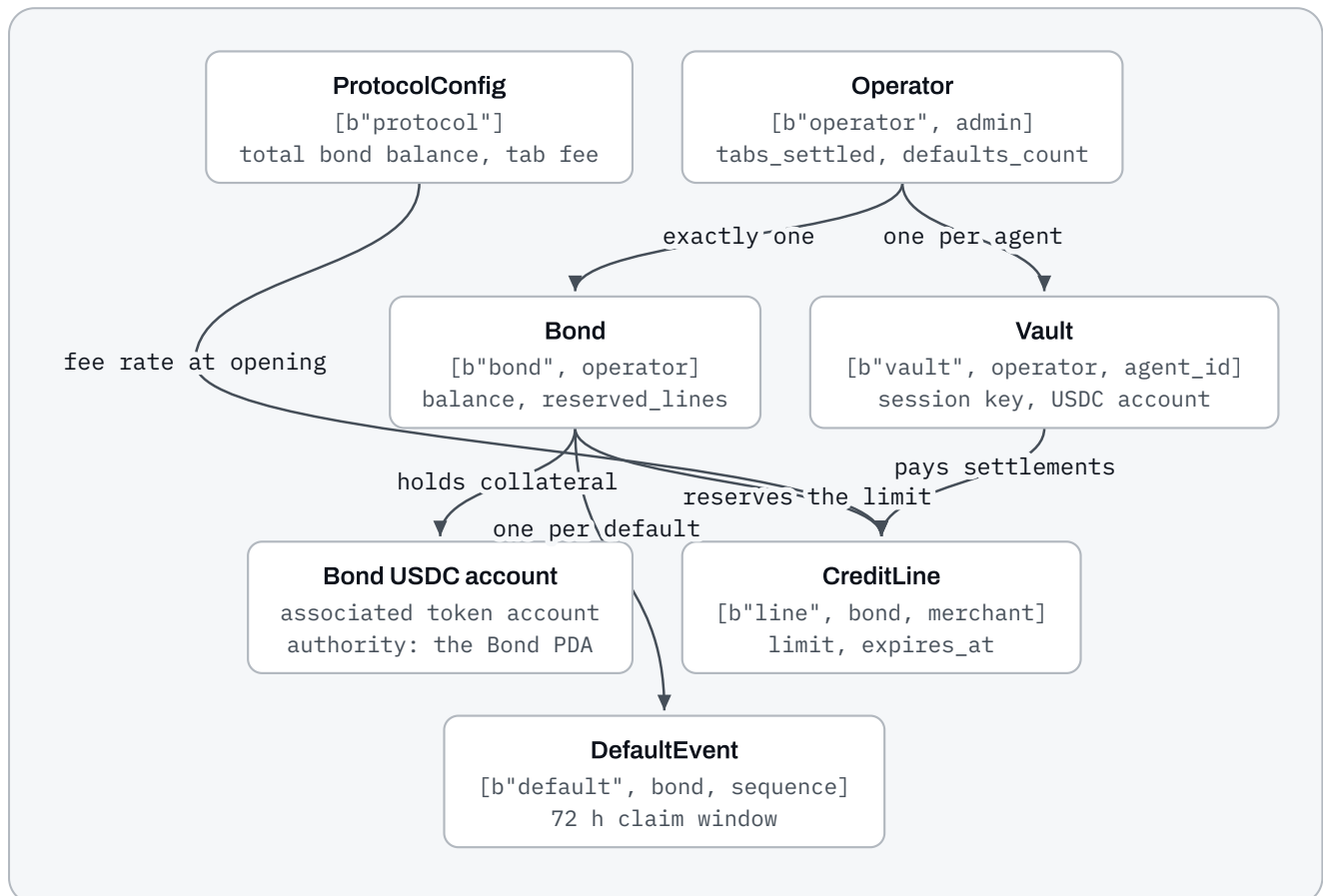
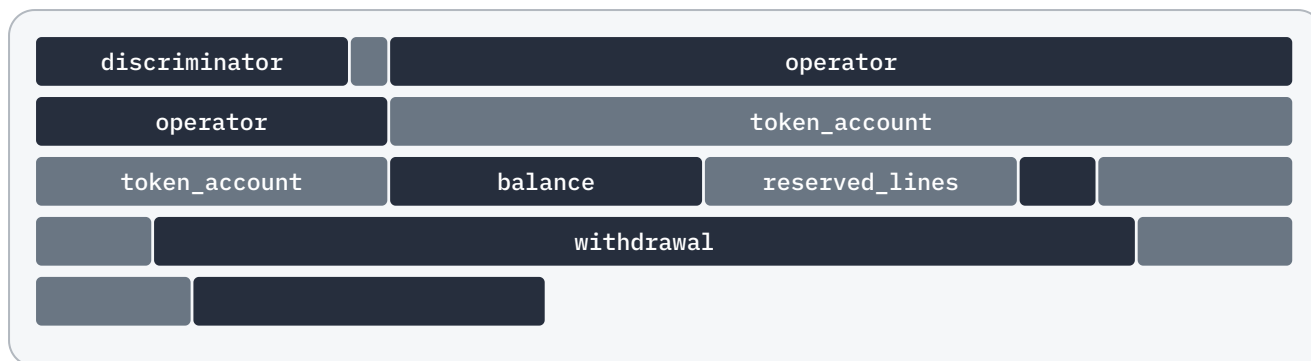


Figure 22. The accounts behind a credit line and the seeds that derive them.

6.1 The Bond account and post_bond

An operator has exactly one bond. `create_operator(guardian, minimum_coverage_bps)`, signed and paid for by the admin key, creates three accounts in one instruction: the **Operator** at `[b"operator", admin]`, the **Bond** at `[b"bond", operator]`, and the bond's USDC associated token account, owned by the **Bond PDA**. It accepts a correctly created token account that already exists. It refuses a `minimum_coverage_bps` below 3,300 with `CoverageTooLow` and emits `CoreEvent` kind 1. The bond backs every line the operator opens, across all of the operator's vaults.



141 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	first 8 bytes of SHA-256("account:Bond")
8	1	bump	PDA bump for [b"bond", operator]
9	32	operator	the Operator PDA
41	32	token_account	the bond's USDC token account
73	8	balance	u64, booked collateral, base units
81	8	reserved_lines	u64, Σ limit of the bond's lines
89	2	minimum_coverage_bps	u16, fixed at create_operator
91	8	longest_line_expiry	i64, high-water mark of line expiries
99	25	withdrawal	Option tag, amount u64, requested_at, unlock_at
124	8	default_sequence	u64, defaults opened on this bond
132	9	active_default_sequence	Option tag + u64

Figure 23. The 141-byte Bond account as the program writes it while a withdrawal is pending.

The offsets after `withdrawal` assume a pending request. Without one, its Borsh `Option` tag is a single zero byte and `default_sequence` and `active_default_sequence` start 24 bytes earlier, at 100 and 108. The account is allocated at its maximum size of 141 bytes.

Field	Written by	Meaning
balance	post_bond (+), finalize_withdrawal (-), execute_default_payout (-)	the collateral every coverage check and payout reads
reserved_lines	open_line (+), resize_line ($\pm$), settle_tab (applies a matured decrease)	the sum of the current limits of every line on the bond
minimum_coverage_bps	create_operator only	the operator's coverage floor (§6.3)
longest_line_expiry	open_line , renew_line	the latest <code>expires_at</code> any line has had; it never moves earlier

Field	Written by	Meaning
<code>withdrawal</code>	<code>request_withdrawal</code> , <code>finalize_withdrawal</code> ; cleared when a default opens	the pending request: <code>amount</code> , <code>requested_at</code> , <code>unlock_at</code>
<code>default_sequence</code>	<code>settle_tab</code> when a default opens	the number of defaults opened, and the current line epoch
<code>active_default_sequence</code>	<code>settle_tab</code> ; cleared by the last payout or by <code>finalize_empty_default</code>	the open default; while set, <code>open_line</code> , <code>resize_line</code> , <code>renew_line</code> , both withdrawal steps and <code>settle_tab</code> fail with <code>DefaultActive</code>

`post_bond`

`post_bond(amount)` is signed by the admin. It transfers `amount` from a USDC token account the admin owns into the bond's token account with `TransferChecked` (decimals 6), then books it:

$$\text{bond.balance}' = \text{bond.balance} + \text{amount} \leq \text{BOND_CAP} = 25\,000\,000\,000$$

$$\text{protocol.total_bond_balance}' = \text{protocol.total_bond_balance} + \text{amount} \leq \text{TOTAL_CAP} = 250\,000\,000\,000$$

<code>amount</code>	greater than 0, USDC base units
<code>BOND_CAP</code>	25,000 USDC per bond, a program constant
<code>TOTAL_CAP</code>	250,000 USDC across all bonds, a program constant

A deposit that would pass either cap fails with `BetaCapExceeded` , and the instruction emits `CoreEvent` kind 4 (`bond_posted`). `post_bond` checks no coverage, freeze or default state. Adding collateral can only raise coverage, so the program never refuses it on those grounds. Collateral posted while a default is open joins the payout pool, which is snapshotted at the first payout (§7.3).

The bond's collateral is the booked `balance` , not the token account's amount. USDC that reaches the bond's token account by any other path is not booked. It counts toward no coverage check, no cap and no payout, and the protocol-wide total that enforces the 250,000 USDC cap stays exact. The token account's authority is the `Bond` PDA, and the program signs for it with `[b"bond", operator, bump]` in two places only: `finalize_withdrawal` , after the cooldown of §6.5, and `execute_default_payout` , to the merchant recorded in a claim.

6.2 `open_line` and the `CreditLine` account

`open_line(limit: u64, expires_at: i64)` gives one merchant credit against the bond. The admin signs and pays the line's rent. The merchant is passed as an unsigned account. A line obliges the operator's bond, not the merchant, and the merchant decides request by request whether to use it (§5.5).

The line is the PDA `[b"line", bond, merchant]`, so a bond has at most one line per merchant. The line also names one of the operator's vaults, and every settlement on it draws from that vault. The checks run in this order:

1. The vault is not frozen (`VaultFrozen`).
2. No default is active on the bond (`DefaultActive`).
3. `limit > 0` and `expires_at > now` (`InvalidAmount`).
4. Coverage holds with the new limit added: $(\text{balance} - \text{pending withdrawal}) \times 10,000 \geq (\text{reserved_lines} + \text{limit}) \times \text{minimum_coverage_bps}$ (`CoverageTooLow`, §6.3).

Then `reserved_lines` grows by `limit`, `longest_line_expiry` becomes the later of itself and `expires_at`, and a pending withdrawal's `unlock_at` moves later if the new expiry requires it (§6.5). The line is initialized and `CoreEvent` kind 5 (`line_opened`) is emitted with the limit as its amount.

Field	Type	Set at opening	Role
<code>bond</code> , <code>vault</code> , <code>merchant</code>	Pubkey	from the accounts	bind settlements and claims to one bond, vault and merchant
<code>voucher_session_key</code>	Pubkey	the vault's current <code>session_key</code>	the key whose vouchers the line accepts (§5.9)
<code>limit</code>	u64	<code>limit</code>	the most the line may owe above its floor; the claim cap in a default
<code>settled_cumulative</code>	u64	0	cumulative amount of the last settled voucher
<code>maximum_nonce</code>	u64	0	nonce of the last settled or paid voucher
<code>expires_at</code>	i64	<code>expires_at</code>	vouchers must expire no later than this, and be judged before it
<code>pending_limit</code>	<code>Option<{value, effective_at}></code>	None	a scheduled decrease (§6.4)
<code>opened_default_sequence</code>	u64	<code>bond.default_sequence</code>	the default epoch the line belongs to
<code>pending_voucher_key</code>	<code>Option<{value, effective_at}></code>	None	a scheduled key rotation (§5.9)
<code>retired_cumulative</code>	u64	0	voucher amount already resolved by a default payout
<code>opened_at</code>	i64	now	original creation time, kept through renewals
<code>freeze_generation_at_open</code>	u64	the vault's <code>freeze_generation</code>	the reference for freeze recourse (§7)
<code>tab_fee_bps</code>	u16	the protocol rate, or 0 when waived	the line's settlement fee, immutable
<code>previous_voucher_key</code> , <code>rotated_at</code>	<code>Option<Pubkey></code> , i64	None, 0	the key before the last switch and when it switched

The fixed part of the layout runs from the discriminator through `expires_at` at offset 161. `pending_limit` starts at offset 169, and the fields after it follow the `Option` encoding. The account is allocated at 302 bytes. At the rent formula $(\text{size} + 128) \times 3,480 \times 2$ lamports [28], that is 2,992,800 lamports, paid by the admin. Lines are never closed: a line is a persistent record whose replay floors (`maximum_nonce`, `settled_cumulative`, `retired_cumulative`) must outlive any one epoch.

`reserved_lines` equals the sum of `limit` over the bond's lines at all times. Opening adds the limit, a raise adds the difference, an applied decrease subtracts it, and nothing else writes either side. Expired lines stay in the sum until their limits are decreased, so reserved exposure is never understated.

Expiry and renewal

A line's expiry bounds every voucher on it: `settle_tab` requires the voucher's expiry to be no later than `expires_at` and the judgment time to be before it (§5.6). It also bounds the bond's withdrawal cooldown (§6.5). Expiry can only move later. `renew_line(expires_at)` requires the vault not frozen, no active default, the operator's collateral not locked by a freeze, an expiry after now and no earlier than the current one, and coverage at the current reservation. It sets the new expiry, moves the line into the bond's current default epoch, raises `longest_line_expiry` and any pending `unlock_at` as `open_line` does, and emits kind 17. Limit, key, fee, `opened_at`, freeze generation and every replay floor stay as they were, so a renewed line cannot collect old debt a second time.

The default epoch is how a default retires lines. When a default opens, `bond.default_sequence` advances and every line's `opened_default_sequence` falls behind. Those lines can no longer settle and can only claim against that default: `settle_tab` refuses them with `LineExpired`, and only `renew_line` brings a line into the new epoch once the default is paid (§7.1).

6.3 The coverage ratio

The coverage ratio is the bond's collateral divided by the exposure it backs. The program enforces it as an exact integer inequality, with `u128` products and no division, at every point where exposure grows or collateral leaves.

$$\text{collateral} \times 10\,000 \geq \text{reserved} \times \text{minimum_coverage_bps}$$

$$\text{coverage_bps} = \lfloor \text{collateral} \times 10\,000 \div \sum \text{line.limit} \rfloor$$

$$\sum \text{line.limit} \leq \lfloor \text{collateral} \times 10\,000 \div \text{minimum_coverage_bps} \rfloor$$

<code>collateral</code>	<code>bond.balance</code> – pending withdrawal amount; for a withdrawal, <code>bond.balance</code> – amount
<code>reserved</code>	<code>bond.reserved_lines</code> after the change being checked
<code>minimum_coverage_bps</code>	the operator's floor, at least 3 300, set at <code>create_operator</code>
<code>coverage_bps</code>	the published ratio; null when the bond has no lines

The first line is the program's check, `coverage(balance, reserved, minimum)`. The second is the published figure. `calculateCoverageRatio(bondBalance, lineLimits, minimumBps)` in the shared contract package returns the floored basis points, or `null` when there are no lines, together with `meetsMinimum`, which is the same inequality as the program's. The API computes it with collateral net of any pending withdrawal, the same figure the program uses for new exposure. The third line is the most exposure a given collateral admits.

Instruction	Collateral	Reserved	Fails with
<code>open_line</code>	balance – pending withdrawal	<code>reserved_lines</code> + limit	<code>CoverageTooLow</code>
<code>resize_line</code> , raise or same value	balance – pending withdrawal	<code>reserved_lines</code> – old limit + new limit	<code>CoverageTooLow</code>
<code>renew_line</code>	balance – pending withdrawal	<code>reserved_lines</code>	<code>CoverageTooLow</code>
<code>request_withdrawal</code>	balance – amount	<code>reserved_lines</code>	<code>CoverageTooLow</code>
<code>finalize_withdrawal</code>	balance – amount	<code>reserved_lines</code>	<code>CoverageTooLow</code>
<code>post_bond</code> , <code>resize_line</code> decrease, <code>settle_tab</code> , <code>execute_default_payout</code>	not checked	—	—

The unchecked instructions either raise collateral, lower exposure, or pay out what the bond already owes. A default payout can leave coverage below the minimum. The bond then accepts no new line, raise, renewal or withdrawal until collateral is posted or reservations are decreased.

The operator's minimum

`minimum_coverage_bps` is chosen once, at `create_operator`, and no instruction changes it. The program's floor is `COVERAGE_FLOOR_BPS` = 3,300 (33%). The field is a `u16`, so a minimum can go up to 65,535 bps (655.35%) for an operator that wants to publish more than full collateral. `COVERAGE_DEFAULT_BPS` = 10,000 (100%) is the value the transaction builder writes when the operator does not choose one. At 10,000 bps the sum of limits can never exceed the collateral. At the 3,300 floor it can reach $\lfloor \text{collateral} \times 10,000 \div 3,300 \rfloor$, about 3.03 times the collateral: a bond of 25,000,000,000 base units (25,000 USDC) can back at most 75,757,575,757 base units of limits.

A line that would take coverage below the operator's minimum is not opened. `open_line` fails with `CoverageTooLow` (program error 6005, "Coverage would fall below the operator minimum."), the transaction writes nothing, and the API's preparation, which simulates every transaction before returning it for signing, reports `COVERAGE_TOO_LOW` before anything is signed.

Worked example

Example. An operator posts 25,000 USDC, the per-bond cap, and chose a minimum of 5,000 bps (50%). Amounts are in USDC; the program compares the same quantities in base units, each side scaled by 10^6 .

Step	Instruction	Reserved after	Check: $\text{collateral} \times 10,000 \geq \text{reserved} \times 5,000$	Result	Published ratio
1	<code>open_line</code> A, 20,000	20,000	$250,000,000 \geq 100,000,000$	opened	12,500 bps

Step	Instruction	Reserved after	Check: collateral × 10,000 ≥ reserved × 5,000	Result	Published ratio
2	open_line B, 15,000	35,000	250,000,000 ≥ 175,000,000	opened	7,142 bps
3	open_line C, 10,000	45,000	250,000,000 ≥ 225,000,000	opened	5,555 bps
4	open_line D, 6,000	—	250,000,000 < 255,000,000	CoverageTooLow	5,555 bps
5	open_line D, 5,000	50,000	250,000,000 = 250,000,000	opened	5,000 bps
6	request_withdrawal 1,000	50,000	240,000,000 < 250,000,000	CoverageTooLow	5,000 bps
7	resize_line D to 3,000	50,000	not checked	decrease scheduled	5,000 bps
8	resize_line D to 3,000, 72 h later	48,000	not checked	applied	5,208 bps
9	request_withdrawal 1,000	48,000	240,000,000 ≥ 240,000,000	requested	5,000 bps

After step 9 the pending 1,000 USDC no longer counts as collateral for new exposure, so the published ratio is $24,000 \div 48,000$. The bond is at its minimum, and any further line or raise fails until the operator posts more or decreases a line.

The ratio is the share of its limit each merchant would recover if every line claimed its full limit at once, and the published figure is a lower bound on that share. Opening a default cancels the pending withdrawal (§6.5), so here the pool would be the full 25,000 USDC against eligible claims of 48,000 USDC, and each claim would receive $[\text{pool} \times \text{eligible} \div \text{total}]$, about 52% of its limit (§7.3). At 10,000 bps or more, every claim up to its limit is paid in full.

6.4 resize_line

`resize_line(limit)` is signed by the admin and changes one line's limit. Raises take effect at once. Decreases take effect only after a 72-hour notice, `DECREASE_DELAY` = 259,200 seconds, which is visible on chain from the moment it is scheduled. Any call fails with `DefaultActive` while a default is open on the bond.

Call	Conditions	Effect	Event
new limit > current	vault not frozen; line in the current epoch and unexpired; coverage holds	limit and reserved_lines raised at once; any pending decrease cancelled	kind 7
new limit = current	line in the current epoch and unexpired; coverage holds	pending decrease cancelled	kind 7
new limit < current, not the pending value	none beyond the default check	<code>pending_limit = {limit, now + 259,200}</code> , replacing any earlier pending value	kind 6
new limit < current, equal to the pending value	<code>now ≥ effective_at (DecreaseDelayNotReached)</code> ; collateral not locked by a freeze (<code>FrozenRecourseActive</code>)	limit lowered, reserved_lines reduced by the difference, pending cleared	kind 7

A matured decrease is also applied by the next `settle_tab` on the line that passes its voucher checks, provided the vault is not frozen and the operator's collateral is not locked, so it does not wait for the admin to repeat the call. The decrease is applied before the delta is compared with the limit. Decreases carry no expiry check, so an expired line can release its reservation through the same schedule.

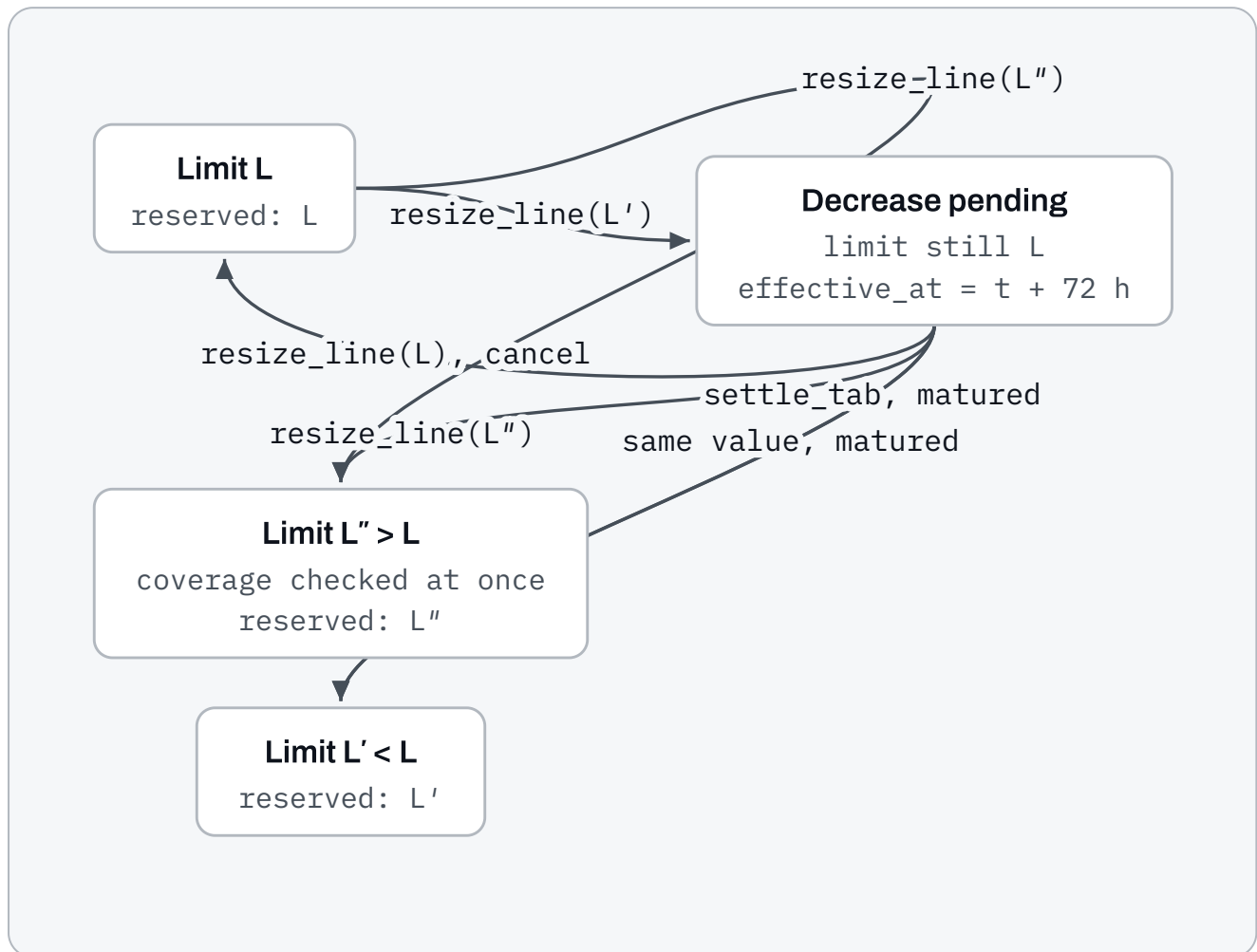


Figure 24. How `resize_line` moves a credit line's limit.

Why decreases wait

A merchant extends credit against the limit it reads. An instant decrease could cut a line below what the merchant had already served, and in a default the merchant's claim is capped by the limit. The notice period protects the credit already extended in three ways:

- **The merchant sees the change coming.** `pending_limit` and its `effective_at` are public. The middleware and the agent SDK both use the lower value from `effective_at` onward, even before the chain applies it (§5.4), and in the meantime the merchant settles its outstanding vouchers so that what it is owed fits the new limit.
- **The reservation stays.** `reserved_lines` keeps backing the old limit until the decrease applies. The operator cannot free collateral for a withdrawal or another line by announcing a decrease.
- **A default judges the limit at its opening.** When the operator's collateral was not locked at opening, a claim is measured against `effective_limit_at(opened_at)`, the limit in force when the default opened, with a pending decrease counted only if it had already matured. When the operator's collateral was locked by a

freeze at opening, or a freeze reference applies, the claim uses the stored `limit` . An unapplied decrease never reduces it (§7.2).

A raise reduces nothing a merchant relied on. It is checked for coverage at the moment it applies, so it needs no notice. A line's voucher key follows the same 72-hour schedule for the same reason: the merchant must be able to see it change before it does (§5.9).

6.5 Bond withdrawals

Collateral leaves the bond in two steps, `request_withdrawal(amount)` and `finalize_withdrawal()` , separated by a cooldown that outlasts every default the bond's current lines could still open. The unlock time is fixed by a formula, not chosen by the operator:

$$\text{unlock_at} = \max(\text{now}, \text{longest_line_expiry}) + \text{CLAIM_WINDOW} + \text{withdrawal_margin_seconds}$$

$$\text{unlock_at}' = \max(\text{unlock_at}, \max(\text{now}, \text{longest_line_expiry}') + \text{CLAIM_WINDOW} + \text{withdrawal_margin_seconds})$$

<code>now</code>	the request time; in the second line, the time a line is opened or renewed
<code>longest_line_expiry</code>	the latest expires_at any line of the bond has had
<code>CLAIM_WINDOW</code>	259 200 s (72 h), a program constant
<code>withdrawal_margin_seconds</code>	set at initialize_protocol, must be positive; 86 400 s (24 h) in the devnet and mainnet-beta parameter files

With the parameter files' margin, `unlock_at` is 345,600 seconds (96 h) after the later of the request time and the longest line expiry.

`request_withdrawal(amount)`

The admin signs. The program requires the operator's collateral not to be locked by a freeze, meaning no frozen vault and the time past `recourse_lock_until` (`FrozenRecourseActive`). It also requires no active default (`DefaultActive`), no pending request (`WithdrawalExists`), $0 < \text{amount} \leq \text{balance}$ (`InvalidAmount`), and coverage with `balance - amount` against the current reservation (`CoverageTooLow`). It stores `{amount, requested_at: now, unlock_at}` and emits kind 13. From then on the requested amount is excluded from the collateral that backs new lines, raises and renewals (§6.3).

`finalize_withdrawal()`

The admin signs and names the destination token account, which may belong to any wallet but must hold the configured USDC mint under classic SPL Token. The program again requires no freeze lock and no active default. It requires a request (`NoWithdrawal`), $\text{now} \geq \text{unlock_at}$ (`WithdrawalCoolingDown`), and coverage with `balance - amount` . It then transfers `amount` from the bond's token account, signing as the bond PDA, lowers `balance` and the protocol total, clears the request and emits kind 14.

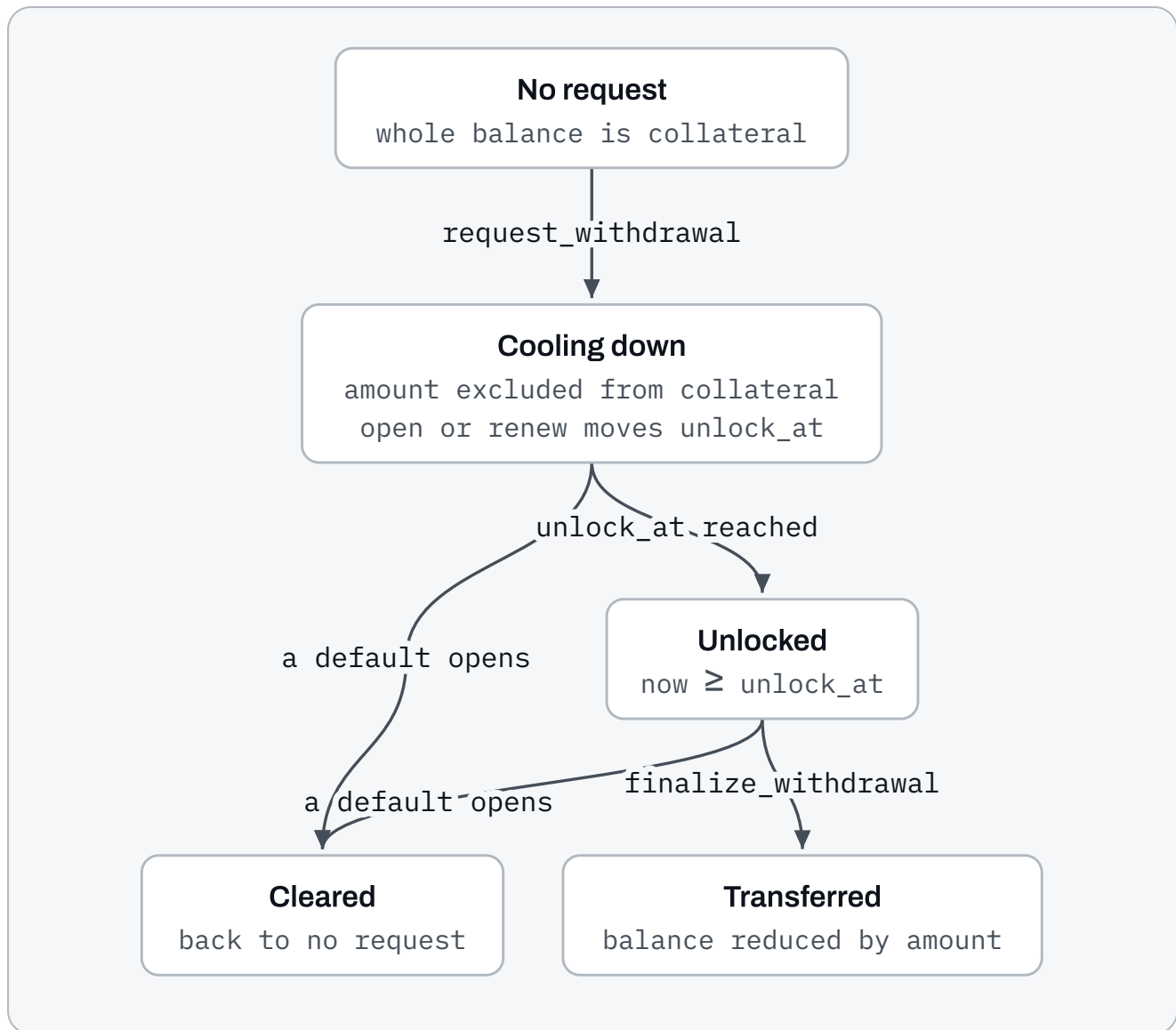


Figure 25. A bond withdrawal from request to transfer.

Why the formula prevents an exit ahead of a default

A default against the bond opens only inside `settle_tab`, for a voucher judged valid on a line of the current epoch at a validation time before that line's `expires_at` (§5.6). Unless a freeze reference applies, the validation time is the moment the default opens. Every default the current lines can open therefore opens before `longest_line_expiry`, a high-water mark the program never lowers. A default opened at time t accepts claims until $t + \text{CLAIM_WINDOW}$, so the latest claim window the current lines can produce ends before `longest_line_expiry + CLAIM_WINDOW`, and `unlock_at` lies at least `withdrawal_margin_seconds` beyond that. The margin is positive, so the boundary is strict. The freeze-reference case is covered below.

Three further rules close the remaining paths:

- **A default cancels the request.** When `settle_tab` opens a default it sets `bond.withdrawal = None`, so a request made before the default can never be finalized after it. The operator requests again once the payout completes, against the residual balance and with a newly computed unlock time.
- **An open default blocks both steps.** While `active_default_sequence` is set, `request_withdrawal` and `finalize_withdrawal` fail with `DefaultActive`, and the pool the claims share cannot shrink before the payout snapshot.

- **New exposure extends the cooldown.** `open_line` and `renew_line` raise `longest_line_expiry` and push a pending `unlock_at` later by the same formula, so a line added during the cooldown is covered by it.

A frozen vault can open a default after its expiry, through the line's freeze reference. That path is guarded by the collateral lock: while any of the operator's vaults is frozen, and until `recourse_lock_until` after unfreeze (the unfreeze time plus `freeze_grace_seconds` plus 72 h), neither withdrawal step runs (§3, §7). The program does not try to prove that a line's exposure ended early. A line decreased to zero still holds `longest_line_expiry` where it was, and the cooldown waits for the latest expiry the bond has ever recorded.

Example. A bond's lines expire 30 days from now, and the operator wants all of its collateral back. Times are in days from now, and the margin is the parameter `files`' 24 hours.

Day	Instruction	Result
0	<code>resize_line</code> to 0 on every line	decreases scheduled, effective day 3
3	<code>resize_line</code> to 0 again on every line	<code>reserved_lines</code> = 0
3	<code>request_withdrawal(balance)</code>	coverage holds with 0 reserved; <code>unlock_at</code> = $\max(3, 30) + 3 + 1$ = day 34
34	<code>finalize_withdrawal()</code>	the full balance is transferred

A full withdrawal needs the reservations released first, because `request_withdrawal` checks coverage with `balance - amount` against `reserved_lines`. Between day 0 and day 3 the old limits still apply. A settlement that found a vault short in that interval would have opened a default against the full bond, and no request could have been made until its payout completed. From day 3 the limits are zero, so no voucher can pass the delta check. The cooldown still runs to day 34, because the high-water mark stays at day 30.

6.6 Public coverage as the merchant's credit signal

A merchant decides whether to accept a line, and how much to serve on it, from public state. Every figure below is an account field written only by the program, or a value derived from one, and anyone can read it. The API publishes an operator's aggregates as a public profile, `PublicOperatorProfile { operator, coverage, lineCount, totalLineLimit }`, and returns that profile with every line a merchant views.

Signal	Source	What it tells the merchant
Coverage ratio, minimum, collateral net of a pending withdrawal	<code>Bond.balance</code> , <code>withdrawal</code> , <code>reserved_lines</code> , <code>minimum_coverage_bps</code>	the share of every line's limit the bond would pay if all lines claimed at once; the floor the operator cannot go below
Line count and total line limit	the bond's <code>CreditLine</code> accounts	how many merchants share the same pool, and for how much
Pending withdrawal and its <code>unlock_at</code>	<code>Bond.withdrawal</code>	collateral that is leaving, and from when
Defaults opened, open default	<code>Bond.default_sequence</code> , <code>active_default_sequence</code>	whether the bond is in a default now, and how many it has had
Tabs settled	<code>Operator.tabs_settled</code>	successful <code>settle_tab</code> settlements; each one may cover many requests
Defaults	<code>Operator.defaults_count</code>	default events opened against the operator's bond

Signal	Source	What it tells the merchant
Days active	<code>[(block time - Operator.created_at) ÷ 86,400]</code> , computed by the indexer	the operator's tenure
Freeze state	<code>Operator.frozen_vault_count</code> , <code>recourse_lock_until</code> , <code>Vault.frozen</code>	whether collateral is locked, and whether the line's vault can pay
The line itself	<code>limit</code> , <code>pending_limit</code> , <code>expires_at</code> , <code>voucher_session_key</code> , <code>pending_voucher_key</code> , <code>tab_fee_bps</code>	the merchant's own claim cap, scheduled changes, and the settlement fee
Settleable now	unsettled voucher amount, vault USDC balance	whether the next settlement would be paid by the vault

The counters are the operator's record. They live in the `Operator` PDA, which is derived from the admin key and cannot be transferred, and `settle_tab` is the only instruction that writes them: `tabs_settled` increments in every healthy settlement, and `defaults_count` in every settlement that opens a default. The operator has no instruction to set or reset either one.

Coverage is a property of the whole bond, not of one line, because every line on the bond shares one payout pool. A merchant's recovery in a default is the smaller of what it is owed above the floor and its own limit, scaled down pro-rata when the bond's claims exceed its balance (§7.3). The total limit of all lines is therefore part of each merchant's terms. Lines are public accounts, so a merchant sees every other line that would share the pool, including lines the operator opens after the merchant's own. The program then enforces the published minimum on every future change: whatever lines are added, coverage cannot fall below `minimum_coverage_bps` through `open_line` , `resize_line` , `renew_line` or a withdrawal. A merchant that reads a minimum of 5,000 bps knows that new lines, raises and withdrawals can bring coverage down to 5,000 bps and no further. Only a default payout takes it lower. `reserved_lines` keeps counting expired lines until their limits are decreased, so the published ratio errs low rather than high.

The middleware applies the operational part of this signal on its own. It refuses new credit on a line whose vault is frozen, whose line has expired or been retired by a default, or whose limit, effective decrease included, would be exceeded (§5.5). Beyond that, the ratio, the counters and the line terms are the inputs to the merchant's own decision to extend credit.

7. Defaults and claims

A tab is credit the merchant extends. The merchant serves requests first and collects later, against cumulative vouchers the agent's session key signed, and the vault pays each settlement. The default path handles the one case where the vault does not pay: `settle_tab` finds less USDC in the vault than the voucher asks for, or finds the vault frozen past its grace period. From that moment the bond pays instead of the vault, and it pays every line holder of that bond under one formula instead of whichever merchant reaches the program first.

Four instructions carry the path. `settle_tab` opens a `DefaultEvent`. During a fixed claim window each line holder calls `file_claim` with its latest voucher. After the window anyone calls `execute_default_payout`, once per claim, and the bond pays each claim its eligible amount in full, or pro-rata when the claims together exceed the bond. If no one filed, `finalize_empty_default` closes the event. Every step is decided by signatures, stored account state and the cluster clock. No step asks anyone for a judgment, so the outcome of a default can be computed in advance from public accounts, and anyone can drive it to completion.

7.1 When `settle_tab` finds the vault short

`settle_tab(default_sequence: u64, ed25519_ix_index: u16)` is signed by the merchant that holds the line; §5.6 describes its healthy path in full. The instruction immediately before it is an Ed25519 precompile instruction carrying the 96-byte voucher (§5.2), the key that signed it and the signature. The program reads that instruction through the instructions sysvar and accepts only the canonical 208-byte form with the fixed offsets header, so the voucher it evaluates is exactly the one the runtime verified.

The handler runs its checks in a fixed order, and the vault balance is the last thing it looks at:

1. The voucher is decoded, bound to this line and to the protocol's voucher domain, and its signer is checked against the line's voucher key. For a frozen vault, the key and validity are judged at the line's freeze reference instead of the current time.
2. If the vault is frozen, its accumulated frozen time in the current freeze episode must have reached `freeze_grace_seconds`, otherwise the call fails with `VaultFrozen`.
3. The bond has no active default (`DefaultActive`), and `default_sequence` equals `bond.default_sequence + 1` (`InvalidDefault`).
4. The line belongs to the bond's current epoch, `line.opened_default_sequence == bond.default_sequence`, and has not expired at the validation time (`LineExpired`).
5. The voucher's expiry lies after the validation time and no later than the line's expiry (`VoucherExpired`).
6. Both the nonce and the cumulative amount advance past the line's replay floors (`VoucherReplay`).
7. A pending limit decrease that has matured is applied, provided the operator's collateral is not locked by a freeze.
8. The amount the voucher adds, `delta`, must fit within the line limit (`LineLimitExceeded`).

Only then does the program compare the vault's token balance with `delta`.

$$\text{delta} = \text{voucher.cumulative} - \max(\text{line.settled_cumulative}, \text{line.retired_cumulative})$$

$$\text{default opens} \Leftrightarrow \text{vault.frozen} \vee \text{vault_token.amount} < \text{delta}$$

$$\text{window_end} = \text{opened_at} + 259\,200$$

<code>voucher.cumulative</code>	total the session key has vouched for on this line, USDC base units
<code>line.settled_cumulative</code>	cumulative amount of the last voucher the vault paid
<code>line.retired_cumulative</code>	highest voucher amount already resolved by an earlier default
<code>vault_token.amount</code>	the vault's USDC balance, including any undrawn float allowance

Because every validity check runs before the balance comparison, only a voucher that would have been payable can open a default: correctly signed, unexpired, advancing, and within the line limit. A merchant cannot turn a voucher the vault would have refused for another reason into a claim on the bond.

What moves

On the healthy path the vault transfers `delta - fee` to the merchant and `fee` to the fee treasury, the line records the voucher as settled, and the operator's `tabs_settled` counter increments. On the default path no USDC moves at all. The vault keeps its balance, the line keeps its settled amount, and the instruction writes the event instead.

State	Healthy settlement	Default opens
Vault USDC	<code>delta - fee</code> to the merchant, <code>fee</code> to the fee treasury	unchanged
<code>line.settled_cumulative</code> , <code>line.maximum_nonce</code>	set to the voucher's cumulative amount and nonce	unchanged
<code>operator.tabs_settled</code>	+1	unchanged
<code>operator.defaults_count</code>	unchanged	+1
<code>bond.default_sequence</code>	unchanged	set to <code>default_sequence</code>
<code>bond.active_default_sequence</code>	None	Some(<code>default_sequence</code>)
<code>bond.withdrawal</code>	unchanged	cleared to None
DefaultEvent PDA	created and closed in the same instruction, rent back to the merchant	created, filled and kept
CoreEvent emitted	kind 8, amount <code>delta</code> , <code>fee</code>	kind 9, amount <code>delta</code>

There is no partial settlement. If the vault paid part of a delta, the first merchant to settle against a draining vault would take what remained and the next would meet an empty account, and the line would record a settled amount that no voucher states. Opening the event instead removes the race: every line holder of the bond becomes a claimant on the same terms, and only the bond takes part in the payout. The vault's remaining balance stays in the vault.

The forced path exists for a frozen vault. A frozen vault cannot transfer, so while the freeze lasts the bond is the only source that can pay a line holder. `settle_tab` therefore takes the default branch whatever the balance, once the vault has spent `freeze_grace_seconds` frozen within its current episode. The beta parameter files set 345,600 seconds (96 hours), and the program refuses any configured value below that floor. Before the grace has run, the call fails with `VaultFrozen` and the operator has time to unfreeze and settle normally.

A pending bond withdrawal is cleared when the default opens. Its amount was checked against the balance before the default, and after payout the residual may be smaller than the request, so the request would otherwise block a valid new one. The operator can request again once the event is paid.

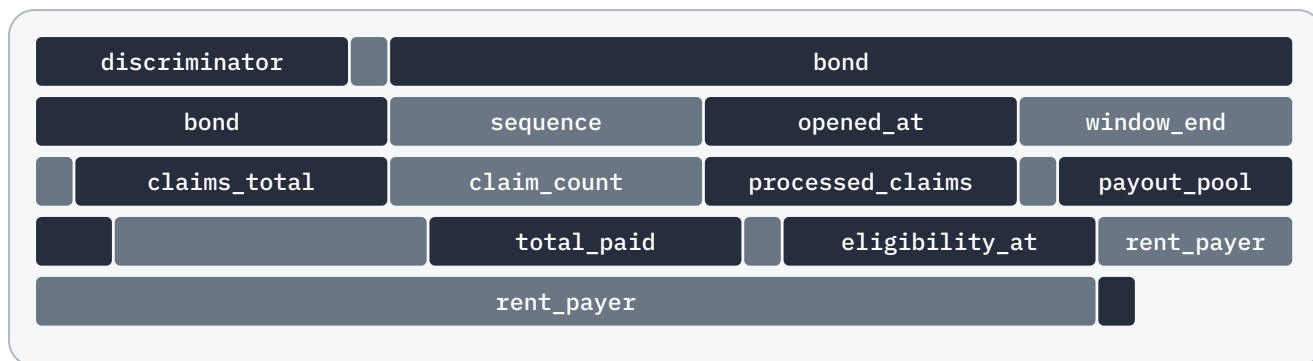
The sequence number

`DefaultEvent` lives at the PDA `[b"default", bond, sequence]`, with the sequence as a little-endian u64. The merchant passes `default_sequence` as an instruction argument because Anchor derives the account address from it, and the program requires it to equal `bond.default_sequence + 1`. The event address is therefore the bond's next unused sequence number, and no caller can open an event at any other address. The builder computes it from the indexed bond.

The account is declared `init_if_needed` with the merchant as payer, so every `settle_tab` transaction carries it. On the healthy path the program closes it again at the end of the same instruction and returns its 1,976,640 lamports of rent to the merchant; the placeholder never outlives the transaction, and its `opened_at` stays zero. On the default path the program fills it and keeps it. The merchant whose settlement opened the default has funded the public record of that default.

`bond.default_sequence` is also the bond's epoch counter. Each `CreditLine` stores `opened_default_sequence`, the epoch it was opened or last renewed in. When a default opens and the counter advances, every line of the bond falls one epoch behind. `settle_tab` rejects such a line with `LineExpired`, `resize_line` refuses to raise it, and only `renew_line` brings it into the new epoch. `file_claim` accepts a line only when `line.opened_default_sequence + 1` equals the event's sequence. The lines a default retires are exactly the lines that may claim against it, and a line renewed afterwards cannot claim against an old event.

While `active_default_sequence` is set, the bond is closed to change: `open_line`, `resize_line`, `renew_line`, `request_withdrawal`, `finalize_withdrawal` and further `settle_tab` calls on that bond all fail with `DefaultActive`. The pool the claims will share cannot be diluted by a new line or reduced by a withdrawal while the window is open.



156 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	first 8 bytes of sha256("account:DefaultEvent")
8	1	bump	PDA bump for [b"default", bond, sequence]
9	32	bond	the bond this default draws on
41	8	sequence	u64 LE, the bond's new default_sequence
49	8	opened_at	i64 Unix seconds; zero marks a placeholder
57	8	window_end	i64, opened_at + 259 200
65	8	claims_total	u64, Σ eligible_amount of filed claims
73	8	claim_count	u64, distinct claims filed
81	8	processed_claims	u64, claims paid so far
89	1	snapshot_taken	bool, set by the first payout
90	8	payout_pool	u64, bond.balance at the first payout
98	8	eligible_total_snapshot	u64, claims_total at the first payout
106	8	total_paid	u64, Σ paid_amount
114	1	paid	bool, true once every claim is paid
115	8	eligibility_at	i64, time at which validity was judged
123	32	rent_payer	the merchant whose settlement opened it
155	1	collateral_locked_at_open	bool, freeze lock active at opening

Figure 26. The 156-byte DefaultEvent account, at the offsets the program writes.

`eligibility_at` records the validation time the opening voucher was judged at: the current time, or the line's freeze reference for a frozen vault. `collateral_locked_at_open` records whether the operator's collateral was locked by a frozen vault or an unfreeze tail when the event opened. The flag decides which limit each claim is measured against (§7.2) and cannot change after opening, whatever happens to the freeze later.

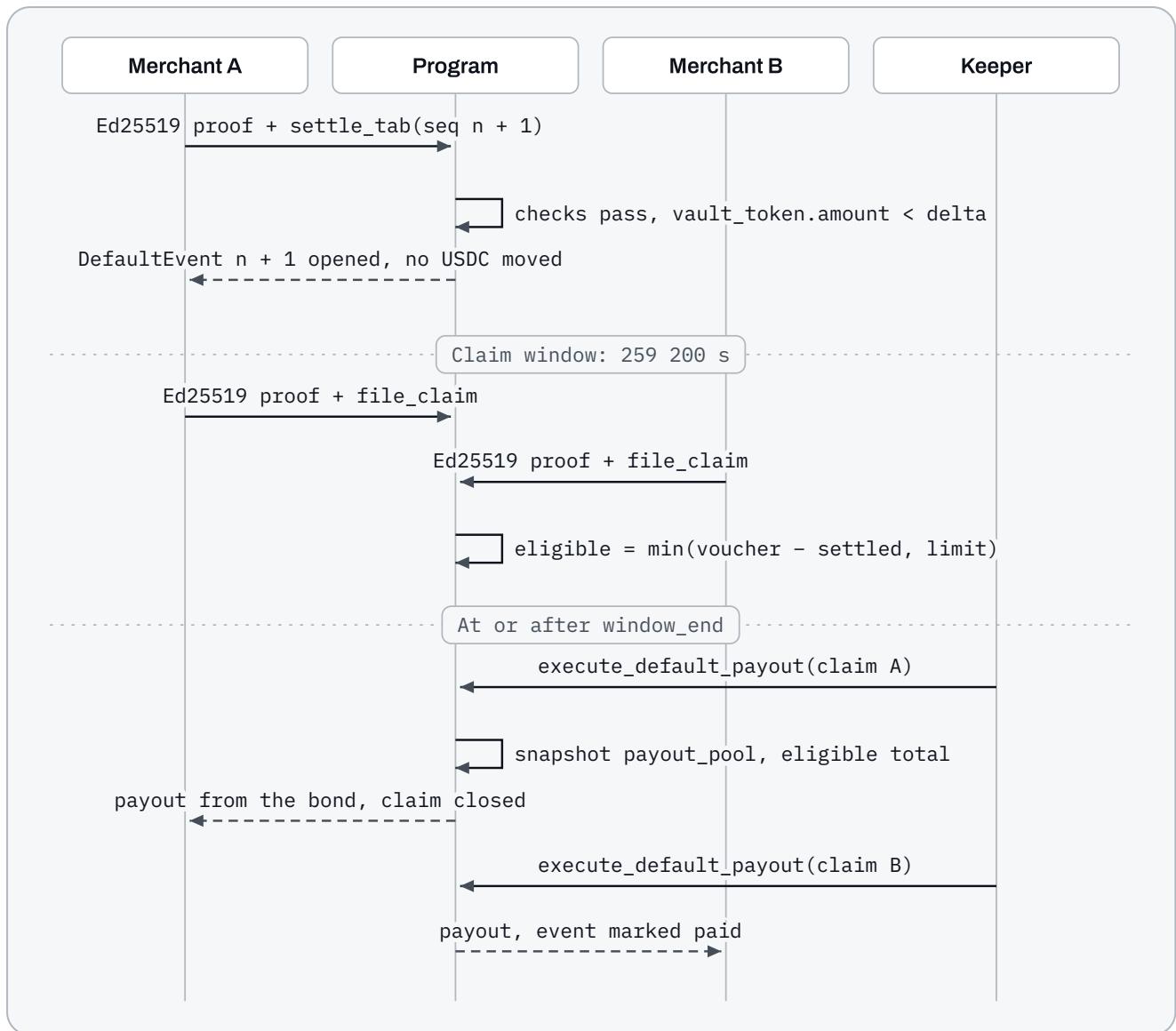


Figure 27. A short vault opens a default event; claims are filed in the window and paid from the bond after it.

7.2 The claim window and `file_claim`

The claim window is a compiled constant, `CLAIM_WINDOW = 72 × 60 × 60` seconds, 259,200 seconds or 72 hours. The deployment parameter files carry the same value and the deploy script refuses a file that differs from the compiled constant. A claim is accepted while `now < window_end`; at `window_end` the window is closed.

The window has two jobs. It gives every line holder time to see the event and file, and it fixes the moment after which the set of claims can no longer change, so the payout can be computed from a closed list. Merchants do not need to watch the chain by hand: the merchant middleware's worker discovers open defaults on its lines and files the newest eligible voucher it holds.

The same 72 hours sits inside the bond's withdrawal unlock, which is what keeps an operator from leaving ahead of a default:

$$\text{unlock_at} = \max(\text{now}, \text{bond.longest_line_expiry}) + 259\,200 + \text{withdrawal_margin_seconds}$$

`bond.longest_line_expiry` high water mark of every line expiry the bond has granted
`withdrawal_margin_seconds` protocol configuration, 86 400 s in the devnet and mainnet-beta files

A withdrawal requested today cannot complete until every voucher on every line of the bond has expired and a full claim window has passed after that. Opening or renewing a line extends a pending request's unlock to match. §6 describes bond withdrawals and their cooldown.

Who files, and with which voucher

`file_claim(ed25519_ix_index: u16)` is signed by the merchant. The line is the PDA `[b"line", bond, merchant]` with `has_one` constraints on bond, merchant and vault, so only the holder of a line on this bond can file, and only for its own line. The merchant whose settlement opened the default files like everyone else: `settle_tab` records the event, not a claim.

The voucher arrives the same way as in `settle_tab`, as the canonical Ed25519 instruction immediately before `file_claim`. The merchant files its latest voucher, the one with the highest cumulative amount. The program checks, in order:

Check	Condition	Error
Window open	<code>opened_at > 0</code> , <code>now < window_end</code> , no payout snapshot taken	<code>ClaimWindowClosed</code>
Event is the bond's active default	<code>bond.active_default_sequence == Some(event.sequence)</code>	<code>InvalidDefault</code>
Line belongs to the retired epoch	<code>line.opened_default_sequence + 1 == event.sequence</code>	<code>InvalidDefault</code>
Canonical proof, right line and domain	208-byte Ed25519 instruction at index <code>current - 1</code> , voucher line equals this line	<code>InvalidPrecompile</code> , <code>InvalidVoucher</code>
Signer authorized	the line's voucher key in force at <code>opened_at</code> , an outgoing key covered by the merchant's <code>VoucherAttestation</code> , or the key at the line's freeze reference	<code>InvalidPrecompile</code>
Valid at the judgment time	line not expired at <code>opened_at</code> (or the freeze reference); voucher expiry after it and no later than the line's expiry	<code>VoucherExpired</code>
Advances the line	<code>nonce > line.maximum_nonce</code> and <code>cumulative > max(settled_cumulative, retired_cumulative)</code>	<code>VoucherReplay</code>
Replaces an earlier filing upward	claim unpaid, higher nonce and higher cumulative than the stored claim	<code>VoucherReplay</code>

Validity is judged at the moment the default opened, not at the moment of filing. A voucher that was valid when the vault came up short stays claimable for the whole window even if its own expiry passes during the window. An outgoing voucher key from a line-key rotation remains usable only up to the nonce and cumulative ceilings

the merchant attested during the rotation notice (§5.9).

Claims against a frozen vault

A freeze, by the guardian or the admin, stops every transfer out of the vault (§3.7). Vouchers keep their expiry while the vault sits frozen, so judging them at the opening time alone would let a long freeze run out the clock on debt the vault was prevented from paying. The program therefore lets a claim be judged at the line's freeze reference instead, under conditions it checks from immutable accounts:

- Every real freeze creates a `FreezeCheckpoint` at `[b"freeze", vault, generation]` recording its freeze time, its episode and the operator's `defaults_count` at that moment. Unfreeze writes the end of the freeze and of its recourse tail once.
- The claim supplies an activity checkpoint proving that the freeze episode was still in force when the default opened: a freeze that began before the opening, whose interval or recourse tail covers the opening, and whose recorded `defaults_count` is below the event's sequence. A freeze that happened after the default, even in the same second, cannot qualify. Passing the program ID instead disables the exception.
- The reference is the first checkpoint the line lived through in that episode, `max(line.freeze_generation_at_open + 1, episode start)`, or for a later voucher key, the earliest checkpoint of the episode at which that key was in force.

A voucher authorized and unexpired at that reference is eligible. The claim's limit snapshot is then the stored `line.limit`, as it is for every claim on an event opened while collateral was locked, because the freeze lock prevented the operator from applying any pending decrease. §12.7 describes the freeze path from the guardian's side.

One claim per line

The claim is the PDA `[b"claim", default_event, merchant]`. A bond has one line per merchant, so an event has at most one claim per line. Filing a second time does not add a claimant; it replaces the filed voucher, and only with one that has both a higher nonce and a higher cumulative amount. The program subtracts the old eligible amount from `claims_total` and adds the new one. `claim_count` increments only on the first filing.

Two snapshots are fixed at the first filing and never move:

- `settled_snapshot = max(line.settled_cumulative, line.retired_cumulative)`, the part of the voucher the vault or an earlier default has already resolved.
- `line_limit_snapshot`, the limit the claim is measured against. In the ordinary case it is the limit in force at `opened_at`, so a pending decrease whose `effective_at` had passed by the opening counts. When the claim supplies a freeze reference, or the event recorded `collateral_locked_at_open`, it is the stored `line.limit`, because a decrease the operator could not apply while collateral was locked cannot reduce what the merchant is owed.

$$\text{eligible_amount} = \min(\max(\text{voucher_amount} - \text{settled_snapshot}, 0), \text{line_limit_snapshot})$$
$$\text{claims_total} = \Sigma \text{eligible_amount over the event's claims}$$

The merchant pays the claim account's rent, 2,074,080 lamports for 170 bytes, and receives it back when the claim is paid. Each filing emits `ClaimFiled` with the full snapshot and `CoreEvent` kind 10, so an indexer that starts after the claim has closed can still reconstruct it.

Claim field	Type	Written	Meaning
<code>default_event</code> , <code>line</code> , <code>merchant</code>	Pubkey × 3	first filing	what the claim is against and who receives the payout
<code>voucher_amount</code> , <code>voucher_nonce</code> , <code>voucher_expiry</code>	u64 × 3	every filing	the filed voucher's scalars
<code>settled_snapshot</code>	u64	first filing	<code>max(settled_cumulative, retired_cumulative)</code> at filing
<code>line_limit_snapshot</code>	u64	first filing	the limit the claim is capped at
<code>eligible_amount</code>	u64	every filing	<code>min(voucher - settled, limit)</code>
<code>paid_amount</code> , <code>paid</code>	u64, bool	payout	what the bond transferred
<code>filed_at</code>	i64	every filing	time of the latest filing

7.3 `execute_default_payout`

`execute_default_payout()` takes no arguments and requires no signer in its account list. Anyone may submit it once `now ≥ window_end` ; before that it fails with `ClaimWindowOpen` . Each call pays exactly one claim, named by its accounts: the claim, its line, its merchant and a token account the merchant owns for the configured mint. Every one of those is bound by a constraint: the claim's seeds and `has_one` links, the line's seeds under the bond, `merchant` equal to `claim.merchant` , and `merchant_token.owner == claim.merchant` . The submitter chooses only when the claim is paid, never where the USDC goes or how much.

The first payout of an event takes the snapshot. It sets `snapshot_taken` , copies the bond's tracked `balance` into `payout_pool` and `claims_total` into `eligible_total_snapshot` . Every payout of the event, including the first, then computes its claim's amount from those two fixed numbers:

$$\begin{aligned} \text{payout} &= \text{eligible_amount} \text{ when } E \leq P \\ \text{payout} &= \lfloor P \times \text{eligible_amount} \div E \rfloor \text{ when } E > P \\ \text{residual} &= P - \sum \text{payout} = P - E \text{ when } E \leq P \\ 0 \leq \text{residual} &\leq \text{claim_count} - 1 \text{ when } E > P \end{aligned}$$

-
- P `payout_pool`, the bond's tracked balance at the first payout
E `eligible_total_snapshot`, $\sum \text{eligible_amount}$ at the first payout

The multiplication runs in u128, so `P × eligible_amount` cannot overflow, and the division floors. Rounding is therefore always downward, per claim, and the base units the floors drop stay in the bond. In the pro-rata case each floor drops less than one base unit and the exact shares sum to `P`, so the residual is a whole number below the number of claims. Because every claim's amount depends only on `P`, `E` and its own `eligible_amount`, the result is the same in whatever order the claims are paid, and one claim per instruction is enough: no instruction needs to see the whole list.

The off-chain contract computes the same numbers. `calculateDefaultPayout` in the shared domain package takes the bond balance and the claims, applies `min(max(voucher - settled, 0), limit)`, sets `proRata` when the eligible total exceeds the balance, floors each share and reports the residual with the rounding rule `floor_each_claim_residual_stays_in_bond`. The program's payout function and the TypeScript function are tested against the same vectors.

The pool is the bond's tracked `balance`, not the raw balance of its token account. Only `post_bond` adds to the tracked balance; USDC sent to the bond's token account by a plain transfer is not booked collateral and does not join the pool. The vault does not join it either.

What one payout writes

In one instruction the program:

1. transfers `payout` from the bond's token account to the merchant's token account, signed by the bond PDA;
2. subtracts `payout` from `bond.balance` and from the protocol-wide `total_bond_balance` that enforces the beta cap;
3. marks the claim paid with `paid_amount = payout`;
4. advances the line's `retired_cumulative` to the claim's full `voucher_amount` and its `maximum_nonce` to the claim's nonce;
5. adds `payout` to `total_paid`, increments `processed_claims`, and emits `CoreEvent` kind 11;
6. closes the claim account and returns its rent to the merchant.

When `processed_claims` reaches `claim_count`, the same instruction sets `paid`, clears `bond.active_default_sequence` and emits `CoreEvent` kind 12 with the event's total. The bond is open to change again.

Step 4 retires the whole voucher, including any part a pro-rata payout did not cover. The written-off fraction becomes part of the line's replay floor, so no later settlement or default on that line can collect it again, even after `renew_line` brings the line back into service. A second submission for a claim already paid transfers nothing: the claim account no longer exists, and if a retained one is passed, the handler returns early on `claim.paid`.

When the pool covers everything

Whether a default pays in full is decided before it happens, by the coverage the operator chose. `open_line`, `limit` increases, `renew_line` and both withdrawal steps all require `collateral × 10 000 ≥ reserved_lines × minimum_coverage_bps`, where `collateral` is the tracked balance less the pending (or requested) withdrawal amount and `reserved_lines` includes the limit of every line that can claim. Every eligible amount is capped at its line's limit. At the v0 default minimum of 10,000 bps the lines that can claim therefore never sum to more

than the bond, and every claim is paid in full. Pro-rata payment occurs only on a bond whose operator set its minimum below 10,000 bps, down to the floor of 3,300, and that minimum is public before any merchant extends credit.

An event no one claimed

If the window closes with `claim_count == 0`, there is nothing for `execute_default_payout` to pay. `finalize_empty_default()` is permissionless as well: after `window_end` it sets the snapshot fields, marks the event paid, clears the bond's active default and emits `CoreEvent` kind 12 with a zero total. The bond keeps its whole balance.

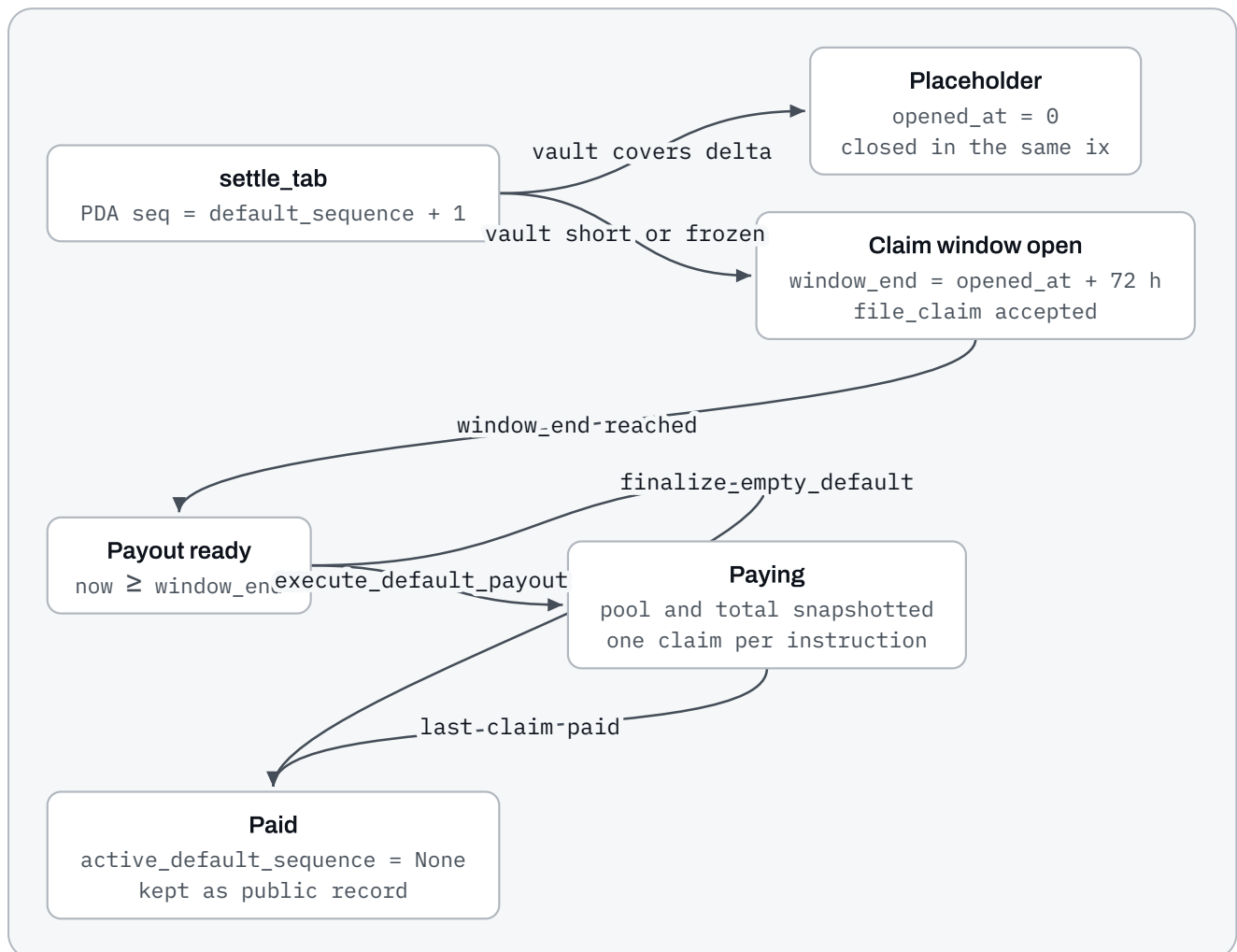


Figure 28. The lifecycle of a DefaultEvent PDA and the instruction that moves it.

7.4 Worked examples

Both examples use dollar figures written in USDC and are carried out in base units, as the program computes them. They are examples, not measurements.

One claimant, paid in full

An operator posts a 10 USDC bond and opens one line of 10 USDC for a merchant that charges 0.25 USDC per request, so coverage is 10,000 bps. The agent makes 40 requests and signs a voucher with each; the last one states a cumulative 10,000,000 base units. The merchant settled once after 16 requests, so `settled_cumulative` is 4,000,000. The operator then draws the vault down to 2,500,000 base units.

Step	Computation	Result
<code>settle_tab</code> with the 40th voucher	$\text{delta} = 10,000,000 - 4,000,000$	$6,000,000 \leq \text{limit } 10,000,000$
Balance check	$2,500,000 < 6,000,000$	default opens at T, $\text{window_end} = T + 259,200$
<code>file_claim</code>	$\min(10,000,000 - 4,000,000, 10,000,000)$	<code>eligible_amount</code> 6,000,000
First payout at or after <code>window_end</code>	$P = 10,000,000, E = 6,000,000, E \leq P$	payout 6,000,000
Afterwards	$\text{bond.balance} = 10,000,000 - 6,000,000$	residual 4,000,000; vault still holds 2,500,000

The merchant receives every base unit the agent vouched for and the vault had not paid. The line's `retired_cumulative` becomes 10,000,000, and the operator's record shows one default.

Five claimants, paid pro-rata

The second example is the five-line bond the site's default passage shows. One bond of 900 USDC backs five lines whose limits sum to 1,500 USDC, a coverage of 6,000 bps, which the operator's minimum allows. At time T the inference API settles its newest voucher: cumulative 1,880 USDC against 1,500 settled gives a delta of 380 USDC, within its 400 USDC limit. The vault holds less than 380 USDC, so the default opens and nothing moves. All five merchants file within the window.

Line	Limit	Voucher	Settled	Unsettled	Eligible	Payout (base units)	Retired unpaid
inference API	400	1,880	1,500	380	380	300,000,000	80.000000
data API	300	950	600	350	300	236,842,105	63.157895
RPC provider	300	420	300	120	120	94,736,842	25.263158
GPU rental	250	610	360	250	250	197,368,421	52.631579
search API	250	290	200	90	90	71,052,631	18.947369
Total	1,500			1,190	1,140	899,999,999	240.000001

Amounts other than the payout column are in USDC. The first payout snapshots $P = 900,000,000$ and $E = 1,140,000,000$. Since $E > P$, each claim receives $\lfloor 900,000,000 \times \text{eligible} \div 1,140,000,000 \rfloor$, which here equals $\lfloor \text{eligible} \times 15 \div 19 \rfloor$:

- inference API: $380,000,000 \times 15 \div 19 = 300,000,000$ exactly.
- data API: $4,500,000,000 \div 19 = 236,842,105$ remainder 5.
- RPC provider: $1,800,000,000 \div 19 = 94,736,842$ remainder 2.
- GPU rental: $3,750,000,000 \div 19 = 197,368,421$ remainder 1.
- search API: $1,350,000,000 \div 19 = 71,052,631$ remainder 11.

The remainders sum to 19 nineteenths, which is the single base unit left in the bond: `residual = 900,000,000 - 899,999,999 = 1`. Every merchant receives 15/19 of its eligible amount, rounded down to the base unit, whichever order the keeper processes them in.

The data API's unsettled 350 USDC exceeds its 300 USDC limit. The bond's liability on a line ends at the line's limit, so the 50 USDC above it is outside any claim. The column on the right is the part of each eligible amount the pool could not cover; each line's `retired_cumulative` moves to the full voucher amount (1,880, 950, 420, 610 and 290 USDC), so none of it can be collected again on the same line.

7.5 The keeper's part

The payout needs a transaction after `window_end`, and someone has to send it. That role is the keeper's, and the program grants it no authority: `execute_default_payout` and `finalize_empty_default` have no signer constraint, every account they touch is fixed by the event and its claim, and the only choice left to the submitter is timing. Any key can run the keeper, any merchant can pay its own claim with the same instruction, and replacing the keeper changes nothing about who is paid or how much. The deployment requires no keeper registration.

Bonded's keeper worker runs in passes. In each pass it:

1. reconciles the chain projection and reads the indexer cursor's last block time; the keeper decides what is due by chain time, not by its own clock;
2. queues every default event that is not paid and whose `window_end` has passed, and every escrow whose deadline has passed (§8.4);
3. for each due event, takes a 60-second lease on the event's job, lists its unpaid claims in address order, and plans one claim per legacy transaction, or one `finalize_empty_default` for an event with no claims;
4. for each claim, keeps a persisted attempt record. If an earlier attempt was submitted, it checks its status and rebroadcasts the identical signed bytes instead of signing anew; if it confirmed, it marks the attempt done;
5. otherwise prepares `execute_default_payout`, refuses to sign unless the live simulation succeeds, signs, submits, and records the signature;
6. after the claims, marks the event's job done if the indexed event is paid, or requeues it.

Failures are local to one claim. A failed attempt goes to retry with a backoff of `min(60 s, 1 s × 2min(attempts, 6))`, and the other claims of the event and other events proceed. A merchant whose token account cannot receive does not delay anyone else's payout. The keeper can also be given a merchant-chosen replacement destination; the program still checks that its owner is the claim's merchant and its mint is the configured USDC mint.

§11.3 describes how the worker is deployed. The builder places an idempotent associated-token-account creation for the merchant before the payout when no replacement destination is given, paid by the transaction's signer. A keeper therefore spends 5,000 lamports in signature fees per payout transaction and the priority fee of its API-prepared compute budget (1,000 micro-lamports per unit by default), plus the rent for the merchant's associated token account if the merchant does not yet have one. In the local development stack the keeper runs every 500 milliseconds against the local bank with a local key.

7.6 What the operator's record shows afterwards

A default leaves a permanent, public trace in the operator's accounts, and the trace is written by the program, not reported by the operator.

`Operator.defaults_count` increments in the instruction that opens the default, in the same step as `bond.default_sequence`. The counter records that the vault came up short, whether the bond then covered every claim or not. `tabs_settled` counts healthy settlements and does not change. The API serves these as the operator's reputation, `tabsSettled` and `defaults`, together with `daysActive`, which the indexer derives from the operator's `created_at` and the last block time. A merchant can read the record before it extends credit on a line.

After the last payout of the five-claimant example, the accounts read:

Account	Field	After the last payout
Operator	<code>defaults_count</code>	one higher than before the default
Bond	<code>balance</code>	1 base unit (the residual)
Bond	<code>default_sequence</code>	the event's sequence
Bond	<code>active_default_sequence</code>	None
Bond	<code>withdrawal</code>	None, cleared at opening
Bond	<code>reserved_lines</code>	1,500,000,000, unchanged
CreditLine × 5	<code>opened_default_sequence</code>	one behind the bond: retired
CreditLine × 5	<code>retired_cumulative</code> , <code>maximum_nonce</code>	the claimed voucher's amount and nonce
DefaultEvent	<code>paid</code> , <code>payout_pool</code> , <code>eligible_total_snapshot</code> , <code>total_paid</code>	true, 900,000,000, 1,140,000,000, 899,999,999
Claim × 5	—	closed; history in <code>ClaimFiled</code> and <code>CoreEvent</code> kind 11

The paid `DefaultEvent` stays on chain as the public record of the default: its pool, its denominator and the total paid. The API reports it as `payout_executed` with a `roundingRemainder` equal to `payout_pool - total_paid`, here 1 base unit.

The lines are retired, not closed. A `CreditLine` PDA persists for the life of the bond, one per merchant, and keeps its watermarks; the API reports each as `defaulted`. Their limits stay in `reserved_lines`, so the bond's public coverage ratio, computed against the limits of its lines, reads 0 bps until the operator acts. Two paths lead out:

- To leave, the operator schedules each line's limit down to zero with `resize_line` and applies the decreases after the 72-hour delay, which frees `reserved_lines`, then withdraws the residual through `request_withdrawal` and `finalize_withdrawal` on the usual unlock.
- To continue, the operator posts new collateral with `post_bond` and calls `renew_line`, which must pass the coverage check against the new balance and moves the line into the current epoch while keeping its replay floors, so nothing already resolved can be claimed again.

`request_withdrawal` checks coverage against every reserved limit, retired lines included, so the residual leaves the bond only after the reservations it no longer backs have been decreased.

8. Escrow

Escrow is the lane for jobs that are paid before they are delivered: a compute run, a dataset, a service with a turnaround. Here the exposure runs the other way from a tab. The agent pays first, and the merchant may never deliver. On a stablecoin rail a payment is final, so the payer has no chargeback to fall back on.

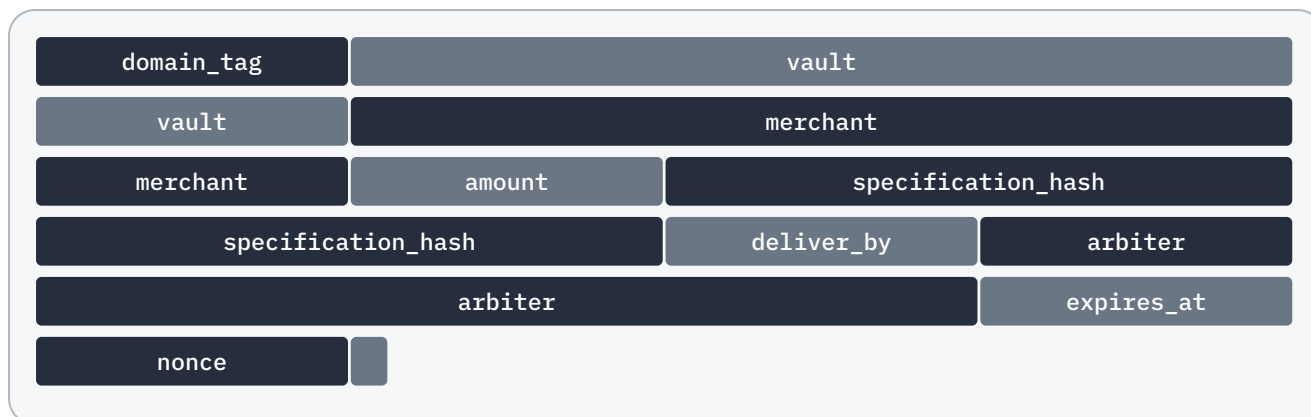
The escrow lane closes that gap without asking anyone to judge whether a job was done well. The agent's payment goes into a token account owned by the program, bound to a quote the merchant signed, and it leaves only on objective events: a delivery mark before the deadline, a dispute window that lapses, a decision by the arbiter the quote named, or a deadline that passes with nothing. If no delivery mark arrives by the deadline, anyone can return the whole payment to the vault.

Eight instructions make up the lane: `pay_escrow` funds a job; `mark_delivered` records delivery; `accept` and `release_after_window` pay the merchant; `dispute` and `resolve` route a contested job to its arbiter; `refund_on_timeout` and `refund_after_resolution_deadline` return the payment. Every one of them except `accept` checks the clock against a deadline determined by the quote, a program constant or the configuration snapshot at funding.

8.1 The merchant-signed quote

A quote is the merchant's offer for one job: the price, a hash of what will be delivered, the delivery deadline, the arbiter if there is one, and how long the offer stands. The merchant signs it, and the program will fund an escrow only against a quote whose signature it has verified. The price, the deadline and the arbiter therefore come from the merchant's signature, never from the agent's arguments. A decimal slip in the agent or an injected instruction cannot change the amount, and the merchant cannot later claim it offered different terms.

The quote is a fixed 169-byte message, little-endian, with a context tag that binds it to one program on one cluster.



169 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	domain_tag	first 8 bytes of sha256(prefix program genesis)
8	32	vault	the paying vault's address
40	32	merchant	the merchant's key, which must sign
72	8	amount	u64 LE principal, USDC base units
80	32	specification_hash	hash of the job specification
112	8	deliver_by	u64 LE Unix seconds, delivery deadline
120	32	arbiter	the arbiter's key, all zero for none
152	8	expires_at	u64 LE Unix seconds, offer expiry
160	8	nonce	u64 LE quote nonce
168	1	lane	0 = pay, 1 = escrow

Figure 29. The 169-byte quote the merchant signs for an escrow job.

domain_tag = sha256("Bonded quote v1\0" || program_id ||
cluster_genesis_hash)[0..8]

quote_hash = sha256(quote[0..169])

program_id, cluster_genesis_hash	raw 32-byte values; the genesis hash is stored in ProtocolConfig at initialization
quote[0..169]	the complete signed message, tag and lane byte included

The tag uses the same derivation as the voucher domain with its own prefix, so a quote signed for a devnet deployment, or for another program, is refused on mainnet-beta, and a quote cannot be replayed as a voucher or an approval. The lane byte separates quotes for an immediate program `pay` (lane 0) from quotes for escrow

(lane 1); each instruction accepts only its own lane. The off-chain encoder writes an absent arbiter as 32 zero bytes and rejects an explicit all-zero arbiter key, so "no arbiter" has exactly one encoding.

`quote_hash`, the SHA-256 of the full message, is the quote's identity. It seeds the `Escrow` PDA, it seeds the persistent `QuoteReceipt` that makes the quote single-use, and the payment intent commits to it.

The signature check

The merchant's Ed25519 signature is verified by the native Ed25519 precompile in the instruction immediately before `pay_escrow`. The program does not trust the precompile's presence alone; it loads that instruction through the instructions `sysvar` and requires its exact canonical form.



281 bytes · one cell per 4 bytes

Offset	Bytes	Field	Meaning
0	16	<code>header</code>	[1,0,48,0,255,255,16,0,255,255,112,0,169,0,255,255]
16	32	<code>public_key</code>	the key that signed, must equal the merchant
48	64	<code>signature</code>	Ed25519 signature over the 169 bytes
112	169	<code>message</code>	the quote

Figure 30. The 281-byte Ed25519 precompile instruction that carries the merchant's quote proof.

The header encodes one signature whose key, signature and message all lie inside this same instruction (index 0xFFFF) at offsets 16, 48 and 112, with a 169-byte message. The instruction must belong to the Ed25519 program, reference no accounts, and sit at index `current - 1`, where `current` is `pay_escrow` itself. Foreign offsets, a second signature, or a proof elsewhere in the transaction are refused with `InvalidPrecompile`. The measured proof instruction is 281 bytes, and the `pay_escrow` instruction data is 82 bytes.

After decoding, `verify_quote` requires every field to agree with the transaction:

Check	Condition	Error
Length and context	169 bytes, tag matches this program and cluster, lane byte ≤ 1	<code>InvalidQuote</code>
Vault	<code>quote.vault</code> equals the vault being debited	<code>InvalidQuote</code>
Merchant and signer	<code>quote.merchant</code> equals the merchant account and the precompile's signer	<code>InvalidQuote</code>
Amount	<code>quote.amount</code> equals the <code>amount</code> argument	<code>InvalidQuote</code>
Lane	<code>quote.lane == 1</code>	<code>InvalidQuote</code>
Offer still open	<code>quote.expires_at > now</code>	<code>InvalidQuote</code>

Check	Condition	Error
Hash	<code>sha256(quote) == quote_hash argument</code>	<code>QuoteMismatch</code>
Arbiter	<code>quote.arbiter $\neq$ quote.merchant</code>	<code>MerchantCannotArbitrate</code>
Deadline	<code>now $\leq$ quote.deliver_by</code>	<code>DeadlinePassed</code>

The API accepts the quote as a `QuoteInput` : `quoteHash` , `amount` , `specificationHash` , `deliverBy` , `arbiter` , `expiresAt` , and the merchant's signature and message in base64. Before it builds a transaction, the API decodes the message, recomputes the hash, compares every field with the DTO, checks vault, merchant and lane, and verifies the signature. The on-chain checks above then repeat the decisive ones against the bytes the runtime verified.

8.2 `pay_escrow` and the `Escrow` account

`pay_escrow(amount: u64, quote_hash: [u8; 32], intent_hash: [u8; 32], ed25519_ix_index: u16)` is signed by the vault's session key, which also pays for the accounts the instruction creates. The merchant must already be allowlisted for the vault: the instruction requires the `MerchantRule` at `[b"rule", vault, merchant]` , and every limit in that rule and in the vault policy applies to the escrow exactly as to a direct payment.

The handler runs in this order:

1. Reads the protocol's escrow configuration. `escrow_fee_bps` , `maximum_escrow_fee_bps` and a positive `escrow_resolve_seconds` must all be set, and the fee must not exceed its maximum; otherwise `InvalidConfiguration` . Escrow funding is refused until governance has configured both the fee and the resolution duration.
2. Computes `fee_amount = [ amount  $\times$  fee_bps  $\div$  10 000 ]` and `total_debit = amount + fee_amount` (§8.7).
3. Verifies the quote and its deadline (§8.1).
4. Recomputes the payment intent and requires it to equal `intent_hash` (`ApprovalMismatch`).
5. Books the spend against the vault's policy with `total_debit` as the amount, through the same routine as `pay` (§3.4): vault not frozen, window reconciled, merchant allowlisted, `total_debit  $\leq$  per_transaction_cap` , merchant window budget, vault window budget including outstanding float allowance (§4), and a co-signer approval when `total_debit` exceeds the threshold (§8.6).
6. Refuses a quote whose `QuoteReceipt` is already used (`PaymentReplay`), then writes the `QuoteReceipt` , the `PaymentReceipt` and the `Escrow` .
7. Transfers `total_debit` from the vault's token account to the escrow's token account, signed by the vault PDA.
8. Emits `PaymentCommitted` and an `EscrowEvent` snapshot that carries the 169-byte quote and the merchant's 64-byte signature.
9. Closes a consumed approval to its original rent payer.

```
intent_hash = sha256(tag || vault || merchant || total_debit || quote.nonce ||
                    quote_hash || 1)
```

```
tag = sha256("Bonded payment intent v1\0" || program_id ||
            cluster_genesis_hash)[0..8]
```

```
total_debit    u64 LE, principal plus the configured fee
quote.nonce    u64 LE, the nonce from the signed quote
1              the lane byte for escrow; the encoded intent is 121 bytes
```

The intent is the single hash that every authorization of this payment refers to. It commits to the vault, the merchant, the full debit including the fee, the quote nonce, the quote itself and the lane. It seeds the `PaymentReceipt` at `[b"payment", vault, intent_hash]`, which is created with `init` and never closed, so the same intent can be paid at most once. The quote has its own persistent receipt at `[b"quote", vault, quote_hash]`, so the same quote cannot fund a second escrow under a different fee or intent. The `Escrow` itself is the PDA `[b"escrow", vault, quote_hash]`.

`pay_escrow` creates up to four accounts, all paid by the session key:

Account	Seeds	Size	Rent (lamports)	Lifetime
<code>Escrow</code>	<code>[b"escrow", vault, quote_hash]</code>	400	3,674,880	closed on the terminal outcome, rent to <code>rent_payer</code>
Escrow token account	associated token account of the <code>Escrow</code> PDA	165	2,039,280	closed on the terminal outcome, rent to <code>rent_payer</code>
<code>PaymentReceipt</code>	<code>[b"payment", vault, intent_hash]</code>	211	2,359,440	persistent replay guard and result
<code>QuoteReceipt</code>	<code>[b"quote", vault, quote_hash]</code>	138	1,851,360	persistent single-use guard

Rent is $(\text{size} + 128) \times 3,480 \times 2$ lamports; §10.5 shows the computation. The escrow token account is created with `init_if_needed` under the associated-token constraints, so a correctly precreated account is accepted.

The escrow token account's authority is the `Escrow` PDA, and the program signs for it with the seeds `[b"escrow", vault, quote_hash, bump]` only inside the handlers of §8.3. No private key controls it: not the session key that funded it, not the operator's admin, not the merchant, and not the protocol's upgrade authority, which can't move even the fee treasury. The escrowed USDC can leave by exactly two routes, release to the quote's merchant or refund to the paying vault, and the state machine decides which.



400 bytes · one cell per 4 bytes

Offset	Bytes	Field	Meaning
0	8	discriminator	first 8 bytes of sha256("account:Escrow")
8	1	bump	PDA bump for [b"escrow", vault, quote_hash]
9	32	vault	the paying vault
41	32	quote_hash	sha256 of the signed quote
73	32	token_account	the escrow's USDC token account
105	32	merchant	who is paid on release
137	8	amount	u64 principal from the quote
145	8	fee_amount	u64 fee reserved at funding
153	2	fee_bps	u16 fee rate snapshot
155	32	specification_hash	from the quote
187	8	deliver_by	i64 delivery deadline
195	8	delivered_at	i64, zero until marked
203	8	dispute_until	i64, delivered_at + 172 800
211	32	deliverable_hash	the merchant's output commitment
243	32	arbiter	from the quote, zero for none
275	1	state	0 funded, 1 delivered, 2 disputed, 3 released, 4 refunded
276	32	payment_receipt	the intent's PaymentReceipt
308	32	quote_receipt	the quote's QuoteReceipt
340	32	rent_payer	the session key that funded
372	8	created_at	i64 funding time
380	4	resolve_seconds	u32 snapshot of the resolution duration
384	8	disputed_at	i64, zero until disputed
392	8	resolve_until	i64, disputed_at + resolve_seconds

Figure 31. The 400-byte Escrow account.

Three values are snapshots taken at funding: `fee_bps` with its `fee_amount`, and `resolve_seconds`. A later governance change to the fee or the resolution duration applies only to escrows funded afterwards. A job's cost and its recourse are fixed at the moment the payer commits.

`specification_hash` comes from the quote and `deliverable_hash` from the merchant's delivery mark. The program stores both and does not compare them: the hashes are commitments the parties and an arbiter can check against the actual specification and output off chain, and each transition's `EscrowEvent` preserves them after the account closes.

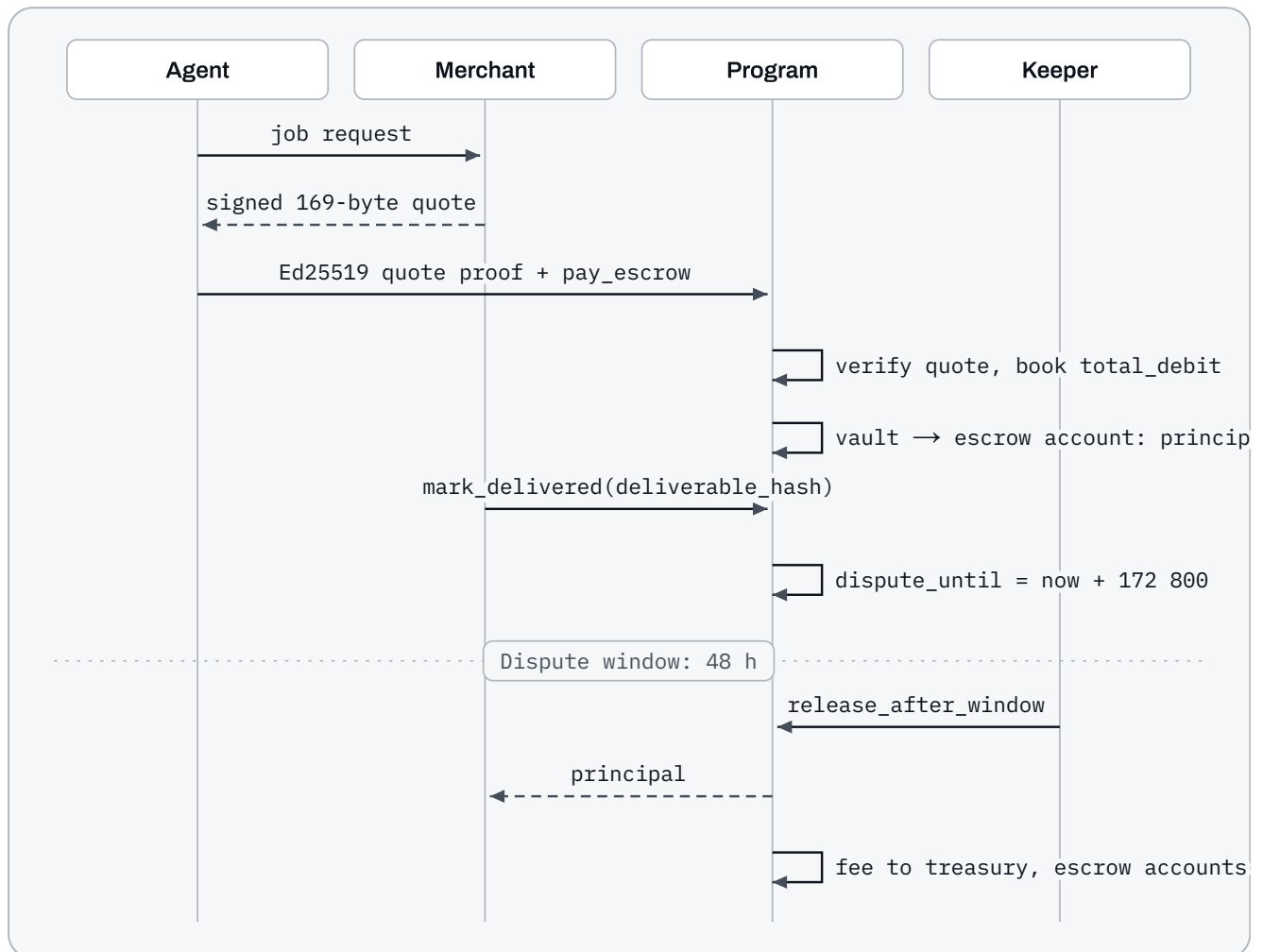


Figure 32. An escrow job funded from a signed quote, delivered, and released when the dispute window lapses.

8.3 The state machine

The program keeps five states in `Escrow.state`: funded (0), delivered (1), disputed (2), released (3) and refunded (4). Acceptance and resolution are instructions, not states: `accept` moves a delivered job to released, and `resolve` moves a disputed job to released or refunded. Released and refunded are terminal. The handler that reaches either one closes the escrow token account and the `Escrow` account in the same instruction, so a terminal state is never stored in a live account; it survives in the final `EscrowEvent` snapshot and in the `PaymentReceipt`, whose `outcome` becomes 1 (merchant paid) or 2 (vault refunded) with `completed_at` set.

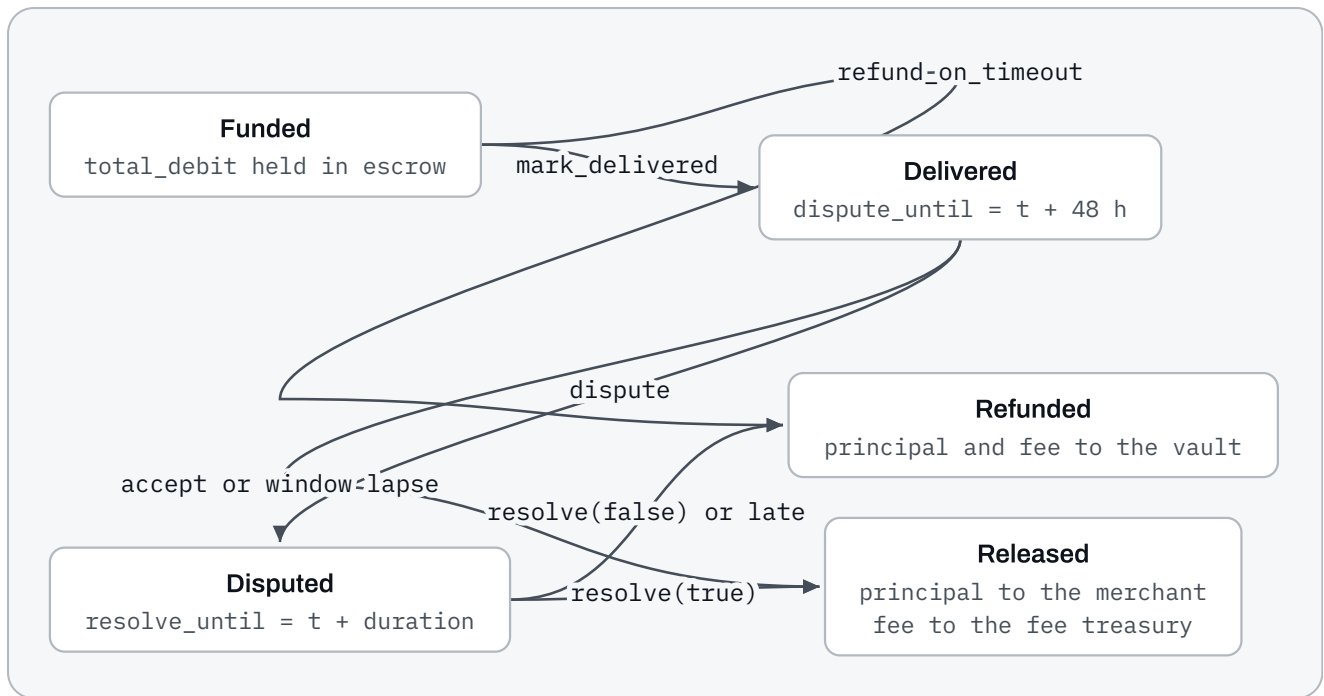


Figure 33. An escrow job's states and the instructions that move it.

Every transition, with who may sign it and when:

From	Instruction	To	Who signs	When	Further condition
funded	<code>mark_delivered(deliverable_hash)</code>	delivered	the escrow's merchant	<code>now ≤ deliver_by</code>	sets <code>delivered_at</code> , <code>dispute_until = now + 172,800</code> , <code>deliverable_hash</code>
funded	<code>refund_on_timeout</code>	refunded	any signer	<code>now > deliver_by</code>	none
delivered	<code>accept</code>	released	the vault's session key or the operator's admin	any time while delivered	none
delivered	<code>dispute</code>	disputed	the vault's session key or the operator's admin	<code>now < dispute_until</code>	an arbiter is named; sets <code>disputed_at</code> , <code>resolve_until = now + resolve_seconds</code>
delivered	<code>release_after_window</code>	released	any signer	<code>now ≥ dispute_until</code>	none
disputed	<code>resolve(true)</code>	released	the named arbiter	<code>now < resolve_until</code>	none
disputed	<code>resolve(false)</code>	refunded	the named arbiter	<code>now < resolve_until</code>	none
disputed	<code>refund_after_resolution_deadline</code>	refunded	any signer	<code>now ≥ resolve_until</code>	<code>resolve_until > 0</code>

Any other instruction in any state fails with `StateTransitionNotAllowed`, a wrong signer with `ActorNotAllowed`, an early call with `DeadlineNotReached`, a late one with `DeadlinePassed`, and a dispute or resolution without an arbiter with `ArbiterRequired`. The off-chain contract carries the same table as `ESCROW_TRANSITIONS`, and `checkEscrowTransition` evaluates state, then actor, then arbiter, then deadline. The program's own order differs in places (`dispute` and `accept` check the actor before the state, `resolve` the arbiter before the actor), so a call that breaks two rules may name a different one on chain. The API uses it to report, for each escrow, which actions are open to which role at the indexed chain time.

The principal on the paying side is either the vault's session key or the operator's admin key; the program accepts both for `accept` and `dispute`. The agent can accept a delivery as a step of its own workflow, and the operator can step in for a job the agent is no longer handling. The merchant can never accept or dispute its own job, and the permissionless instructions exist only where the outcome is already decided by a deadline: once a deadline has passed, the result does not depend on who sends the transaction.

How a job finishes

Every path to released or refunded ends in one routine. It first requires the escrow token account to hold at least `amount + fee_amount`, then:

- **Release** (`accept`, `release_after_window`, `resolve(true)`). The destination must be a classic SPL token account whose owner is `escrow.merchant` and whose mint is the configured USDC mint. The routine transfers `amount` to it, then `fee_amount` to the protocol's fee treasury when the fee is positive, then any surplus above `amount + fee_amount` back to the vault.
- **Refund** (`refund_on_timeout`, `resolve(false)`, `refund_after_resolution_deadline`). The routine transfers the escrow token account's entire balance to the vault: principal, the reserved fee, and any surplus.

It then closes the escrow token account and the `Escrow` account, sending both rents to `rent_payer`, the session key that funded the job, and never to the caller. It writes the outcome into the `PaymentReceipt` and emits the final snapshot.

Two details keep the refund path unconditional. The merchant's destination account is passed unchecked and deserialized only on release, so a merchant that closes or corrupts its token account can block its own payment but never the vault's refund. And USDC sent to the escrow token account by a plain transfer is returned to the vault rather than left behind, so a donation cannot keep the account from closing.

The record after closure

Closing the accounts on every terminal outcome keeps finished jobs from holding rent, and it means the chain's account state shows only open jobs. The history lives in events. Every transition emits an `EscrowEvent` with the complete account snapshot, the actor and the time; the funding event alone also carries the 169-byte quote and the merchant's 64-byte signature, and later events carry zeroes in those fields. The indexer rebuilds each job from this evidence: before it records a funded escrow it decodes the quote from the funding event, checks that vault, merchant, amount, specification hash, deadline, arbiter and hash match the snapshot, and verifies the merchant's signature. A job that closed before the indexer first saw it is reconstructed from the same committed events. The persistent `PaymentReceipt` and `QuoteReceipt` remain on chain as the payment's single-use guards and its result.

A refund returns the USDC but does not rewind the vault's window counters or the merchant rule's window spend. The window budget limits what the session key can commit within a window, whatever the outcome, so a job that times out does not grant the agent fresh spending authority inside the same window.

8.4 Deadlines and windows

Four clocks govern a job. Two come from the quote, one is a program constant and one is protocol configuration.

Deadline	Source	Set at	Boundary
<code>expires_at</code>	quote	merchant's signature	funding requires <code>now < expires_at</code>
<code>deliver_by</code>	quote	merchant's signature	funding and <code>mark_delivered</code> require <code>now ≤ deliver_by</code> ; <code>refund_on_timeout</code> requires <code>now > deliver_by</code>
<code>dispute_until</code>	<code>delivered_at + DISPUTE_WINDOW_SECONDS</code>	<code>mark_delivered</code>	<code>dispute</code> requires <code>now < dispute_until</code> ; <code>release_after_window</code> requires <code>now ≥ dispute_until</code>
<code>resolve_until</code>	<code>disputed_at + resolve_seconds</code>	<code>dispute</code>	<code>resolve</code> requires <code>now < resolve_until</code> ; <code>refund_after_resolution_deadline</code> requires <code>now ≥ resolve_until</code>

`DISPUTE_WINDOW_SECONDS` is a compiled constant, $48 \times 60 \times 60$, 172,800 seconds. The resolution duration is protocol configuration: governance sets it with `configure_escrow_resolution(resolve_seconds)`, which requires a positive value, and every escrow snapshots it at funding. The devnet and mainnet-beta parameter files set 259,200 seconds, 72 hours, as does the local development stack.

Each pair of boundaries meets without overlap or gap. At exactly `deliver_by` the merchant may still mark delivery and the refund is refused; one second later only the refund is possible. At exactly `dispute_until` the dispute is closed and the release is open. At exactly `resolve_until` the arbiter's authority has ended and the refund is open. There is no second at which both sides of a boundary are available, and none at which neither is.

`expires_at` and `deliver_by` measure different things. `expires_at` bounds how long the merchant's offer can be taken up; `deliver_by` bounds the job itself. A merchant can offer a quote valid for an hour on a job due in sixty.

`refund_on_timeout` stays available after `deliver_by` for as long as the job is funded, and it needs no signature from the agent. The keeper runs it: each pass, it selects funded escrows with `deliver_by` before the indexed chain time, delivered escrows at or past `dispute_until`, and disputed escrows at or past `resolve_until`, and sends `refund_on_timeout`, `release_after_window` or `refund_after_resolution_deadline` respectively, with the same persisted attempts, leases and retries it uses for default payouts (§7.5, §11.3).

A timeline, as an example

A merchant quotes a job at 340 USDC with `deliver_by` 60 hours after funding and names an arbiter. The fee at 25 bps is 850,000 base units, so the vault's debit is 340,850,000. Times are hours after funding at `t0`.

Path	Events	Outcome
Delivered, undisputed	<code>mark_delivered</code> at 41 h sets <code>dispute_until</code> to 89 h; at 89 h the keeper calls <code>release_after_window</code>	merchant 340,000,000; fee treasury 850,000
Delivered, accepted early	<code>mark_delivered</code> at 41 h; the agent calls <code>accept</code> at 43 h	merchant 340,000,000; fee treasury 850,000

Path	Events	Outcome
Never delivered	nothing by 60 h; from 60 h plus one second, anyone calls <code>refund_on_timeout</code>	vault 340,850,000
Disputed, arbiter decides	delivered at 41 h; disputed at 70 h, <code>resolve_until</code> = 70 + 72 = 142 h; arbiter calls <code>resolve(false)</code> at 100 h	vault 340,850,000
Disputed, arbiter silent	delivered at 41 h; disputed at 70 h; at 142 h anyone calls <code>refund_after_resolution_deadline</code>	vault 340,850,000

The longest any of these jobs can hold the vault's USDC is fixed at funding. With delivery on the last permitted second and a dispute on the last second of its window, every path is open to some signer by `deliver_by` + 172,800 + `resolve_seconds`, here 60 + 48 + 72 = 180 hours after funding.

8.5 The arbiter and `resolve`

The arbiter is a key the merchant names in the quote, at bytes 120 to 151. The payer agrees to it by funding: the session key signs `pay_escrow` for that exact quote hash, and when a co-signer is required, the approval commits to an intent that contains the quote hash. The program places one rule on the choice: the arbiter cannot be the merchant (`MerchantCannotArbitrate`), since a merchant that could decide its own dispute would make the dispute meaningless. Beyond that the program keeps no allowlist of arbiters, and any key qualifies, a multisig included. Which arbiters an agent accepts is decided by which quotes it funds.

A dispute exists only if an arbiter was named. `dispute` requires a nonzero arbiter (`ArbiterRequired`) and can be raised only by the paying side, only on a delivered job, and only before `dispute_until`. A quote without an arbiter has no party to decide a dispute, so the program does not open one: the principal can accept early, and otherwise the job releases to the merchant when the window lapses. The payer chose a quote that named no arbiter, and the quote's terms decide.

`resolve(release: bool)` is signed by the arbiter, `actor == escrow.arbiter`, on a disputed job before `resolve_until`. Its outcome is binary. `true` releases the principal to the merchant and earns the fee; `false` refunds principal and fee to the vault. There is no split: the instruction carries no amount, and every destination it can reach, the merchant's own token account, the vault's token account and the fee treasury, is fixed by account constraints. The arbiter decides which of the two outcomes the quote already defined takes effect, and cannot direct USDC to itself or to any third account.

The resolution deadline bounds the arbiter's role in time. The duration is copied into the escrow at funding, the deadline is set at `disputed_at + resolve_seconds`, and from `resolve_until` onwards anyone may call `refund_after_resolution_deadline`. An arbiter that never answers cannot hold either the principal or the unearned fee indefinitely, and the outcome of its silence is known in advance: the payer gets the full debit back.

8.6 Co-signing above the threshold

The vault policy can name a co-signer and a threshold. `book_spend`, the policy check `pay_escrow` shares with `pay`, requires an approval whenever the vault has a co-signer and the amount booked exceeds `cosign_threshold`. For escrow the amount booked is `total_debit`, the principal plus the fee, and the comparison is strict.

The approval is a separate account created in a separate, earlier transaction. A WebAuthn passkey assertion carries authenticator data and client-data JSON besides the signature; the measured passkey approval transaction is 897 bytes and escrow funding with a consumed approval is 1,097 bytes, so one transaction carrying both would exceed the 1,232-byte legacy limit. A separate approval also works with wallets that sign only legacy transactions.

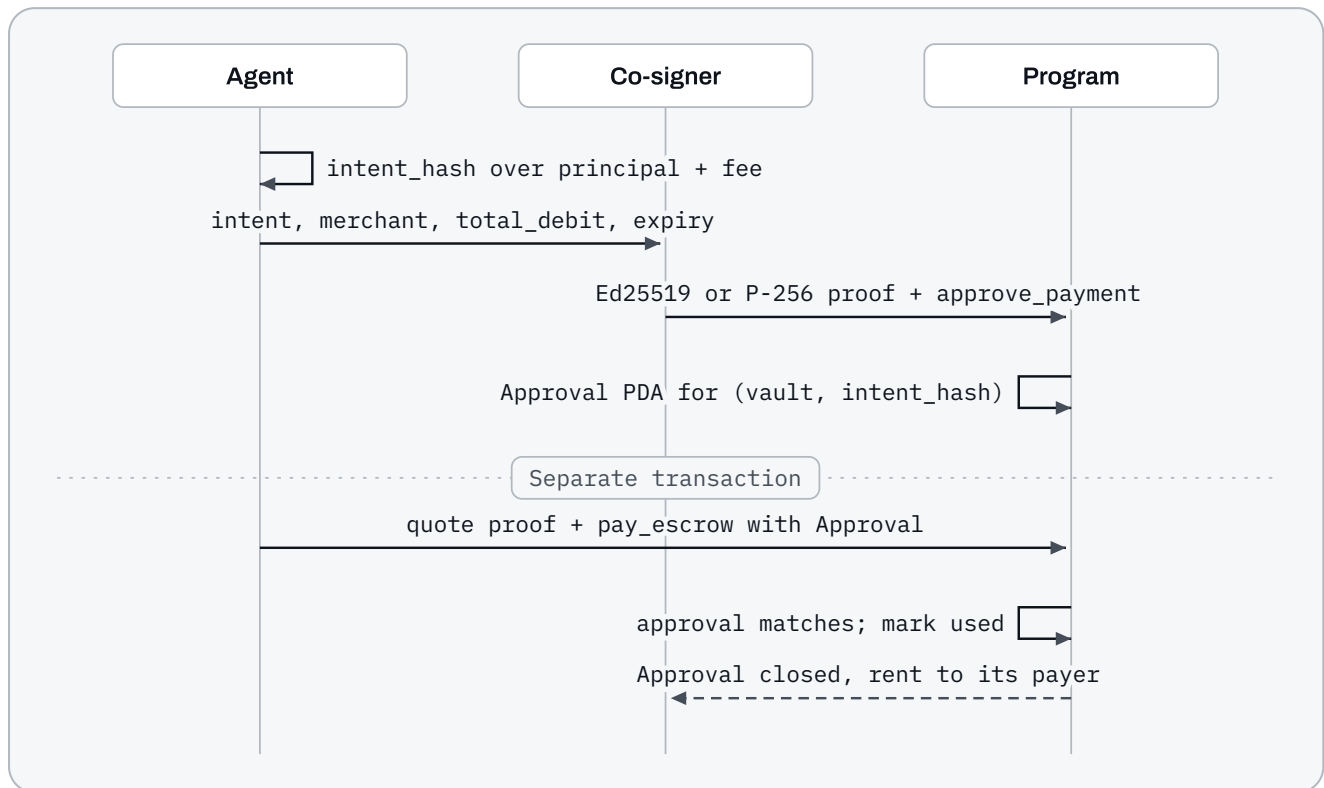


Figure 34. Funding an escrow above the co-sign threshold: the approval is its own transaction and pay_escrow consumes it.

§3.5 describes approvals for every payment lane; the escrow-specific part is what the approval binds. The co-signer signs a 128-byte approval message: an 8-byte tag derived from "Bonded approval v2\0" , the vault, the intent hash, the merchant, the amount, the expiry and the vault's current policy_revision .

approve_payment verifies an Ed25519 signature through the precompile; approve_payment_passkey verifies a P-256 WebAuthn assertion through the secp256r1 precompile against a registered PasskeyCredential . Either one first refuses an intent whose PaymentReceipt already exists, then creates the Approval at [b"approval", vault, intent_hash] , paid by the transaction's payer.

pay_escrow then consumes it only if every bound value matches:

Check	Condition	Error
Present	an Approval account for this intent_hash is supplied	ApprovalRequired
Unused	used == false	ApprovalUsed
Unexpired	now < expiry	ApprovalExpired
Current policy	approval.policy_revision == vault.policy_revision	ApprovalPolicyChanged
Exact payment	intent hash, merchant and amount (total_debit) all equal	ApprovalMismatch

The approval is marked used and closed in the same instruction, its rent returned to whoever paid for it. Because the intent commits to the quote hash, the co-signer has approved one specific job at one specific price including its fee, not a budget the agent may spend as it likes. Any `set_policy` increments the vault's policy revision and voids every outstanding approval at once.

An approval that is never used does not linger. Once its expiry has passed, or once the vault's policy revision has moved past the one it was signed under, `close_approval` closes it, signed by any key, and returns the rent to the original payer. The signed revision also keeps an old proof from recreating the approval under a newer policy, and the intent's `PaymentReceipt`, once it exists, keeps any proof from recreating it after the payment.

The threshold, the per-transaction cap, the merchant budget and the vault's window budget are all checked against the same `total_debit`, in that one booking step. A job that passes the co-signer can still be refused by a cap, and a job under the threshold is still bounded by every budget. A `MerchantRule` with `quote_required` set is satisfied by every escrow, since `pay_escrow` cannot run without a verified quote.

As an example, take a vault whose co-sign threshold is 100 USDC and a merchant rule with a per-transaction cap of 150 USDC. A 120 USDC job has a 300,000 base-unit fee, a total debit of 120,300,000, and needs an approval. So does a job priced at exactly 100 USDC: its total debit of 100,250,000 exceeds the threshold, because the threshold is compared with what leaves the vault.

8.7 Escrow fees

The escrow fee is charged to the payer, in USDC, on top of the principal, and is earned only when the merchant is paid.

$$\text{fee_amount} = \lfloor \text{amount} \times \text{escrow_fee_bps} \div 10\,000 \rfloor$$
$$\text{total_debit} = \text{amount} + \text{fee_amount}$$

`amount` the quote's principal, USDC base units

`escrow_fee_bps` `ProtocolConfig.escrow_fee_bps` at funding, copied into `Escrow.fee_bps`

The multiplication runs in u128 and the division floors, so the fee rounds down, in the payer's favour, and never exceeds the configured rate. A principal below 400 base units pays no fee at 25 bps. The merchant receives exactly the quoted principal on release; the fee is never taken out of it.

The fee moves in three steps. At funding it is reserved: `pay_escrow` debits `total_debit` from the vault, and the vault's policy counts the fee inside every cap, budget and co-sign threshold. While the job is open it sits in the escrow token account with the principal. At the terminal outcome it goes one of two ways: on release it is transferred to the protocol's fee treasury, and on any refund it returns to the vault with the principal. A job the merchant never delivers, or loses before the arbiter, costs the payer nothing in protocol fees.

Governance sets the rate with `configure_escrow_fee(fee_bps, maximum_fee_bps)`, signed by the program's upgrade authority. Both values start unset, and funding fails until they are configured. The first call fixes the maximum, which may not exceed 10,000 bps; every later call must pass the same maximum, and the

fee must not exceed it. `pay_escrow` checks the fee against the maximum again at funding. The parameter files configure 25 bps with a maximum of 100 bps.

Worked through for the 340 USDC job of §8.4:

Quantity	Computation	Base units
Principal	from the quote	340,000,000
Fee	<code>[340,000,000 × 25 ÷ 10,000]</code>	850,000
Total debit	principal + fee	340,850,000
Merchant on release	principal	340,000,000
Fee treasury on release	fee	850,000
Vault on refund	principal + fee	340,850,000

The fee treasury, its buyback, and the tab settlement fee are described in §10.

9. Program architecture

Bonded's on-chain side is one Anchor program, `bonded`. It owns every vault, bond, credit line, default, claim and escrow account; it signs every USDC movement through a program-derived address (PDA); and it checks every off-chain signature (vouchers, merchant quotes, co-signer approvals, passkey assertions) by reading the native precompile instruction placed immediately before it. One program means one audit surface and no cross-program calls between Bonded's own modules. The program has 47 instructions, 15 account types, 10 event types and 61 error codes. This chapter describes how they fit together. Appendix A gives every instruction's arguments, accounts and checks in order, Appendix B every account's byte layout, and Appendix C every error.

9.1 The program

Program ID

The program ID is the `declare_id!` value in `lib.rs`. The source carries the local test ID `Fg6PaFpoGXkYsidMpWxTWqkZqjc5pNFGWET1AG2LskRa`, which the local LiteSVM bank loads directly and `Anchor.toml` lists for localnet. The ID is set per deployment: a deployment build passes `BONDED_BUILD_PROGRAM_ID`, the build helper replaces the single `declare_id!` for the duration of the build, restores the source afterwards and records the ID in `bonded-build.json` in its output directory (`target/deplopy/<cluster>/` for a deployment build).

The ID does more than address the code. Every PDA is derived under it, and the four signed message domains (voucher, quote, payment intent, approval) hash it together with the cluster's genesis hash. A voucher signed for one deployment on one cluster cannot verify against another deployment or another cluster (see §5.3).

Source layout

File	What it holds
<code>lib.rs</code>	The <code>#[program]</code> module with all 47 handlers; protocol governance, operator, bond, credit lines, <code>settle_tab</code> , defaults and claims, freeze and unfreeze; voucher decoding and precompile checks; the <code>BondedError</code> enum
<code>state.rs</code>	The core account structs (<code>ProtocolConfig</code> , <code>Operator</code> , <code>Vault</code> , <code>FreezeCheckpoint</code> , <code>Bond</code> , <code>CreditLine</code> , <code>VoucherAttestation</code> , <code>DefaultEvent</code> , <code>Claim</code>), the program constants and five events
<code>policy.rs</code>	<code>MerchantRule</code> , <code>Approval</code> , <code>PaymentReceipt</code> , <code>QuoteReceipt</code> ; window accounting, float refill, <code>pay</code> , <code>Ed25519</code> approvals, the quote and payment-intent codecs; three events
<code>passkey.rs</code>	<code>PasskeyCredential</code> , the WebAuthn client-data parser and the secp256r1 approval path
<code>escrow.rs</code>	<code>Escrow</code> , its state machine, <code>EscrowEvent</code> and the <code>EscrowError</code> enum
<code>buyback.rs</code>	<code>configure_buyback</code> , <code>claim_buyback</code> and <code>burn_buyback</code> : the fee account's one way out, a claim, a swap and a burn checked together through the instructions <code>sysvar</code> (§10.7); its two events
<code>vector_tests.rs</code>	Rust tests that run the program's own decoder, eligibility and payout functions against golden vectors generated by the TypeScript contract

Build

The crate depends on `anchor-lang 0.31.1` (with `init-if-needed`), `anchor-spl 0.31.1` (`token`, `associated_token`; `token_2022` is enabled only so Anchor's IDL generator compiles, while every executable context requires `Program<Token>`), and `serde / serde_json` without the standard library, used only to parse WebAuthn client data. `npm run build:program` runs `programs/bonded/build.mjs`, which calls `cargo-build-sbf` with the installed `platform-tools v1.52` compiler and the flags `--skip-tools-install --no-rustup-override --offline`, writing `target/deploy/bonded.so`. `Anchor.toml` pins Anchor 0.31.1 and Solana 3.1.14. `idl.json` is generated from the same sources with the `idl-build` feature.

Encoding conventions

- Instruction data starts with the first 8 bytes of SHA-256 of `global:<instruction_name>`; account data with SHA-256 of `account:<AccountName>`; event data with SHA-256 of `event:<EventName>`.
- Fields are Borsh, little endian. An `Option` is a one-byte tag (0 or 1) followed by its value when present.
- Amounts are `u64` USDC base units (6 decimals). Account timestamps are `i64` Unix seconds; vouchers and quotes carry `u64` expiries on the wire.
- Every addition and subtraction on balances, counters and timestamps is checked and fails with `ArithmeticOverflow`; every product of two amounts or of an amount and basis points is computed in `u128`.
- Every program account begins with its 8-byte discriminator and a 1-byte PDA bump. There is no layout-version byte.

9.2 Accounts

The program owns 15 account types. Two more addresses complete the set: the fee token account, a token account at the PDA `[b"fees"]`, and the freeze reserve, a system-owned PDA that holds lamports for freeze checkpoint rent. The vault, bond and escrow balances sit in classic SPL associated token accounts whose authority is the respective PDA (§9.5).

The table below lists each account with its seeds, its allocated size and the rent-exempt minimum for that size. Sizes are `8 + INIT_SPACE`, which reserves the full payload of every `Option`. Rent follows Solana's formula [28]:

$$\text{rent_exempt_lamports} = (\text{size} + 128) \times 3\,480 \times 2$$

`size` allocated account bytes, discriminator included

Account	Seeds	Bytes	Rent (lamports)	Created by, rent payer	Closed by, rent to
ProtocolC onfig	<code>b"protocol"</code>	286	2,881,440	<code>initialize_protocol</code> , upgrade authority	never
Fee token account	<code>b"fees"</code>	165	2,039,280	<code>initialize_protocol</code> , upgrade authority	never

Account	Seeds	Bytes	Rent (lamports)	Created by, rent payer	Closed by, rent to
Operator	b"operator" , admin	150	1,934,880	create_operator , admin	never
Bond	b"bond" , operator	141	1,872,240	create_operator , admin	never
Vault	b"vault" , operator, agent_id [32]	351	3,333,840	create_vault , admin	never
MerchantRule	b"rule" , vault, merchant	106	1,628,640	set_merchant_rule , admin	never; updated in place
PasskeyCredential	b"passkey" , vault, credential_id_hash	174	2,101,920	set_passkey_credential , admin	never
Approval	b"approval" , vault, intent_hash	162	2,018,400	approve_payment , approve_payment_passkey , any payer	pay , pay_escrow on consumption, or close_approval ; to the recorded payer
PaymentReceipt	b"payment" , vault, intent_hash	211	2,359,440	pay (payer), pay_escrow (session key)	never; replay guard
QuoteReceipt	b"quote" , vault, quote_hash	138	1,851,360	quoted pay (payer), pay_escrow (session key)	never; replay guard
Escrow	b"escrow" , vault, quote_hash	400	3,674,880	pay_escrow , session key	every terminal escrow instruction; to the session key that funded it
CreditLine	b"line" , bond, merchant	302	2,992,800	open_line , admin	never; renewed in place
VoucherAttestation	b"voucher- attestation" , line, outgoing key, effective_at i64 LE	145	1,900,080	attest_line_voucher , merchant	never; archival
DefaultEvent	b"default" , bond, sequence u64 LE	156	1,976,640	settle_tab , merchant	healthy settlement closes its placeholder at once, to the merchant; a real default stays open
Claim	b"claim" , default event, merchant	170	2,074,080	file_claim , merchant	execute_default_payout or close_claim , to the merchant
FreezeCheckpoint	b"freeze" , vault, generation u64 LE	89	1,510,320	freeze , freeze reserve or freezing signer	close_freeze_checkpoint , to the freeze reserve
Freeze reserve	b"freeze-reserve" , vault	0	n/a	fund_freeze_reserve , admin	never; system-owned

Seeds bind cardinality. One admin key has one `Operator`, which has one `Bond`. An operator has a `Vault` per 32-byte `agent_id` (the TypeScript builder hashes a UTF-8 agent ID with SHA-256). A bond has one `CreditLine` per merchant, and that line names the single vault that pays its tab. Each default has a sequence number; each default has at most one `Claim` per merchant.

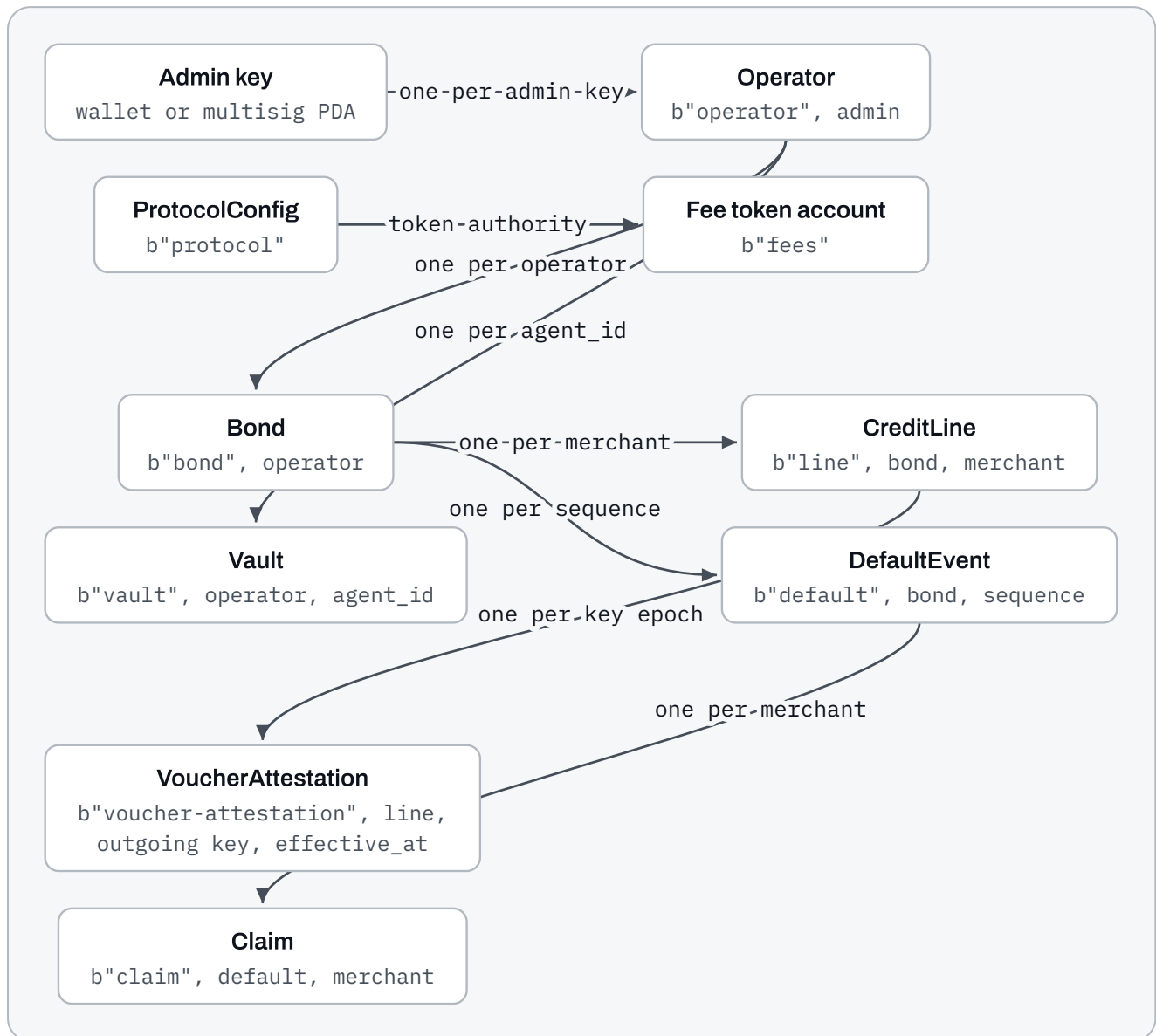


Figure 35. PDAs on the operator side, each derived from its parent's address.

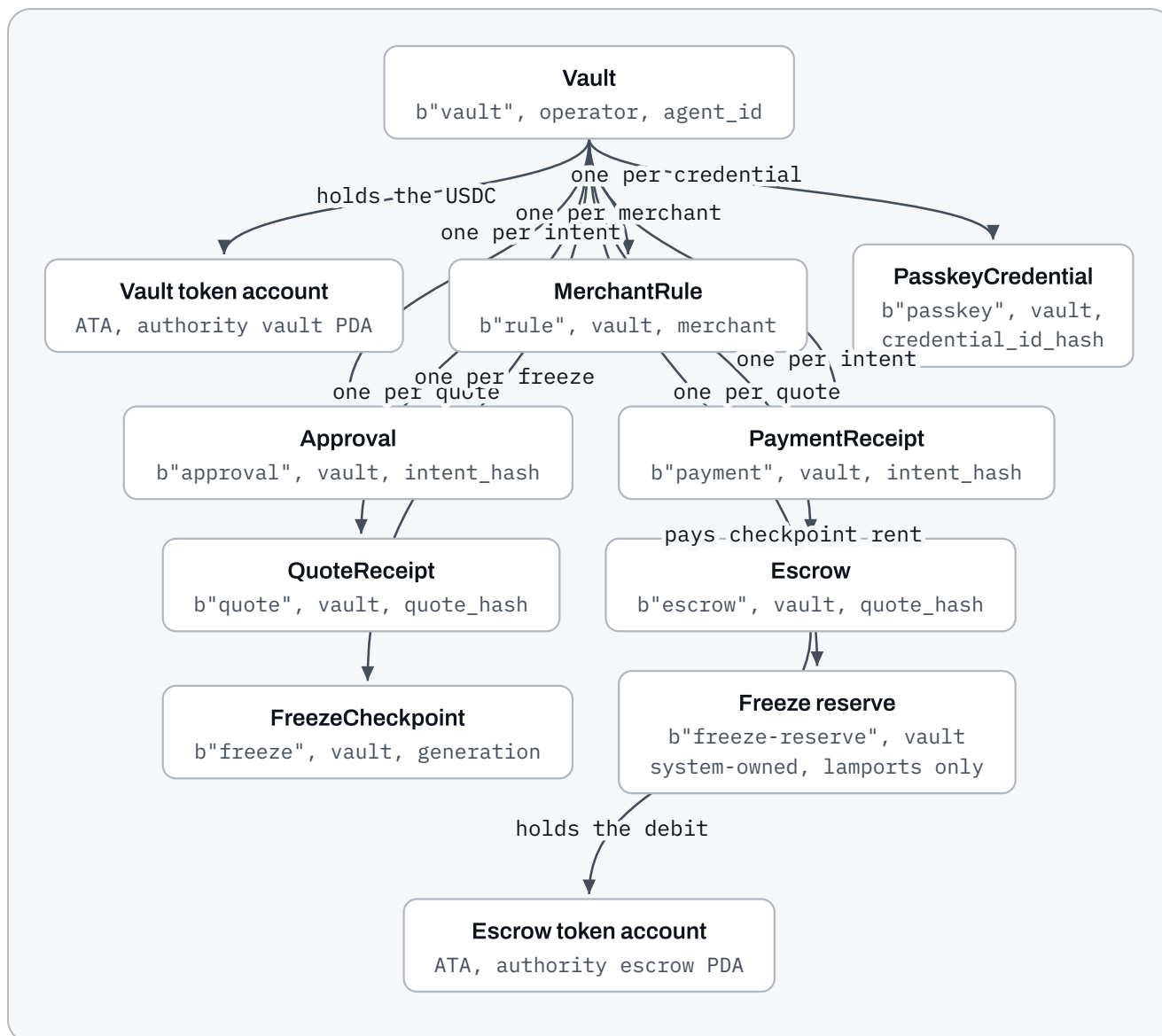


Figure 36. PDAs derived from one vault, and the token accounts that two of them own.

What each account is for

ProtocolConfig is the singleton that fixes the mint, the fee account, the fees and the time parameters every other instruction reads (§9.3). It also tracks `total_bond_balance`, the sum of booked bond collateral, against the 250,000 USDC beta total.

Operator is the operator's identity: its `admin`, its operator-level `guardian`, its reputation counters `tabs_settled` and `defaults_count`, and two collateral locks, `frozen_vault_count` and `recourse_lock_until`. Any frozen vault, or any unfreeze tail still running, blocks bond withdrawals, decrease application and renewals across the whole operator.

Bond holds the collateral book: `balance` (booked deposits less payouts and withdrawals, so a token donation is never collateral), `reserved_lines` (the sum of line limits), `minimum_coverage_bps`, `longest_line_expiry` (a high-water mark that shortening a line never lowers), the pending `withdrawal`, and the default counters `default_sequence` and `active_default_sequence`.

Vault holds one agent's policy (`PolicyArgs`: session key, float cap, window length and budget, co-signer, co-sign threshold, guardian), the window counters `window_started_at`, `window_spent` and `last_approved`, the freeze state and episode counters, `policy_revision`, and a pending guardian change.

CreditLine is the merchant's line: `limit` and a `pending_limit` decrease, `settled_cumulative` and `retired_cumulative` (the two replay floors), `maximum_nonce`, `expires_at`, the voucher keys (`voucher_session_key`, `pending_voucher_key`, `previous_voucher_key`, `rotated_at`), the default epoch it belongs to (`opened_default_sequence`), its first freeze reference (`freeze_generation_at_open`) and its immutable `tab_fee_bps`. The line PDA is persistent: after a default it is renewed in place, and its watermarks prevent the same voucher from being recovered twice.

VoucherAttestation archives a merchant's acknowledgement of the latest voucher signed by an outgoing line key, during the 72-hour notice before a key switch: its maximum nonce and cumulative amount become ceilings for old-key vouchers after the switch (see §5.9).

DefaultEvent and **Claim** carry one default from opening through payout: window end, claim count and total, the payout snapshot (`payout_pool`, `eligible_total_snapshot`), `total_paid`, and per claim the voucher scalars, the settled and limit snapshots, eligibility and paid amount (see §7).

FreezeCheckpoint is immutable evidence of one freeze: generation, freeze time, the unfreeze time and recourse end written once by `unfreeze`, the operator's `defaults_count` at freeze (which orders a freeze against a default opened in the same second), and the episode it belongs to. Claims cite checkpoints to prove which freeze protected a line (see §12.7).

MerchantRule is an allowlist entry: per-transaction cap, window budget and spent amount, and whether a merchant-signed quote is required. **Approval** binds one co-signed payment intent. **PaymentReceipt** and **QuoteReceipt** are permanent single-use guards keyed by intent hash and quote hash; they also record the lane and outcome of each program payment. **PasskeyCredential** stores a registered P-256 key with its relying-party and origin hashes and its signature counter. **Escrow** holds one job's quote commitment, principal, reserved fee, deadlines, hashes, arbiter and state (see §8).

Optional fields and reading accounts

Allocation reserves the maximum payload of every `Option`, but Borsh writes an absent value as its single tag byte and moves every later field up. `Vault.pending_guardian`, `Bond.withdrawal`, `Bond.active_default_sequence`, `CreditLine.pending_limit`, `CreditLine.pending_voucher_key`, `CreditLine.previous_voucher_key` and the escrow and buyback options in `ProtocolConfig` therefore shift the offsets of the fields after them. Readers decode an account field by field from the discriminator, as the indexer's codec does, rather than reading fixed offsets past the first `Option`. Appendix B gives offsets with every option present.

Lifetimes

Accounts that describe standing relationships (operator, bond, vault, rule, line, credential, attestation) never close. Accounts that describe one obligation close when it ends, and their rent goes to a recipient fixed in advance: an approval on consumption or expiry, to its recorded payer; an escrow and its token account at the terminal transition, to the session key that funded them; a claim at payout, to its merchant; a freeze checkpoint once no claim window can cite it, to the vault's freeze reserve. The two receipts stay open on purpose. A closed account can be re-created at the same address, so a closed approval or escrow alone could not stop the same intent or quote from being replayed; the persistent receipt at `[b"payment", vault, intent_hash]` does, because `pay` and `pay_escrow` create it with `init` and fail if it exists (see §12.3).

9.3 Protocol configuration

`ProtocolConfig` lives at `[b"protocol"]` and is created once by `initialize_protocol`, signed by the program's upgrade authority. The table below lists its fields, who sets each, whether it can change, and the value in the production parameter file. The local stack uses the same values except a one-second withdrawal margin and its own test mint and genesis hash.

Field	Meaning	Set by	Can change	Production value
<code>authority</code>	Stored copy of the upgrade authority	<code>initialize_protocol</code>	Rewritten to the signer by every governed update	the deployer key
<code>usdc_mint</code>	The classic SPL mint; must have 6 decimals	<code>initialize_protocol</code>	No	<code>EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v</code>
<code>fee_treasury</code>	Address of the <code>[b"fees"]</code> token account	<code>initialize_protocol</code>	No	derived
<code>total_bond_balance</code>	Sum of booked bond balances	<code>post_bond</code> , <code>finalize_withdrawal</code> , <code>execute_default_payout</code>	Program-maintained	0 at start
<code>tab_fee_bps</code>	Fee on settled tab deltas, for lines opened from now on	<code>initialize_protocol</code> , <code>update_protocol</code>	Yes, $\leq$ <code>maximum_tab_fee_bps</code>	10
<code>tab_fee_withdrawn</code>	New lines snapshot a 0 bps fee	<code>initialize_protocol</code> , <code>update_protocol</code>	Yes	true
<code>withdrawal_margin_seconds</code>	Added to every bond withdrawal unlock	<code>initialize_protocol</code>	No; must be > 0	86,400 (24 h)
<code>maximum_tab_fee_bps</code>	Ceiling on <code>tab_fee_bps</code>	<code>initialize_protocol</code>	No; $\leq 10,000$	100
<code>freeze_grace_seconds</code>	Frozen time before a merchant may force a default	<code>initialize_protocol</code>	No; $\geq 345,600$	345,600 (96 h)
<code>cluster_genesis_hash</code>	Binds every signed domain to one cluster	<code>initialize_protocol</code>	No	read from the cluster at initialization
<code>voucher_domain</code>	8-byte voucher tag, derived	<code>initialize_protocol</code>	No	derived
<code>escrow_fee_bps</code>	Fee reserved on escrow principal	<code>configure_escrow_fee</code>	Yes, $\leq$ maximum	25
<code>maximum_escrow_fee_bps</code>	Ceiling on the escrow fee	first <code>configure_escrow_fee</code>	No once set; $\leq 10,000$	100
<code>escrow_resolution_seconds</code>	Arbiter's time after a dispute	<code>configure_escrow_resolution</code>	Yes; must be > 0	259,200 (72 h)
<code>buyback_token_mint</code>	The \$BONDED mint the fees buy and burn	<code>configure_buyback</code>	Yes, while no buyback is pending	set when the mint is configured

Field	Meaning	Set by	Can change	Production value
<code>buyback_crank</code>	The one key that may run a buyback	<code>configure_buyback</code>	Yes, while no buyback is pending	the worker's keeper key
<code>buyback_minimum_usdc</code>	The smallest claim a buyback may take	<code>configure_buyback</code>	Yes; must be > 0	10,000,000 (10 USDC)
<code>pending_buyback</code>	Amount, minimum output, the crank's balances before, the claim's index	<code>claim_buyback</code> , cleared by <code>burn_buyback</code>	Exists only inside one transaction	none

The voucher tag is computed once at initialization and stored, so `settle_tab` , `file_claim` and `attest_line_voucher` compare 8 bytes instead of hashing on every call. Quote, intent and approval tags are derived the same way from their own prefixes on use.

```
voucher_domain = SHA-256("Bonded voucher v2\0" || program_id ||
                           cluster_genesis_hash)[0..8]
```

`program_id` , `cluster_genesis_hash` raw 32-byte values

The three escrow options start empty. `pay_escrow` refuses to fund (`InvalidConfiguration`) until the fee, its maximum and a positive resolution duration are set, so no escrow ever runs under an implicit fee or an unbounded dispute. The production initialization sends `configure_escrow_fee` and `configure_escrow_resolution` in the transaction after `initialize_protocol` .

Governance changes never reprice accepted obligations. `open_line` copies the effective tab fee into `CreditLine.tab_fee_bps` (0 when waived) and the line keeps it through renewals. `pay_escrow` copies `fee_bps` , the resulting `fee_amount` and `resolve_seconds` into the `Escrow` . A changed rate applies to new lines and new escrows only.

Some values are program constants rather than configuration:

Constant	Value	Role
<code>CLAIM_WINDOW</code>	259,200 s (72 h)	Claim window of a default; part of every withdrawal unlock and unfreeze tail
<code>DECREASE_DELAY</code>	259,200 s (72 h)	Notice for a line limit decrease, a line key rotation and a guardian rotation
<code>MIN_FREEZE_GRACE_SECONDS</code>	345,600 s (96 h)	Floor for <code>freeze_grace_seconds</code> : 72 h to replace a guardian plus 24 h to respond
<code>DISPUTE_WINDOW_SECONDS</code>	172,800 s (48 h)	Dispute window after <code>mark_delivered</code>
<code>BOND_CAP</code>	25,000,000,000 (25,000 USDC)	Maximum booked balance of one bond

Constant	Value	Role
TOTAL_CAP	250,000,000,000 (250,000 USDC)	Maximum total_bond_balance
COVERAGE_FLOOR_BPS	3,300	Lowest minimum_coverage_bps an operator may choose
COVERAGE_DEFAULT_BP S	10,000	The shared default the clients propose; create_operator takes the minimum as an argument
VOUCHER_DOMAIN_PREF IX	"Bonded voucher v2\0"	Voucher tag prefix

9.4 The instruction set

Every instruction follows the same order. Anchor first loads each declared account and checks its PDA seeds, its owner and discriminator, has_one relations, fixed addresses and signer flags; a substituted account fails here, before any handler code runs. The handler then checks state in a fixed order, writes, performs its token CPIs and emits its events. The tables below group the 47 instructions by area; Appendix A gives each one in full.

Signers fall into fixed roles. The **admin** signs everything that changes an operator's policy, collateral or lines. The **session key** signs program payments and escrow funding. The **merchant** signs settlement, claims, delivery and attestations. The **co-signer** never signs a transaction: its signature arrives inside an Ed25519 or secp256r1 precompile instruction. The **guardian** can sign freeze and nothing else. The **crank**, one key named in ProtocolConfig, signs the two buyback instructions and nothing else. **Permissionless** instructions take either any signer or no signer account at all, and the transaction fee payer runs them.

Protocol

Instruction	Signer	Writes	Checks	Emits
initialize_protocol	upgrade authority (pays rent)	protocol (init), fee_treasury (init)	ProgramData authority is the signer; maximum ≤ 10,000; fee ≤ maximum; margin > 0; grace ≥ 345,600; mint has 6 decimals	none
update_protocol	upgrade authority	protocol	ProgramData authority; fee ≤ maximum_tab_fee_bps	Protocol Updated
configure_escrow_fee	upgrade authority	protocol	maximum ≤ 10,000; fee ≤ maximum; maximum equals the stored one once set	Protocol Updated
configure_escrow_resolution	upgrade authority	protocol	duration > 0	Protocol Updated
configure_buyback_k	upgrade authority	protocol	ProgramData authority; minimum > 0; a crank key; no buyback pending; the mint is SPL Token or Token-2022 with only metadata extensions	BuybackConfigured
claim_buyback	crank	protocol, fee_treasury, crank_usdc	the configured crank; nothing pending; minimum ≤ amount ≤ the fee account; a matching burn_buyback later in the transaction	none
burn_buyback	crank	protocol, token_mint, crank_tokens	the matching claim earlier in the transaction; the claimed USDC spent; gain ≥ the minimum	BuybackBurned

Vault

Instruction	Signer	Writes	Checks	Emits
<code>create_vault</code>	admin (pays rent)	<code>vault</code> (init), <code>vault_token</code> (ATA)	session key set; window > 0; float cap ≤ window budget	<code>CoreEvent</code> 2
<code>set_policy</code>	admin	<code>operator</code> , <code>vault</code> , <code>vault_token</code>	same policy checks; guardian unchanged	<code>CoreEvent</code> 3
<code>rotate_session_key</code>	admin	<code>operator</code> , <code>vault</code> , <code>vault_token</code>	key is not zero	<code>CoreEvent</code> 24
<code>withdraw</code>	admin	<code>vault</code> , <code>vault_token</code> , <code>destination</code>	vault not frozen; amount > 0	<code>CoreEvent</code> 25
<code>set_merchant_rule</code>	admin (pays rent)	<code>rule</code> (init if needed)	operator owns the vault	<code>CoreEvent</code> 21
<code>refill_float</code>	none	<code>vault</code> , <code>vault_token</code>	not frozen; delegate within <code>last_approved</code> ; spent ≤ budget	<code>CoreEvent</code> 20
<code>pay</code>	session key, payer	<code>vault</code> , <code>vault_token</code> , <code>rule</code> , <code>merchant_token</code> , <code>payment_receipt</code> (init), <code>quote_receipt</code> , approval	quote proof if quoted; intent hash; cap, budgets, approval	<code>PaymentCommitted</code> , <code>CoreEvent</code> 23
<code>approve_payment</code>	payer	<code>approval</code> (init)	no receipt for the intent; current revision; Ed25519 proof by the co-signer over the 128-byte approval	<code>ApprovalCreated</code> , <code>CoreEvent</code> 22
<code>approve_payment_passkey</code>	payer	<code>credential</code> , <code>approval</code> (init)	as above, with a WebAuthn assertion and secp256r1 proof	<code>ApprovalCreated</code> , <code>CoreEvent</code> 22
<code>set_passkey_credential</code>	admin (pays rent)	<code>credential</code> (init if needed)	compressed key; a repeat registration is identical	none
<code>close_approval</code>	any	<code>approval</code> (closed), <code>rent_payer</code>	expired, or issued under an older policy revision	<code>ApprovalClosed</code>

`set_policy` , `rotate_session_key` , `withdraw` and `freeze` all first reconcile the window against the token account's real `delegated_amount` , then revoke the float delegate and set `last_approved` to zero, so no instruction ever books a consumed allowance twice or leaves a stale delegate behind. `set_policy` also increments `policy_revision` , which invalidates every unconsumed approval. A guardian change goes through the delayed rotation instructions (`GuardianRotationRequired` otherwise).

The window check reserves the outstanding float before any program payment:

$$\begin{aligned}\text{consumed} &= \text{last_approved} - \text{delegated_amount} \\ \text{window_spent} + \text{amount} + \text{delegated_amount} &\leq \text{window_budget} \\ \text{rule.window_spent} + \text{amount} &\leq \text{rule.window_budget} \\ \text{allowance} &= \min(\text{float_cap}, \text{window_budget} - \text{window_spent})\end{aligned}$$

<code>delegated_amount</code>	the SPL delegate allowance still unspent
<code>allowance</code>	what <code>refill_float</code> approves for the session key

Example: a vault has a 200 USDC window budget, 37 USDC already spent and 20 USDC of float still delegated. A `pay` of 150 USDC passes a 150 USDC merchant cap and a 400 USDC merchant budget, then fails the vault check, $37 + 150 + 20 = 207 > 200$, with `WindowBudgetExceeded`, before the co-signer threshold is even considered.

A payment above the co-sign threshold needs an `Approval` created in an earlier transaction. A `WebAuthn` assertion carries several hundred bytes of authenticator and client data; beside a merchant quote and `pay` it would crowd the 1,232-byte legacy limit, and the measured passkey approval transaction alone is 897 bytes (§9.7). The separate account also lets any wallet carry the approval.

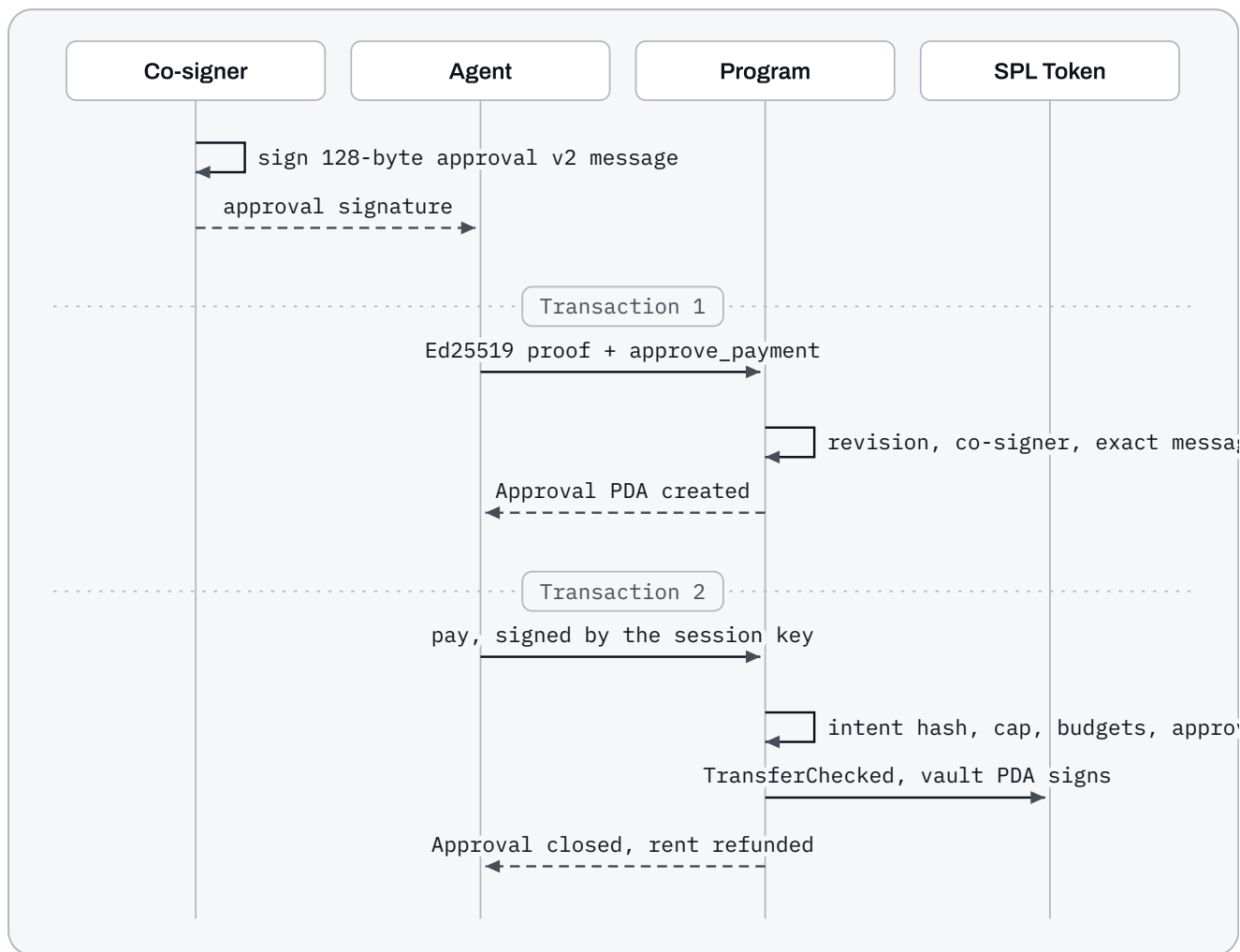


Figure 37. A payment above the co-sign threshold takes two transactions.

Bond and tab

Instruction	Signer	Writes	Checks	Emits
<code>create_operator</code>	admin (pays rent)	<code>operator</code> , <code>bond</code> (init), <code>bond_token</code> (ATA)	minimum coverage $\geq 3,300$ bps	<code>CoreEvent</code> 1
<code>post_bond</code>	admin	<code>protocol</code> , <code>bond</code> , <code>source</code> , <code>bond_token</code>	amount > 0 ; bond $\leq 25,000$ USDC; total $\leq 250,000$ USDC	<code>CoreEvent</code> 4
<code>open_line</code>	admin (pays rent)	<code>bond</code> , <code>line</code> (init)	vault not frozen; no active default; limit > 0 ; expiry in the future; coverage	<code>CoreEvent</code> 5
<code>resize_line</code>	admin	<code>bond</code> , <code>line</code>	no active default; a decrease is scheduled, or applied after 72 h with collateral unlocked; an increase needs an unfrozen vault, a current line and coverage	<code>CoreEvent</code> 6 or 7
<code>renew_line</code>	admin	<code>bond</code> , <code>line</code>	vault not frozen; no active default; collateral unlocked; new expiry $\geq$ old and in the future; coverage	<code>CoreEvent</code> 17
<code>rotate_line_key</code>	admin	<code>line</code>	key not zero and not current; 72 h notice	<code>CoreEvent</code> 18 or 19

Instruction	Signer	Writes	Checks	Emits
<code>attest_line_voucher</code>	merchant (pays rent)	<code>voucher_attestation</code> (init if needed)	a rotation is pending and not yet effective; outgoing-key proof; advances the line floors; within the limit; ceilings only rise	<code>CoreEvent</code> 26
<code>settle_tab</code>	merchant (pays placeholder rent)	<code>operator</code> , <code>vault_token</code> , <code>bond</code> , <code>line</code> , <code>default_event</code> , <code>merchant_token</code> , <code>fee_treasury</code>	proof placement and bytes; signing key; line epoch and expiry; voucher expiry; nonce and cumulative advance; $\Delta \leq$ limit	<code>CoreEvent</code> 8, or 9 when a default opens
<code>request_withdrawal</code>	admin	<code>bond</code>	collateral unlocked; no active default; none pending; $\text{amount} \leq \text{balance}$; coverage after it	<code>CoreEvent</code> 13
<code>finalize_withdrawal</code>	admin	<code>protocol</code> , <code>bond</code> , <code>bond_token</code> , <code>destination</code>	collateral unlocked; no active default; unlock time reached; coverage	<code>CoreEvent</code> 14

Coverage is one exact inequality, checked whenever collateral falls or exposure rises. A pending withdrawal no longer counts as collateral for new exposure:

$$\begin{aligned} \text{available} &= \text{bond.balance} - \text{pending_withdrawal.amount} \\ \text{available} \times 10\,000 &\geq \text{reserved_lines} \times \text{minimum_coverage_bps} \\ \text{unlock_at} &= \max(\text{now}, \text{longest_line_expiry}) + 259\,200 + \\ &\quad \text{withdrawal_margin_seconds} \end{aligned}$$

`reserved_lines` Σ limit over the bond's lines

`unlock_at` earliest `finalize_withdrawal`; `open_line` and `renew_line` push a pending one later

`settle_tab` is the one instruction a merchant runs routinely. Its transaction carries the voucher's Ed25519 proof immediately before it; the program reads that proof through the instructions sysvar, then either pays the delta or opens a default (see §5.6 and §7.1).

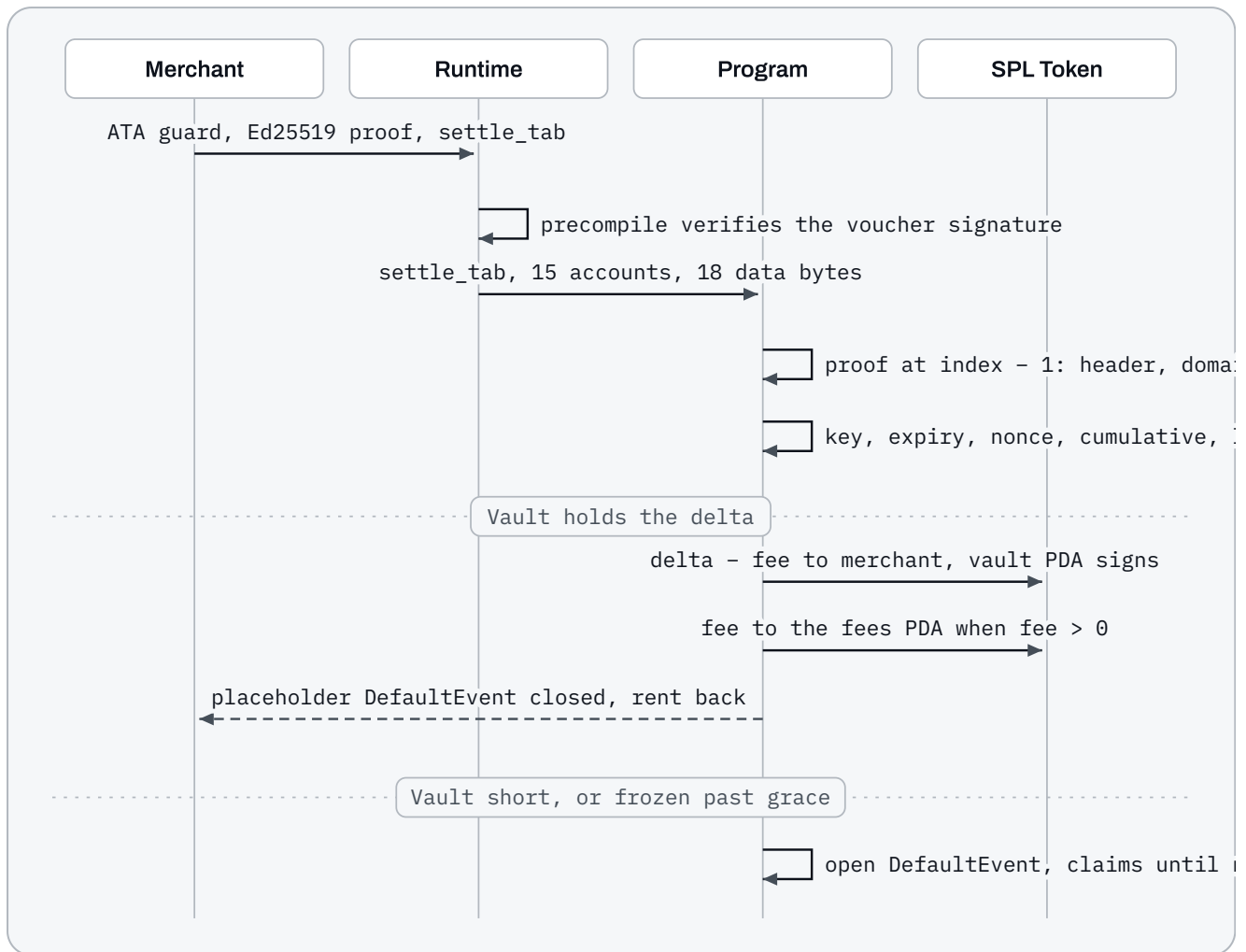


Figure 38. One `settle_tab` transaction, from the proof instruction to the transfer.

$$\text{floor} = \max(\text{settled_cumulative}, \text{retired_cumulative})$$

$$\text{delta} = \text{voucher.cumulative} - \text{floor} \leq \text{line.limit}$$

$$\text{fee} = \lfloor \text{delta} \times \text{line.tab_fee_bps} \div 10\,000 \rfloor$$

delta what this voucher adds over what the line has already paid or retired

fee kept in the fees PDA; the merchant receives $\text{delta} - \text{fee}$

The `default_event` account is created with `init_if_needed` at `[b"default", bond, bond.default_sequence + 1]` on every settlement. A healthy settlement closes it again in the same instruction and returns the rent to the merchant; only a real default keeps it. This keeps the account list identical on both paths, so the merchant never has to predict whether the vault is short.

Default

Instruction	Signer	Writes	Checks	Emits
<code>file_claim</code>	merchant (pays rent)	<code>default_event</code> , <code>claim</code> (init if needed)	claim window open; event is the bond's active default; line belongs to the epoch before it; proof; key and expiry at the right reference; replay floors; a refiling advances both nonce and amount	<code>ClaimFile</code> <code>d</code> , <code>CoreEvent</code> 10
<code>execute_default_payout</code>	none	<code>protocol</code> , <code>bond</code> , <code>bond_token</code> , <code>default_event</code> , <code>claim</code> (closed), <code>line</code> , <code>merchant</code> , <code>merchant_token</code>	window ended; event active and unpaid	<code>CoreEvent</code> 11, and 12 after the last claim
<code>finalize_empty_default</code>	none	<code>bond</code> , <code>default_event</code>	window ended; zero claims	<code>CoreEvent</code> 12
<code>close_claim</code>	merchant	<code>claim</code> (closed)	claim paid; line watermarks cover it	none
<code>close_default_placeholder</code>	merchant	<code>default_event</code> (closed)	<code>opened_at</code> is 0; merchant is the recorded rent payer	none

The first payout fixes the pool and the denominator; each later payout reads the same snapshot, so the result does not depend on the order in which the keeper pays claims:

$$\text{eligible} = \min(\max(\text{voucher} - \text{settled_snapshot}, 0), \text{line_limit_snapshot})$$

$$\text{payout} = \text{eligible}, \text{ when } \sum \text{eligible} \leq \text{payout_pool}$$

$$\text{payout} = \lfloor \text{payout_pool} \times \text{eligible} \div \sum \text{eligible} \rfloor, \text{ otherwise}$$

`payout_pool` `bond.balance` at the first payout

$\sum \text{eligible}$ `claims_total` at the first payout

Example: a 1,000 USDC pool and three claims eligible for 600, 500 and 400 USDC (1,500 in total). The payouts are 400,000,000, 333,333,333 and 266,666,666 base units. One base unit stays in the bond; floor rounding never pays out more than the pool. Each payout also lifts the line's `retired_cumulative` to the full claimed voucher amount and its `maximum_nonce` to the voucher nonce, so the unpaid fraction can never be claimed again (see §7.3–§7.4).

Escrow

Instruction	Signer	Writes	Checks	Emits
<code>pay_escrow</code>	session key (pays rent)	<code>vault</code> , <code>vault_token</code> , <code>rule</code> , <code>escrow</code> (init), <code>escrow_token</code> (ATA), <code>payment_receipt</code> (init), <code>quote_receipt</code> , <code>approval</code>	escrow fee and resolution configured; quote proof in lane 1; <code>arbitrator</code> ≠ merchant; <code>now</code> ≤ <code>deliver_by</code> ; intent over the total debit; cap, budgets, approval	<code>PaymentCommitted</code> , <code>EscrowEvent</code>
<code>mark_delivered</code>	merchant	<code>escrow</code>	state <code>Funded</code> ; <code>now</code> ≤ <code>deliver_by</code>	<code>EscrowEvent</code>
<code>dispute</code>	session key or admin	<code>escrow</code>	state <code>Delivered</code> ; <code>arbitrator</code> named; <code>now</code> < <code>dispute_until</code>	<code>EscrowEvent</code>
<code>accept</code>	session key or admin	escrow and token accounts	state <code>Delivered</code>	<code>EscrowEvent</code>
<code>release_after_window</code>	any	escrow and token accounts	state <code>Delivered</code> ; <code>now</code> ≥ <code>dispute_until</code>	<code>EscrowEvent</code>
<code>resolve</code>	the named arbitrator	escrow and token accounts	state <code>Disputed</code> ; <code>now</code> < <code>resolve_until</code>	<code>EscrowEvent</code>
<code>refund_on_timeout</code>	any	escrow and token accounts	state <code>Funded</code> ; <code>now</code> > <code>deliver_by</code>	<code>EscrowEvent</code>
<code>refund_after_resolution_deadline</code>	any	escrow and token accounts	state <code>Disputed</code> ; <code>now</code> ≥ <code>resolve_until</code>	<code>EscrowEvent</code>

The five terminal instructions share one account context (`FinishEscrow`) and one routine. A release pays `amount` to a token account that the routine itself checks is a classic SPL account of the configured mint owned by `escrow.merchant` , pays `fee_amount` to the fees PDA and returns any donated surplus to the vault. A refund returns the whole escrow balance, principal and reserved fee, to the vault without touching the merchant account, so a merchant that closes its destination cannot block a refund. Both close the escrow token account and the `Escrow` to the recorded rent payer and write the outcome into the persistent `PaymentReceipt` (see §8.3).

$$\text{fee_amount} = \lfloor \text{amount} \times \text{escrow_fee_bps} \div 10\,000 \rfloor$$

$$\text{total_debit} = \text{amount} + \text{fee_amount}$$

`amount` the quote principal

`total_debit` what caps, budgets and the approval bind, and what the escrow holds

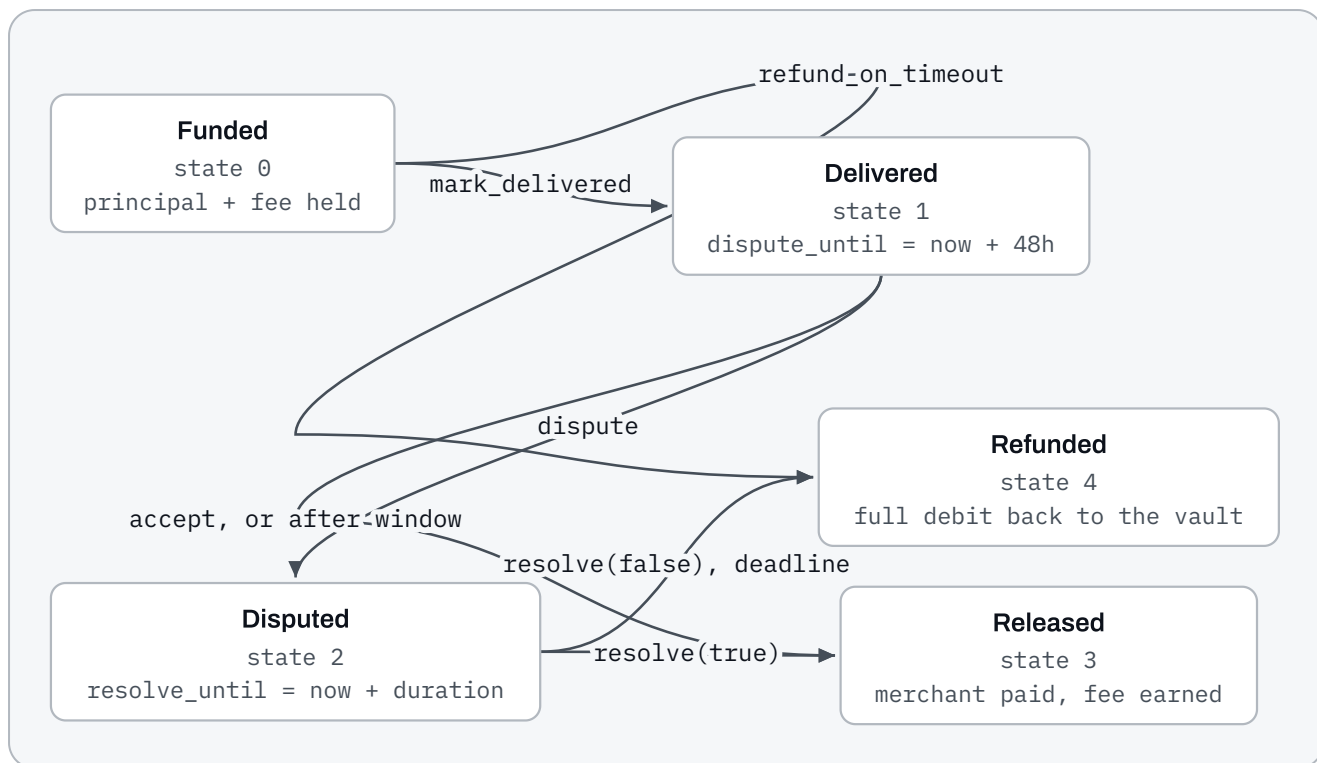


Figure 39. Escrow states and the instruction that moves each one; terminal states close the account.

Guardian and freeze

Instruction	Signer	Writes	Checks	Emits
<code>freeze</code>	admin, operator guardian or vault guardian	<code>operator</code> , <code>vault</code> , <code>vault_token</code> , <code>freeze_checkpoint</code> , <code>freeze_reserve</code>	signer role; checkpoint is the canonical empty PDA; rent from the reserve or the signer	<code>CoreEvent 15</code>
<code>unfreeze</code>	admin	<code>operator</code> , <code>vault</code> , <code>vault_token</code> , <code>freeze_checkpoint</code>	checkpoint is the vault's current one and not yet finalized	<code>CoreEvent 16</code>
<code>fund_freeze_reserve</code>	admin	<code>freeze_reserve</code>	amount > 0	none
<code>close_freeze_checkpoint</code>	none	<code>freeze_checkpoint</code> (closed), <code>freeze_reserve</code>	from an older episode: 72 h have passed since the current episode began; from the current episode: the vault is unfrozen and 72 h have passed since <code>recourse_until</code>	none
<code>rotate_operator_guardian</code>	admin	<code>operator</code>	new key differs; the same key again applies it after 72 h	<code>GuardianRotated</code>
<code>rotate_vault_guardian</code>	admin	<code>vault</code>	as above	<code>GuardianRotated</code>

A first `freeze` increments `freeze_generation` , creates the checkpoint for that generation, revokes the float delegate and raises `operator.frozen_vault_count` . A repeated `freeze` on a frozen vault reuses the same checkpoint and generation. The checkpoint is created by the handler (system `transfer` , `allocate` ,

`assign`, signed with the checkpoint's own seeds) rather than by Anchor's `init`, so that its rent can come from the vault's reserve: a guardian key holding no lamports can freeze while a separate fee payer sponsors the network fee. `unfreeze` finalizes the checkpoint once, accumulates the frozen seconds of the episode, and sets the vault's and operator's recourse deadline to `now + freeze_grace_seconds + 72 h` (see §12.7).

9.5 Token accounts and authorities

USDC on Solana is a classic SPL Token mint, owned by `TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA` [32]. Token-2022 transfer hooks are therefore not available for it, and every rule about who may move a vault's USDC has to live in the program that is the token account's authority [32]. Bonded is that program: the vault, bond and escrow token accounts all have a Bonded PDA as their authority, and the program types every token account and token program as classic SPL (`Account<TokenAccount>`, `Program<Token>`). A Token-2022 account or program fails account validation. `initialize_protocol` accepts any classic mint with 6 decimals and stores it; every later instruction requires `mint` to equal `protocol.usdc_mint`, and every transfer is a `TransferChecked` with 6 decimals.

Token account	Address	Authority	Delegate	Created by	Closed
Vault token	ATA of the <code>Vault</code> PDA for the mint	<code>Vault</code> PDA	session key, up to the float allowance	<code>create_vault</code> (accepts a correctly precreated ATA)	never
Bond token	ATA of the <code>Bond</code> PDA	<code>Bond</code> PDA	none	<code>create_operator</code>	never
Escrow token	ATA of the <code>Escrow</code> PDA	<code>Escrow</code> PDA	none	<code>pay_escrow</code>	at the terminal transition, rent to the escrow's rent payer
Fee account	PDA <code>[b"fees"]</code> , not an ATA	<code>ProtocolConfig</code> PDA	none	<code>initialize_protocol</code>	never
Merchant token	any classic account of the mint owned by the merchant	merchant	n/a	the merchant, or an idempotent ATA instruction in the same transaction	n/a

The fee account sits at a program-derived address with the protocol PDA as authority, so no external wallet can choose its address or close it, and a waived-fee settlement that names it can never be blocked by its absence.

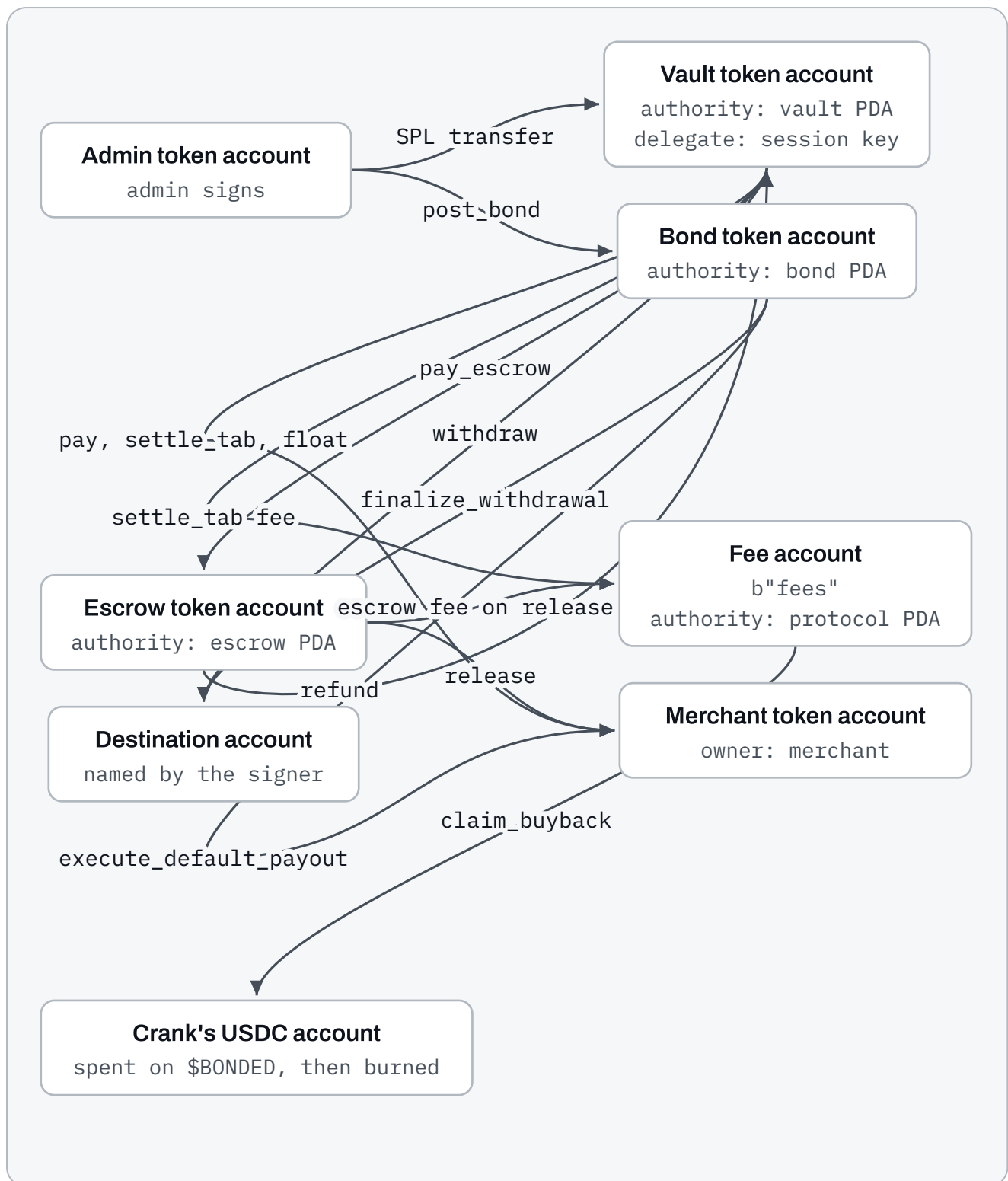


Figure 40. Where USDC moves, and the instruction that moves it.

CPI signing

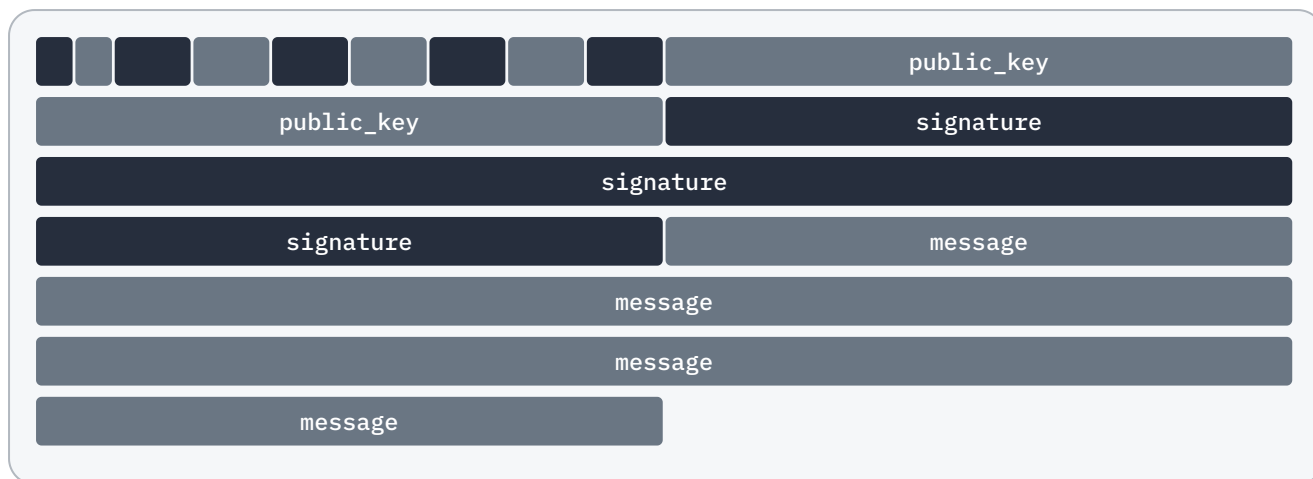
A PDA has no private key; the program signs for it by passing its seeds and bump to `invoke_signed`. Each authority signs with exactly the seeds it was derived from:

PDA	Signer seeds	Signs
Vault	b"vault" , operator, agent_id , bump	TransferChecked from the vault (pay , pay_escrow , settle_tab , withdraw); Approve (refill_float); Revoke (freeze , set_policy , rotate_session_key , withdraw)
Bond	b"bond" , operator, bump	TransferChecked from the bond (execute_default_payout , finalize_withdrawal)
Escrow	b"escrow" , vault, quote_hash , bump	TransferChecked and CloseAccount on the escrow token account
ProtocolConfig	b"protocol" , bump	TransferChecked from the fee account to the crank (claim_buyback)
Freeze reserve	b"freeze-reserve" , vault, bump	system transfer of checkpoint rent (freeze)
FreezeCheckpoint	b"freeze" , vault, generation LE, bump	system allocate and assign of itself (freeze)

No CPI calls back into Bonded. Almost every one is a single call into SPL Token or the System program; creating an associated token account adds the Associated Token program's own calls beneath it, well within the runtime's 5-frame limit [29]. The float lane is the one place where Bonded does not sign the transfer. The x402 exact scheme requires a top-level TransferChecked [7], so refill_float approves the session key as SPL delegate of the vault token account for at most the float allowance, and the session key signs its own top-level transfers inside that allowance. The program bounds the float through the allowance it approves and reconciles what was spent from the token account's delegated_amount .

Precompile proofs

The program never verifies a signature in its own compute. Each signed message travels in a native precompile instruction placed at the index immediately before the Bonded instruction, and the handler reads it through the instructions sysvar. It requires the current instruction to be Bonded's own top-level instruction (a CPI wrapper fails), the proof to be at `current - 1` , the precompile program to be Ed25519 or secp256r1, the proof to carry no accounts, and its 16-byte offset header to match one canonical value byte for byte, so the signature, key and message all sit inside the proof instruction itself. Then it decodes the message and compares the signer with the key the account state expects.



208 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	1	<code>num_signatures</code>	1
1	1	<code>padding</code>	0
2	2	<code>signature_offset</code>	48, u16 LE
4	2	<code>signature_instruction_index</code>	0xFFFF, this instruction
6	2	<code>public_key_offset</code>	16
8	2	<code>public_key_instruction_index</code>	0xFFFF
10	2	<code>message_data_offset</code>	112
12	2	<code>message_data_size</code>	96
14	2	<code>message_instruction_index</code>	0xFFFF
16	32	<code>public_key</code>	the line's voucher session key
48	64	<code>signature</code>	Ed25519 over the message
112	96	<code>message</code>	the voucher, see §5.2

Figure 41. The 208-byte Ed25519 proof instruction that precedes `settle_tab`, `file_claim` and `attest_line_voucher`.

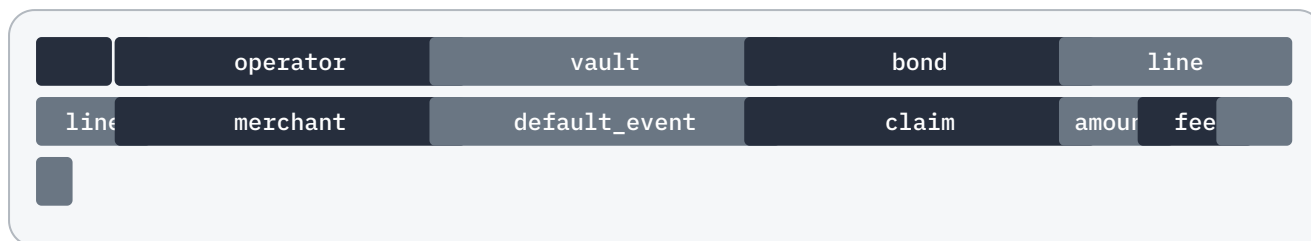
Proof	Precompile	Message bytes	Instruction data bytes	Read by
Voucher	Ed25519	96	208	<code>settle_tab</code> , <code>file_claim</code> , <code>attest_line_voucher</code>
Merchant quote	Ed25519	169	281	<code>pay</code> with a quote, <code>pay_escrow</code>
Approval v2	Ed25519	128	240	<code>approve_payment</code>
Passkey assertion	secp256r1	<code>authenticatorData</code> SHA-256(<code>clientDataJSON</code>)	113 + message	<code>approve_payment_passkey</code>

The Ed25519 header for quotes and approvals differs from the voucher header only in the message size field. The secp256r1 header places the 33-byte compressed key at offset 16, the 64-byte signature at 49 and the message at 113.

9.6 Events

The program writes its history as Anchor events: each `emit!` logs `Program data:` followed by the base64 of the event discriminator and its Borsh fields. Events carry what closed accounts no longer can. A claim closes at payout, an approval on consumption, an escrow at its terminal transition; their events keep the full record for an indexer that starts later.

Event	Emitted by	Fields	Bytes
CoreEvent	24 instructions, in 26 kinds (below)	kind, seven keys, amount, fee, timestamp	257
ClaimFiled	<code>file_claim</code> (every first or replacement filing)	claim, event, line, merchant; voucher amount, nonce, expiry; settled and limit snapshots; eligible; filed at	192
PaymentCommitted	<code>pay</code> , <code>pay_escrow</code>	receipt, vault, merchant, intent and quote hashes, nonce, amount, total debit, fee, approval, timestamp, lane	241
ApprovalCreated	<code>approve_payment</code> , <code>approve_payment_passkey</code>	approval, vault, intent, merchant, amount, expiry, approving key or credential, rent payer, created at, policy revision	232
ApprovalClosed	<code>close_approval</code>	approval, vault, timestamp	80
EscrowEvent	every escrow instruction	the full <code>Escrow</code> snapshot, updated at, actor; the 169-byte quote and 64-byte merchant signature on funding only	704
ProtocolUpdated	<code>update_protocol</code> , <code>configure_escrow_fee</code> , <code>configure_escrow_resolution</code>	protocol, tab fee, waiver, timestamp	51
GuardianRotated	<code>rotate_operator_guardian</code> , <code>rotate_vault_guardian</code>	operator, vault (zero for the operator level), guardian, applied, effective at, timestamp	121
BuybackConfigured	<code>configure_buyback</code>	token mint, crank, minimum USDC, timestamp	88
BuybackBurned	<code>burn_buyback</code>	USDC in, tokens burned, token mint, crank, timestamp	96



257 bytes · one cell per 4 bytes

Offset	Bytes	Field	Meaning
0	8	discriminator	SHA-256("event:CoreEvent")[0..8]
8	1	kind	1 to 26
9	32	operator	zero in kind 26
41	32	vault	
73	32	bond	
105	32	line	
137	32	merchant	
169	32	default_event	
201	32	claim	
233	8	amount	u64, meaning set by kind
241	8	fee	u64, tab fee in kind 8
249	8	timestamp	i64 Unix seconds

Figure 42. The 257-byte CoreEvent, with zero keys where a reference does not apply.

Kind	Activity	Instruction	amount
1	operator created	create_operator	0
2	vault created	create_vault	0
3	policy updated	set_policy	0
4	bond posted	post_bond	deposit
5	line opened	open_line	limit
6	line decrease requested	resize_line	requested limit
7	line limit applied	resize_line	new limit
8	tab settled	settle_tab	delta (fee in fee)
9	default opened	settle_tab	delta that found the vault short
10	claim filed	file_claim	eligible amount
11	claim paid	execute_default_payout	payout

Kind	Activity	Instruction	amount
12	default paid	<code>execute_default_payout</code> , <code>finalize_empty_default</code>	total paid
13	withdrawal requested	<code>request_withdrawal</code>	amount
14	bond withdrawn	<code>finalize_withdrawal</code>	amount
15	vault frozen	<code>freeze</code>	0
16	vault unfrozen	<code>unfreeze</code>	0
17	line renewed	<code>renew_line</code>	limit
18	line key rotation requested	<code>rotate_line_key</code>	0
19	line key rotated	<code>rotate_line_key</code>	0
20	float refilled	<code>refill_float</code>	new allowance
21	merchant rule updated	<code>set_merchant_rule</code>	per-transaction cap
22	approval created	<code>approve_payment</code> , <code>approve_payment_passkey</code>	amount
23	payment	<code>pay</code>	total debit
24	session key rotated	<code>rotate_session_key</code>	0
25	vault withdrawn	<code>withdraw</code>	amount
26	line voucher attested	<code>attest_line_voucher</code>	acknowledged cumulative

How off-chain readers use them

The indexer reads successful transactions at `confirmed` commitment that touch the program and decodes only `Program data`: lines logged while Bonded is the active invocation, tracking the invoke stack so a log from a token CPI or another program can never pass as a Bonded event. A failed transaction produces no activity. Each event is identified by transaction signature, top-level instruction index and event position, which makes reprocessing idempotent. Detail events are joined to the `CoreEvent` of the same instruction (`ClaimFiled` to kind 10, `ApprovalCreated` to kind 22, `PaymentCommitted` to kind 23 or to the funding `EscrowEvent`), and the join checks that their account bindings agree. Alongside the log stream, the indexer snapshots the program's accounts at a known slot and reconciles its projections against them, so the chain state stays authoritative and the event history explains how it got there (see §11.2).

9.7 Compute and size budgets

Each figure below is a complete signed legacy transaction executed in LiteSVM against the compiled program, with signature verification, classic SPL Token and the native Ed25519 and P-256 precompiles enabled. "Bare" means only the instructions the action needs; "SDK ATA guard" adds an idempotent create-ATA instruction for the merchant's destination; "API memo" adds the SPL Memo that carries the API's preparation ID.

Signed legacy transaction	Bytes	Compute units	Signatures	Account keys	Fits 1,232 B
<code>settle_tab</code> , one voucher; bare; placeholder closes	894	61,044	1	17	yes
<code>settle_tab</code> , one voucher; SDK ATA guard	936	65,382	1	18	yes

Signed legacy transaction	Bytes	Compute units	Signatures	Account keys	Fits 1,232 B
<code>settle_tab</code> , one voucher; API ATA guard + memo	1,014	82,363	1	19	yes
<code>settle_tab</code> , two merchants; bare	1,302	122,088	2	20	no
<code>settle_tab</code> , two merchants; SDK ATA guards	1,354	132,264	2	21	no
<code>settle_tab</code> , one merchant, two operators; bare	1,366	114,588	1	24	no
<code>settle_tab</code> , archived old-key voucher, funded tail	927	68,692	1	18	yes
<code>pay_escrow</code> , no approval; bare	1,033	79,500	1	17	yes
<code>pay_escrow</code> , no approval; API memo	1,111	96,481	1	18	yes
<code>pay_escrow</code> , scoped approval consumed; bare	1,097	82,053	1	19	yes
<code>pay_escrow</code> , scoped approval consumed; API memo	1,175	99,034	1	20	yes
<code>approve_payment_passkey</code> , WebAuthn P-256; bare	897	32,701	1	10	yes
<code>approve_payment_passkey</code> , WebAuthn P-256; API memo	975	49,682	1	11	yes
<code>freeze</code> , reserve-funded checkpoint	409	54,390	1	9	yes
<code>unfreeze</code> , accumulating frozen seconds	376	28,056	1	8	yes
<code>freeze</code> again in the same episode	409	33,387	1	9	yes
<code>attest_line_voucher</code> during notice	589	17,652	1	8	yes

Every measured transaction stays far below the 1.4M compute-unit ceiling [27]; the largest single-action transaction uses under 100,000. The binding constraint is size. A settlement uses 15 account metas and 18 bytes of instruction data (8-byte discriminator, `u64` sequence, `u16` proof index) next to its 208-byte proof; escrow funding uses 82 bytes of data next to a 281-byte quote proof. One settlement fits a legacy packet with room for an ATA guard and a memo; two do not, so the settlement planner builds exactly one line per transaction and rejects any fully serialized transaction over 1,232 bytes. Only the latest cumulative voucher of a line ever needs settling, so one transaction per line per batch is the whole cost (see §5.7). The passkey approval runs in its own transaction for the same reason: the instruction carries `authenticatorData` (37–128 bytes) and `clientDataJSON` (up to 512 bytes), and its `secp256r1` proof carries the authenticator data again, so the measured approval with 37 bytes of authenticator data already takes 897 bytes.

These values are measured on LiteSVM 0.7.1 with the Agave 2.3 runtime and Anchor 0.31.1 output, for the build whose SHA-256 is `42bed621895ab4ab58686c7f986cb6b3aecd8919cf88d081d3363504c787d235` , with reproducible test keys. Compute depends slightly on the keys through PDA bump searches, so the table describes this benchmark, not a protocol constant.

9.8 Upgrade authority and program versioning

The upgrade authority governs

The program is deployed under the upgradeable loader (loader-v3), and its upgrade authority is the only governance key. The five governed instructions (`initialize_protocol` , `update_protocol` , `configure_escrow_fee` , `configure_escrow_resolution` , `configure_buyback`) each take the program account and its ProgramData account and require two facts at execution: the program account points to that ProgramData, and its `upgrade_authority_address` is the signer. Each update then writes the signer into `ProtocolConfig.authority` . The stored field is a record, not a gate. When the loader's upgrade authority moves, to a Squads multisig for example, protocol governance moves with it, with no program change and no stale key left in control (see §12.9 for custody and verified builds).

The same check fixes what governance can reach. It can change the tab fee and waiver within the immutable maximum, set the escrow fee within its maximum, set the resolution duration and name the buyback's mint, crank and minimum. It cannot move collected fees: they leave only through the crank's buyback, and that burns what it buys. It cannot change the mint, the fee account, the withdrawal margin, the freeze grace, the cluster genesis hash or either maximum fee once set, and it cannot touch any operator's vault, bond or line: those instructions require the operator's admin. If the program is made immutable, the loader records no upgrade authority, no signer can satisfy the constraint, and the protocol configuration is fixed at its last values. A timelock on upgrades is a property of the key that holds the authority, such as a Squads multisig with a 7-day timelock (§12.9); the program accepts whatever the loader records.

The deployment script deploys with the deployer as upgrade authority, compares the SHA-256 of the deployed ProgramData bytes with the local build, then initializes the protocol and the escrow configuration with that same key and verifies the fee account. Run again against an initialized protocol, it verifies every stored value and sends nothing.

Versions

The crate and its IDL are version 0.1.0. Versioning that matters to integrators lives in the wire formats and the layouts:

Format	Domain prefix	Size	Note
Voucher	"Bonded voucher v2\0"	96 B	The v1 literal tag <code>BONDTAB1</code> is rejected
Merchant quote	"Bonded quote v1\0"	169 B	Lane byte: 0 for <code>pay</code> , 1 for <code>pay_escrow</code>
Payment intent	"Bonded payment intent v1\0"	32 B hash	Hash of tag, vault, merchant, total debit, nonce, quote hash, lane
Approval	"Bonded approval v2\0"	128 B	Binds the policy revision; 120-byte v1 approvals are rejected

Each tag is the first 8 bytes of SHA-256 over the prefix, the program ID and the cluster genesis hash. A format change takes a new prefix, so an old signature can never be read as a new message. Account layouts carry no version byte: an account type is identified by its discriminator alone, and layout revisions append fields at the end of a struct, which is how the freeze-episode fields joined `Vault` and `FreezeCheckpoint` and the key-epoch fields joined `CreditLine` . A redeployment to a new program ID changes every PDA and every signature domain at once, so state and signatures from one deployment never carry over to another.

10. Fees and settlement

Bonded charges two protocol fees: a tab settlement fee on what a merchant settles from a vault, and an escrow fee on jobs that release to the merchant. Fees are paid in USDC, and every fee the protocol keeps is for buying the protocol's token, \$BONDED, on the market and burning it (§10.7). Every other cost a participant bears is Solana's own: the signature fee on each transaction and the rent deposit on each account it creates. This chapter gives each charge, the formula the program applies, who pays it, when it is taken and where it goes, and ends with what it costs to settle a 100 USDC tab.

10.1 What the protocol charges

Charge	Paid by	Configured rate	Computed on	Taken	Goes to
Tab settlement fee	merchant, out of its proceeds	10 bps, waived during beta	<code>delta</code> of each <code>settle_tab</code>	at settlement	fee treasury
Escrow fee	payer, on top of the principal	25 bps	quote principal	reserved at funding, earned at release	fee treasury on release, back to the vault on refund
Signature fee	transaction fee payer	5,000 lamports per signature [27]	every signature, precompile signatures included	per transaction	the Solana network
Rent deposit	creator of each account	$(\text{size} + 128) \times 3,480 \times 2$ lamports [28]	account size	when the account is created	held in the account, returned when it closes

Nothing else carries a protocol fee. Float payments are plain SPL transfers the program does not touch. Program `pay` transfers, bond deposits and withdrawals, vault withdrawals and default payouts move USDC without any deduction and record a fee of zero in their `CoreEvent` or `PaymentCommitted` events. An escrow refund returns the reserved fee to the vault with the principal. A default payout in particular reaches the merchant whole: `execute_default_payout` transfers the computed payout and nothing goes to the treasury.

The rates in the table are configuration, not program constants. `initialize_protocol` takes the tab fee, the waiver and a maximum; `configure_escrow_fee` takes the escrow fee and its maximum. The values above are those in the deployment parameter files for devnet and mainnet-beta, and the local development stack initializes with the same fee values; §14 lists every initialization value.

10.2 The tab settlement fee

A merchant settles a tab by presenting its latest cumulative voucher. The vault pays the voucher's `delta`, the amount it adds over what was settled before, and the fee is deducted from the merchant's side of that transfer:

$$\text{fee} = \lfloor \text{delta} \times \text{line.tab_fee_bps} \div 10\,000 \rfloor$$

$$\text{merchant receives} = \text{delta} - \text{fee}$$

$$\text{fee treasury receives} = \text{fee}$$

$$\text{vault pays} = \text{delta}$$

<code>delta</code>	<code>voucher.cumulative - max(settled_cumulative, retired_cumulative)</code> , USDC base units
<code>line.tab_fee_bps</code>	the rate stored in the credit line when it was opened

The agent's debt does not depend on the fee: the vault pays exactly what the session key vouched for, and the merchant receives that amount less the fee. The multiplication runs in u128 and the division floors, so the fee rounds down, in the merchant's favour, and never exceeds the rate. The program skips the second transfer when the fee is zero.

Because a voucher is cumulative, one settlement covers every request since the last one, and the fee is computed once on the whole delta. Settling 400 requests of 0.25 USDC in one transaction rounds once, not 400 times. At 10 bps a delta below 1,000 base units carries no fee. A delta of 100 USDC, 100,000,000 base units, carries 100,000 base units, 0.10 USDC.

The rate is fixed per line

The rate is not read from the protocol at settlement. `open_line` copies it into the new `CreditLine`, and `settle_tab` uses the line's copy:

$$\begin{aligned} \text{line.tab_fee_bps} &= 0 \text{ if protocol.tab_fee_waived at open_line} \\ \text{line.tab_fee_bps} &= \text{protocol.tab_fee_bps otherwise} \end{aligned}$$

The stored rate is immutable for the life of the line's PDA, and `renew_line` keeps it. A merchant therefore knows, before it serves the first request on a line, the exact fee every settlement on that line will pay, and no governance action can reprice vouchers it has already accepted. Changes made with `update_protocol` apply to lines opened afterwards.

How the beta waiver works on chain

The waiver is the `tab_fee_waived` flag in `ProtocolConfig`. While it is set, `open_line` writes a rate of zero into every line it opens, whatever `tab_fee_bps` says, and those lines settle fee-free for as long as they exist. The protocol keeps `tab_fee_bps` at its configured 10 bps throughout, so ending the waiver is a single `update_protocol` call that clears the flag; lines opened from then on store 10 bps. Lines opened during the waiver keep their zero rate, including after renewal, because a line's rate never changes. A bond has one line per merchant, so a merchant whose line was opened during beta settles that line without a fee.

Even with every fee waived, `settle_tab` requires the fee treasury account, at its stored address, in its account list. The treasury is a program-derived token account whose authority is the `ProtocolConfig` PDA, and no external wallet can close it, so its absence can never block a fee-free settlement.

10.3 The escrow fee

The escrow fee is charged on the quote's principal and added to the debit, so the merchant receives the quoted amount in full:

$$\text{fee_amount} = \lfloor \text{amount} \times \text{escrow_fee_bps} \div 10\,000 \rfloor$$

$$\text{total_debit} = \text{amount} + \text{fee_amount}$$

`pay_escrow` reserves `total_debit` from the vault into the escrow token account and snapshots the rate and the fee into the `Escrow`. The vault's caps, budgets and co-sign threshold all measure `total_debit`. On release the fee goes to the fee treasury; on a timeout refund, an arbiter's refund or a refund after the resolution deadline it returns to the vault with the principal. The fee rounds down, in the payer's favour, and a principal below 400 base units pays none at 25 bps. A 340 USDC job carries 850,000 base units of fee and debits 340,850,000. §8.7 gives the full sequence.

10.4 Ceilings, the fee treasury and governance

Each fee rate has a ceiling the program enforces, and the ceiling cannot be raised after it is set.

- **Tab fee ceiling.** `maximum_tab_fee_bps` is an argument of `initialize_protocol`, at most 10,000, and no instruction changes it afterwards. `initialize_protocol` and every `update_protocol` require `tab_fee_bps ≤ maximum_tab_fee_bps`. The parameter files set the ceiling at 100 bps.
- **Escrow fee ceiling.** `maximum_escrow_fee_bps` starts unset. The first `configure_escrow_fee` fixes it, at most 10,000; every later call must pass the same maximum, and the fee must not exceed it. `pay_escrow` checks the fee against the ceiling again before it funds. The parameter files set 25 bps under a 100 bps ceiling.

Governance over fees belongs to the program's upgrade authority, checked against the chain itself. Every instruction in the table below requires the executable `bonded` program account to point at the supplied loader-v3 `ProgramData` account, and the signer to be that account's `upgrade_authority_address`. There is no separate fee key and no first-caller bootstrap. The updating instructions also write the signer into `ProtocolConfig.authority`, so a rotation of the upgrade authority through the loader carries over to fee governance.

Instruction	Arguments	Effect	Constraint
<code>initialize_protocol</code>	<code>tab_fee_bps</code> , <code>tab_fee_waived</code> , <code>withdrawal_margin_seconds</code> , <code>maximum_tab_fee_bps</code> , <code>freeze_grace_seconds</code> , <code>cluster_genesis_hash</code>	creates <code>ProtocolConfig</code> and the fee treasury	<code>ceiling ≤ 10,000</code> , <code>fee ≤ ceiling</code> , <code>margin > 0</code> , <code>grace ≥ 345,600 s</code> , six-decimal mint

Instruction	Arguments	Effect	Constraint
<code>update_protocol</code>	<code>tab_fee_bps</code> , <code>tab_fee_waived</code>	changes the rate and waiver for future lines; emits <code>ProtocolUpdated</code>	$\text{fee} \leq \text{the immutable ceiling}$
<code>configure_escrow_fee</code>	<code>fee_bps</code> , <code>maximum_fee_bps</code>	sets the escrow fee; fixes the ceiling on first use	$\text{ceiling} \leq 10,000$ and unchanged, $\text{fee} \leq \text{ceiling}$
<code>configure_escrow_resolution</code>	<code>resolve_seconds</code>	sets the dispute resolution duration	$\text{resolve_seconds} > 0$
<code>configure_buyback</code>	<code>minimum_usdc</code> , <code>crank</code> ; the \$BONDED mint as an account	sets the token the fees buy, the one key allowed to run buybacks, and the smallest claim; emits <code>BuybackConfigured</code>	$\text{minimum_usdc} > 0$, a crank key, no buyback pending; the mint is SPL Token or Token-2022 with only metadata extensions

The fee treasury is the token account at the PDA `[b"fees"]` , for the configured USDC mint, with the `ProtocolConfig PDA ( [b"protocol"] )` as its authority. `initialize_protocol` creates it and stores its address in `ProtocolConfig.fee_treasury` ; `settle_tab` and the escrow release path accept only that address. No instruction sends its USDC to an address someone names. It leaves only through the buyback in §10.7: `claim_buyback` , a swap, and `burn_buyback` , in one transaction signed by the configured crank.

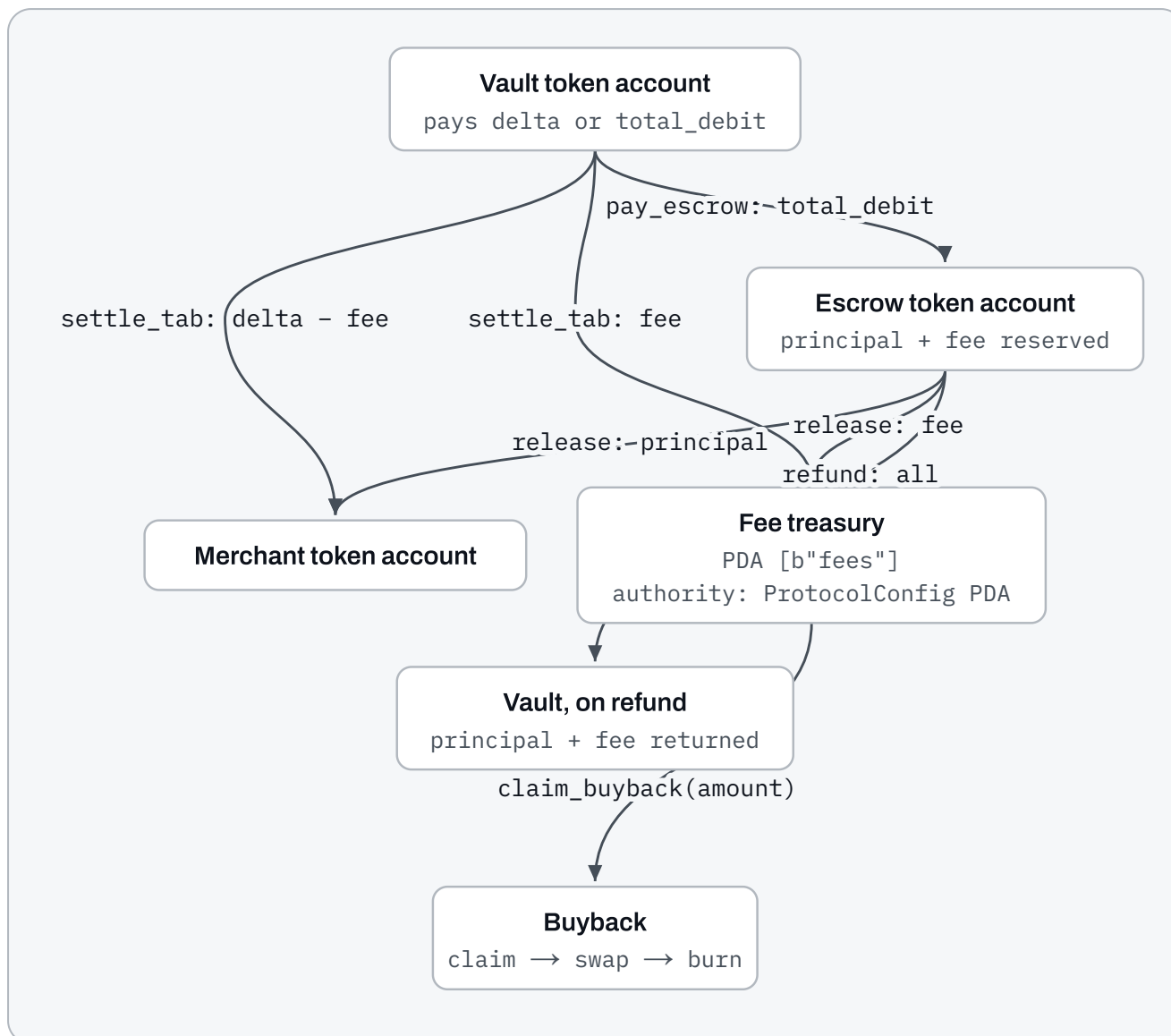


Figure 43. Where protocol fees go: settlement and escrow release pay into the fee treasury, and only the buyback takes USDC out, to burn \$BONDED.

10.5 Solana costs

Signature fees

Solana charges a base fee of 5,000 lamports per signature [27]. The count includes the transaction's own signatures and the signatures that precompile instructions verify, so a Bonded transaction with one signer and one Ed25519 or secp256r1 proof pays for two. Every transaction the API prepares also starts with two compute-budget instructions: a unit limit sized at $1.2 \times$ the simulated units plus 1,000, and a unit price of 1,000 micro-lamports by default (`TRANSACTION_COMPUTE_UNIT_PRICE_MICROLAMPORTS`). The priority fee is that price times the limit, about 100 lamports for a settlement, on top of the base fee. The measured compute of a settlement, 61,044 compute units for one voucher, does not change the base fee.

Transaction	Signer and fee payer	Transaction signatures	Precompile signatures	Base fee (lamports)
<code>settle_tab</code> , one voucher	merchant	1	1 Ed25519	10,000
<code>file_claim</code>	merchant	1	1 Ed25519	10,000

Transaction	Signer and fee payer	Transaction signatures	Precompile signatures	Base fee (lamports)
<code>execute_default_payout</code> , one claim	keeper or any key	1	0	5,000
<code>pay_escrow</code>	session key	1	1 Ed25519	10,000
<code>approve_payment</code> / <code>approve_payment_passkey</code>	any payer	1	1 Ed25519 / secp256r1	10,000
<code>mark_delivered</code> , <code>accept</code> , <code>dispute</code> , <code>resolve</code> , <code>refund_on_timeout</code> , <code>release_after_window</code> , <code>refund_after_resolution_deadline</code>	the instruction's signer	1	0	5,000
<code>refill_float</code>	any key	1	0	5,000

Size and compute of settlement transactions

The program was measured on a local bank executing the compiled SBF binary with signature verification, with every transaction fully signed and serialized. Sizes include the signature section; compute is the metered total for the transaction.

Signed legacy transaction	Bytes	Compute units	Fits 1,232 bytes
<code>settle_tab</code> , one voucher	894	61,044	yes
<code>settle_tab</code> , one voucher, with the SDK's ATA guard	936	65,382	yes
<code>settle_tab</code> , one voucher, API-prepared with memo	1,014	82,363	yes
<code>settle_tab</code> , two merchants	1,302	122,088	no
<code>pay_escrow</code> , no approval	1,033	79,500	yes
<code>pay_escrow</code> , consuming an approval	1,097	82,053	yes
<code>approve_payment_passkey</code> (WebAuthn P-256)	897	32,701	yes

A legacy transaction settles one line. Two merchants in one transaction exceed the 1,232-byte limit, so the shared planner sends one line per transaction and checks the serialized size before signing (§5.7, §5.8). A merchant's settlement cost therefore grows with the number of lines it settles and how often it settles them, never with the number of requests behind each voucher. Every row stays far below the 1,400,000 compute-unit transaction ceiling [27].

Rent

Every account a participant creates holds a rent-exempt deposit. The standard rate is 3,480 lamports per byte-year, with exemption at two years' worth, charged on the account's data plus 128 bytes of storage overhead [28]:

$$\text{rent_exempt_minimum} = (\text{data_size} + 128) \times 3\,480 \times 2 = (\text{data_size} + 128) \times 6\,960 \text{ lamports}$$

$$\text{data_size} = 8 + \sum \text{field sizes}$$

8 the Anchor account discriminator

field sizes Borsh sizes from programs/bonded/src/state.rs; Pubkey 32, u64 and i64 8, u32 4, u16 2, u8 and bool 1, Option<T> 1 + size of T

layout.md points to state.rs for the authoritative field order and allocated sizes, which every account reserves through InitSpace with no padding and with each Option reserving its full payload. The sizes below are computed that way and match the field types in the generated IDL. A classic SPL token account is 165 bytes.

Account	Data size (bytes)	Rent (lamports)	Created by, paid by	Returned
ProtocolConfig	177	2,122,800	initialize_protocol , upgrade authority	persistent
Fee treasury token account	165	2,039,280	initialize_protocol , upgrade authority	persistent
Operator	150	1,934,880	create_operator , admin	persistent
Bond	141	1,872,240	create_operator , admin	persistent
Bond token account	165	2,039,280	create_operator , admin	persistent
Vault	351	3,333,840	create_vault , admin	persistent
Vault token account	165	2,039,280	create_vault , admin	persistent
MerchantRule	106	1,628,640	set_merchant_rule , admin	persistent
CreditLine	302	2,992,800	open_line , admin	persistent
PasskeyCredent ial	174	2,101,920	set_passkey_credential , admin	persistent
FreezeCheckpoi nt	89	1,510,320	freeze , the vault's freeze reserve or the freezing signer	to the freeze reserve via close_freeze_checkpoint
Approval	162	2,018,400	approve_payment , its payer	to that payer on use, expiry or policy change
PaymentReceip t	211	2,359,440	pay or pay_escrow , payer or session key	persistent
QuoteReceipt	138	1,851,360	quoted pay or pay_escrow	persistent

Account	Data size (bytes)	Rent (lamports)	Created by, paid by	Returned
Escrow	400	3,674,880	pay_escrow , session key	to the session key at the terminal outcome
Escrow token account	165	2,039,280	pay_escrow , session key	to the session key at the terminal outcome
DefaultEvent	156	1,976,640	settle_tab , merchant	same instruction when healthy; kept as the record of a real default
Claim	170	2,074,080	file_claim , merchant	to the merchant at payout
VoucherAttestation	145	1,900,080	attest_line_voucher , merchant	persistent

As a check on the arithmetic: `Escrow` has 392 bytes of fields, the discriminator brings it to 400, and $(400 + 128) \times 6,960 = 3,674,880$ lamports, 0.00367488 SOL.

An operator's setup for one agent and one merchant line, `create_operator` , `create_vault` and `open_line` , deposits $1,934,880 + 1,872,240 + 2,039,280 + 3,333,840 + 2,039,280 + 2,992,800 = 14,212,320$ lamports, about 0.0142 SOL, and each further line adds 2,992,800. The two token accounts are `init_if_needed` , so the admin pays nothing for one that was created beforehand at the correct associated address.

Three of these deposits work differently from the rest. Every `settle_tab` fronts the `DefaultEvent` deposit of 1,976,640 lamports and, on a healthy settlement, gets it back before the instruction ends, so a merchant needs that balance available but spends none of it. On the settlement that opens a default, the deposit stays in the event, which remains on chain as the public record. Each `file_claim` deposit comes back with the payout. And the builders prepend an idempotent creation of the recipient's associated token account to `settle_tab` , `pay` , escrow releases and default payouts, paid by the transaction's signer, which costs 2,039,280 lamports only when the recipient has none.

10.6 Worked example: settling a 100 USDC tab

As an example, an agent makes 400 requests to one merchant at 0.25 USDC each, 100 USDC in all. The vault is funded, the merchant already holds a USDC token account, and the comparison counts only what the 400 requests cost to pay for on chain.

On a tab. The agent signs 400 vouchers off chain; signing costs nothing on chain, and the merchant verifies each locally. The merchant then settles once with the 400th voucher, whose cumulative amount is 100,000,000 base units. The transaction carries the merchant's signature and the Ed25519 proof.

Per request through the float. Each request is paid with its own top-level `TransferChecked` signed by the session key as delegate, with the facilitator as fee payer, so each payment carries two signatures. The float's allowance is approved in amounts of at most the float cap; at a cap of 20 USDC, 100 USDC of spending needs at least five `refill_float` transactions.

Per request through program pay . Each request is a `pay` instruction with one signature, and each creates a persistent `PaymentReceipt` , the payment's single-use guard.

$$\text{tab} = 5\,000 \times (1 + 1) = 10\,000 \text{ lamports}$$

$$\text{float} = 5\,000 \times (400 \times 2 + 5) = 4\,025\,000 \text{ lamports}$$

$$\text{pay} = 5\,000 \times 400 + 400 \times 2\,359\,440 = 945\,776\,000 \text{ lamports, of which } 943\,776\,000 \text{ is rent held in receipts}$$

Method	Transactions	Signatures charged	Base fees (lamports)	Rent left in accounts (lamports)	Protocol fee (base units)
Tab, one <code>settle_tab</code>	1	2	10,000	0 (placeholder returned in the same instruction)	100,000 at 10 bps; 0 on a line opened during the waiver
Float, <code>exact</code> per request	405	805	4,025,000	0	0
Program <code>pay</code> per request	400	400	2,000,000	943,776,000	0

The figures are base fees; each API-prepared transaction adds its priority fee, about 100 lamports. The tab settles the same 100 USDC for 10,000 lamports, a little under one four-hundredth of the network fees the float lane spends, and it scales with settlements rather than requests: the merchant pays the same 10,000 lamports whether the voucher covers 40 requests or 40,000. The merchant receives 99,900,000 base units on a line that carries the 10 bps fee and 100,000,000 on a line opened during the waiver. The float lane carries no protocol fee and no rent but pays for every request, and program `pay` adds a receipt deposit per payment, which suits payments that need a receipt and quote binding rather than metered calls.

10.7 \$BONDED

Supply and launch

\$BONDED is the protocol's token. Its supply is fixed at 1,000,000,000 and all of it went to one public launch on equal terms: no presale, no team, investor, airdrop or reserved allocation, and no vesting. No instruction of the program mints it or holds it; the program touches it in one place, `burn_buyback`, which burns what a buyback bought.

What the fees buy

Both of Bonded's fees land in one place, the program-owned USDC account at `[b"fees"]` (§10.4), and the protocol keeps all of both: no share of either goes to anyone else. Every base unit in that account has one use: buying \$BONDED on the market and burning it.

$$\text{kept} = \Sigma \lfloor \text{delta} \times \text{tab_fee_bps} \div 10\,000 \rfloor + \Sigma \lfloor \text{price} \times \text{escrow_fee_bps} \div 10\,000 \rfloor$$

burned = the \$BONDED that `kept` buys on the market

delta the amount each settle_tab moves from a vault to a merchant
price the principal of each escrow job that releases to its merchant
kept the USDC in the fee account, all of it spent on \$BONDED and burned

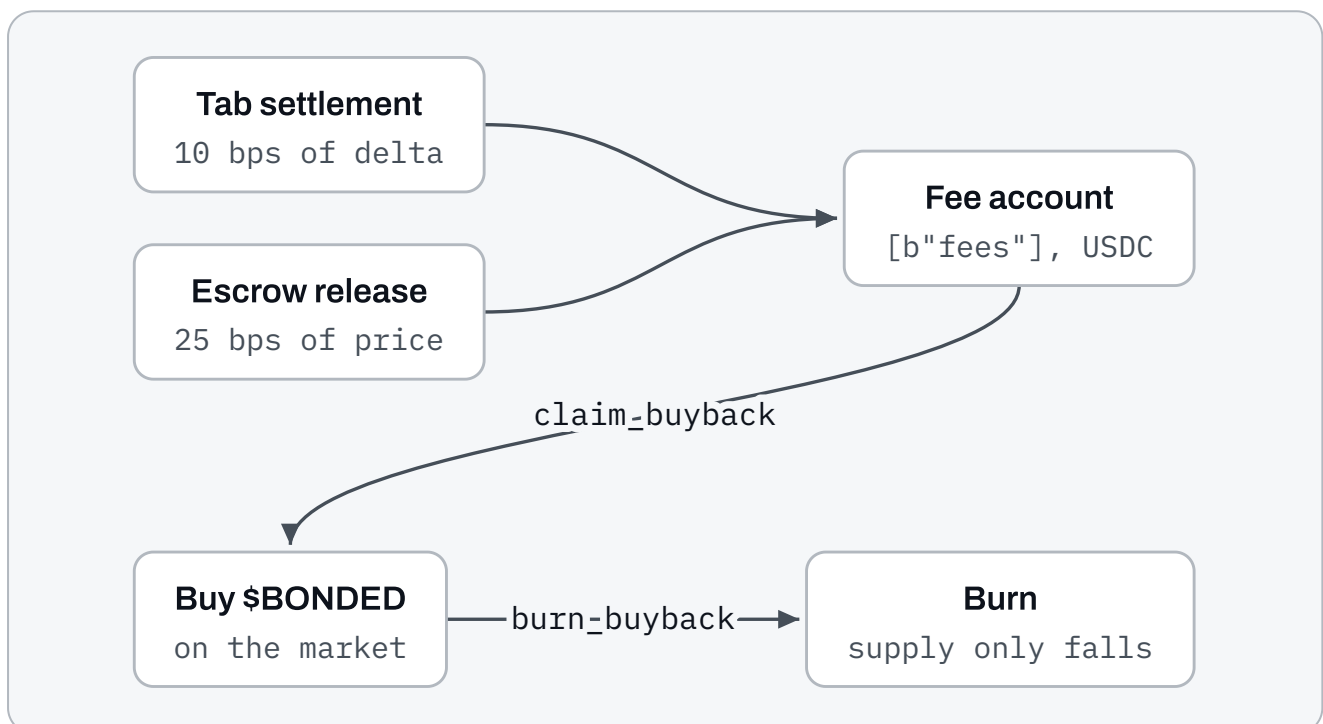


Figure 44. Where Bonded's fees go: two fees into one account, and from there into \$BONDED, bought and burned.

The burn therefore follows the money that moves through Bonded, not trading in the token itself. A 340 USDC escrow job that releases puts 0.85 USDC into the fee account; a 100 USDC settlement on a line opened after the beta waiver puts 0.10 USDC there. The fee account moves only through the buyback below.

The buyback transaction

`configure_buyback`, signed by the upgrade authority, sets three things in `ProtocolConfig`: the \$BONDED mint, the one crank key allowed to run buybacks, and the smallest claim worth running (10 USDC in the parameter files). The worker's keeper key is that crank. Every five minutes, once the fee account holds at least the minimum, it builds one versioned transaction:

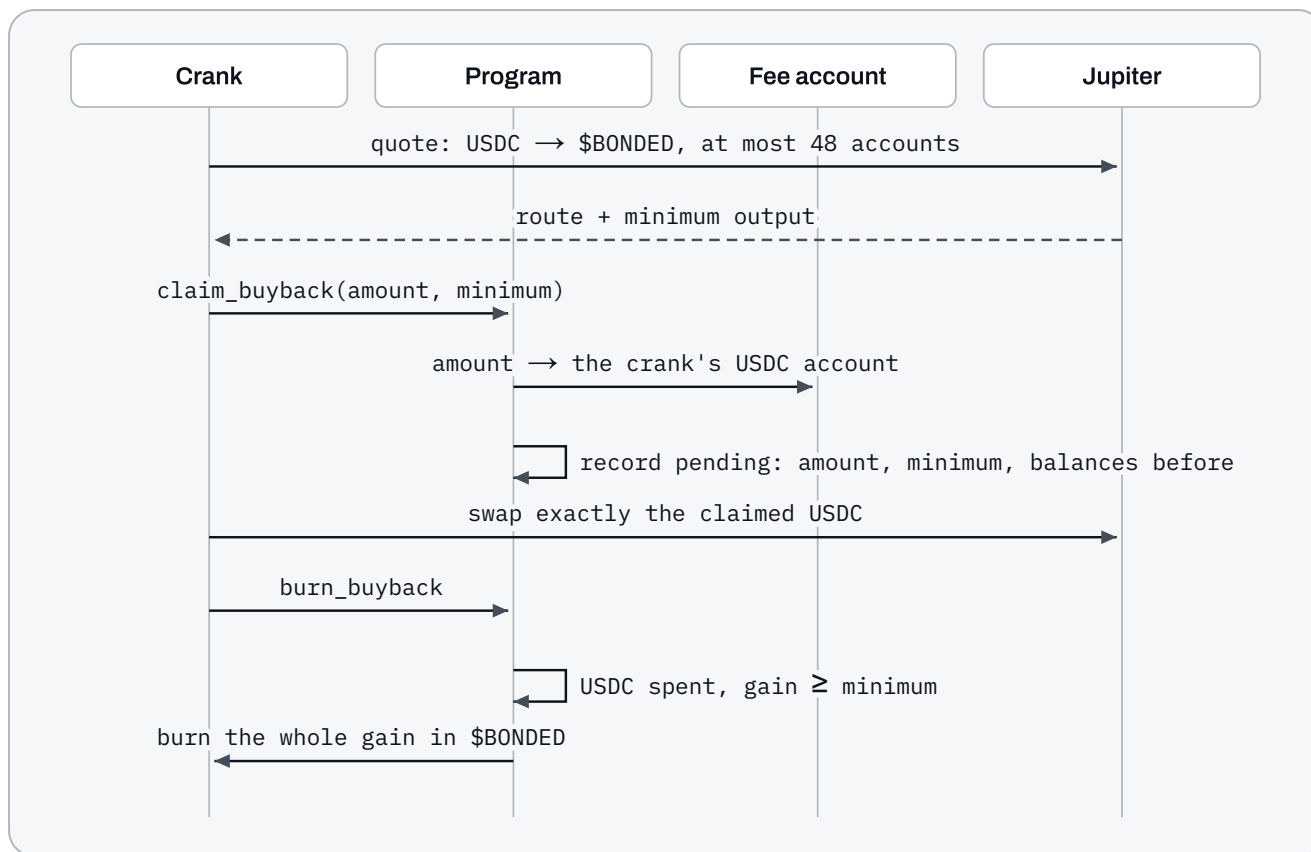


Figure 45. One buyback: the fees leave the fee account, become \$BONDED, and are burned, all in one transaction.

`claim_buyback` reads the instructions sysvar and requires a `burn_buyback` of this program later in the same transaction; `burn_buyback` requires the matching claim before it, with the same amount, minimum and accounts. Between them the crank's USDC must be spent down to what it held before the claim, and its \$BONDED balance must have grown by at least the minimum the quote promised. The burn takes the whole increase, including anything above the quote, under the mint's own token program (SPL Token or Token-2022). Any refusal fails the transaction, and the claimed USDC never left the fee account. `BuybackBurned` records the USDC spent and the \$BONDED burned.

The program refuses a claim from any key but the configured crank, a claim below the minimum or above the fee account's balance, a second claim while one is pending, and a mint other than the configured one. An open caller could route the swap through a pool of its own with a minimum of zero; the crank key closes that path, and the quote's minimum bounds the price it accepts.

Collateral stays in USDC

A bond is the operator's own USDC, and a claim is paid from it by `min(voucher - settled, line limit)`, in full or pro-rata. \$BONDED is never bond collateral and never decides a claim. A bond posted in USDC pays the same amount on the day a default opens as on the day it was posted, and a claim resolves from signatures and deadlines alone, so what a merchant is owed and what it receives never depend on the token.

11. Off-chain services

The `bonded` program holds every rule that moves money. Around it runs a small set of services that make it usable from a browser, a merchant's server and an agent runtime: an API that authenticates wallets, prepares unsigned transactions and serves indexed reads; a worker that indexes the chain and cranks permissionless deadlines; a guardian worker that watches spending and can freeze a vault; middleware that lets a merchant bill against a tab; and an agent SDK with an MCP server through which an agent signs. Each service either reads public chain state or holds one narrowly scoped key of its own. None holds an operator's admin key, an agent's session key on another party's behalf or a merchant's key, and none can move USDC that the program would not move for any other caller. Two rules run through all of them: the chain is authoritative, so Postgres rows are projections rebuilt from program accounts and logs; and the keeper, the guardian and the merchant worker judge deadlines in chain time, the block time recorded by the last committed indexer pass, never a server clock. The middleware's voucher lifetime and the SDK's quote checks use the local clock, and the program judges every deadline again on chain.

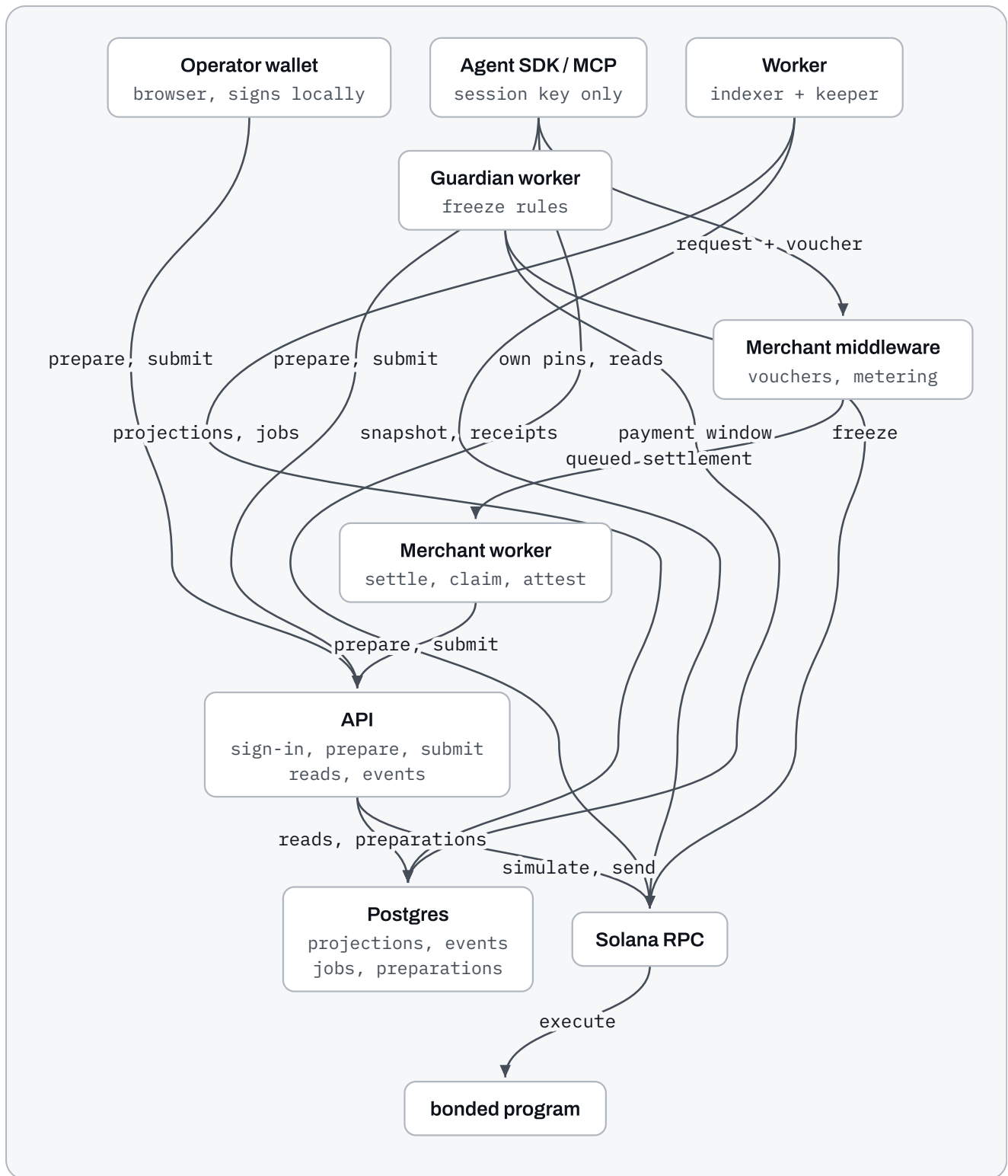


Figure 46. The off-chain services, the stores they share and the one program they all reach.

Service	Process	Reads	Signs with	Key material in the process
API	src/server.ts	Postgres projections; RPC for blockhash, simulation, broadcast	nothing	none; a session secret for cookie tags

Service	Process	Reads	Signs with	Key material in the process
Indexer	<code>src/worker.ts</code>	RPC: program accounts, signatures, receipts, token balances	nothing	none
Keeper	<code>src/worker.ts</code>	projections of escrows and defaults	its own key, for permissionless instructions only	<code>KEEPER_SECRET_KEY</code>
Guardian worker	<code>operator-run, src/guardian.ts</code>	payment projections, vault state	guardian key and fee sponsor, through an external transport	none; the transport holds the keys
Merchant middleware and worker	merchant host	line state, the merchant's metering tables	the merchant key, through the host's signer	the host's merchant signer
Agent SDK and MCP server	agent runtime	API protocol data, its own RPC	the session key: a local seed or a Turnkey, Crossmint or Coinbase transport	the session signer

11.1 The API

The API is one HTTP surface for the dashboard, merchant hosts and agents. It proves which wallet is present, decides which records that wallet may see, builds the unsigned transaction for each action, relays the signed bytes and streams committed activity. It never signs: every fee payer is the user's own wallet. In the production profile (`SERVICE_PROFILE=production`) it runs in live mode on `0.0.0.0` at the injected `PORT` , always sets Secure cookies and refuses to start without a `SESSION_SECRET` of at least 32 characters.

Wallet sign-in

A dashboard needs to know which wallet is present without asking it to sign anything that could move funds, so sign-in uses an off-chain challenge that cannot be mistaken for a transaction, a voucher or a quote.

1. `POST /v1/auth/challenge` with `{ wallet }` . The API decodes the wallet as a 32-byte Ed25519 key, draws a 32-byte random nonce (43 base64url characters) and formats an 11-line message binding the wallet, the origin, the mode, the nonce and a lifetime of `CHALLENGE_TTL_SECONDS` (300 by default, bounded to 30–600). The origin is the request's `Origin` , which must be an allowed origin, or else the first configured one.
2. The wallet signs the exact UTF-8 bytes with `signMessage` .
3. `POST /v1/auth/verify` with `{ challengeId, signatureBase64 }` . The API removes the challenge on lookup, so it works once, requires canonical base64 of exactly 64 bytes and verifies the signature against the challenged wallet.

```
Bonded off-chain authentication v1
Bonded wallet sign-in
Version: 1
Mode: live
Origin: https://app.example
Wallet: 7kD1...vQ2s
Nonce: Zx3e...Tq0A
Fixture actor: none
Issued at: 2026-10-16T12:00:00.000Z
Expires at: 2026-10-16T12:05:00.000Z
This signature creates a session and does not authorize a transaction.
```

The example shows the format with shortened values. Its first line separates it from every other byte string the same key signs: a transaction message starts with its header bytes, and vouchers, quotes and approvals start with an 8-byte domain tag (§12.3). The SDK signs nothing else as authentication:

`validateAuthenticationChallenge` re-formats the message from its parsed fields and compares it byte for byte, and checks the wallet, the configured origin, the nonce, the mode, a lifetime of at most 600 seconds and an issue time at most 30 seconds ahead of the local clock.

A verified signature creates a session with a random ID, the wallet, its identities, an expiry of `SESSION_TTL_SECONDS` (3,600 by default, bounded to 60–86,400) and a 32-byte random CSRF token returned in the body. The browser receives a separate opaque 32-byte token in `bonded_session=...; Path=/; HttpOnly; SameSite=Lax; Max-Age=...; Secure ;` with a session secret configured, the value carries an HMAC-SHA-256 tag, `<token>.<tag>`, compared in constant time before lookup. Every later request other than `GET` must carry `X-CSRF-Token`, also compared in constant time. `DELETE /v1/session` revokes the session, clears the cookie and closes its event streams.

Identities are recomputed from projections on every live request, so a role follows chain state. A wallet is `admin` of each operator it administers, `guardian` where it is an operator's or a vault's guardian, `session_key` of each vault whose policy names it, `cosigner` where it is a vault's Ed25519 co-signer (or the admin of a vault with a passkey co-signer), `merchant` once it holds a line, escrow, payment or merchant rule, `arbiter` of each escrow that names it, and `protocol_authority` when it is the current upgrade authority read from `ProgramData`. A session key rotated out on chain loses its identity on its next request.

Prepare, sign, submit

Every state change a user makes is a transaction that user's wallet signs. The API builds it because the builder knows the account order, the PDAs, the precompile layouts and the current projections; the wallet signs it because only the wallet holds the key.

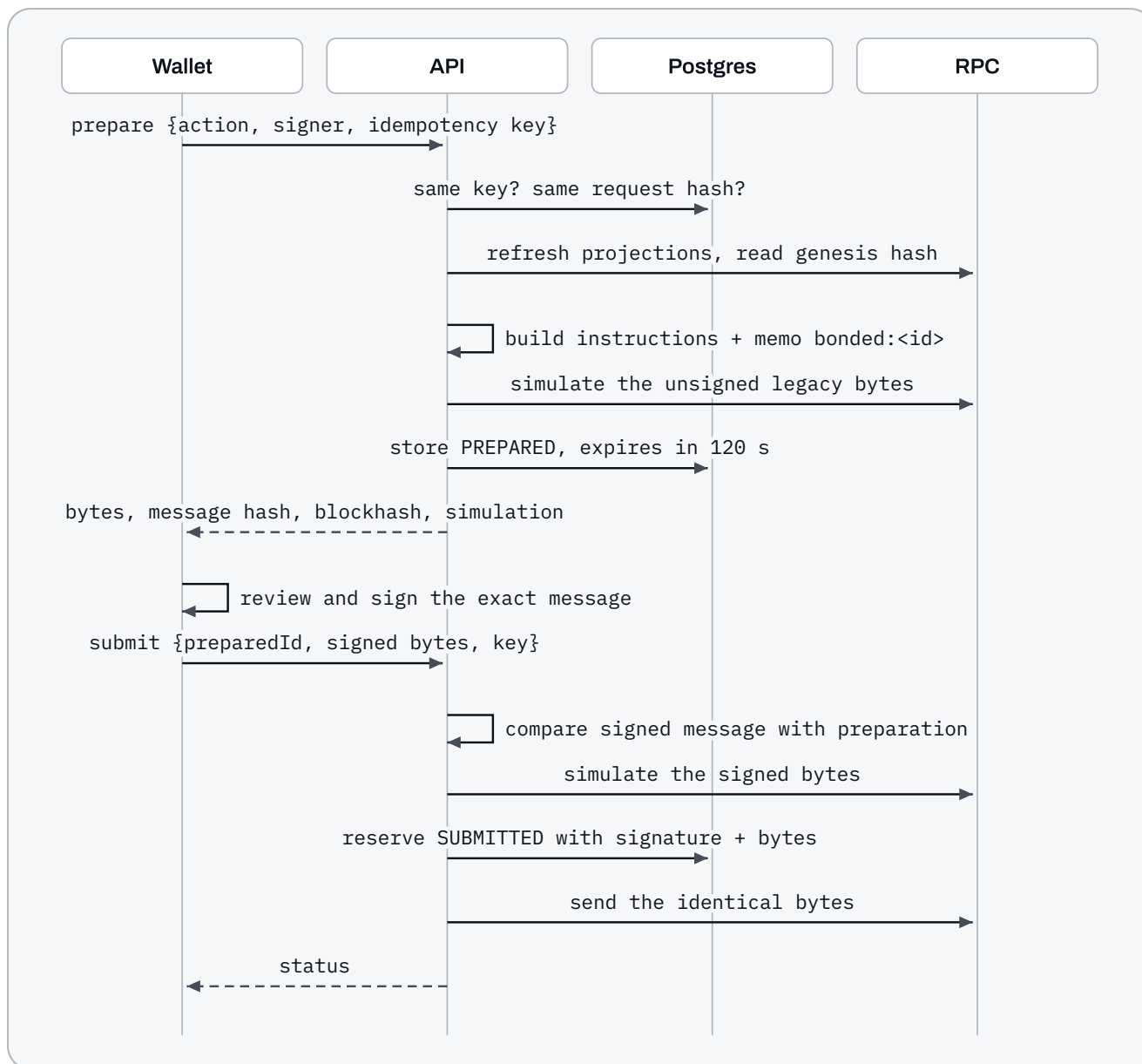


Figure 47. A wallet action from preparation to broadcast; the API never holds a signing key.

Prepare (`POST /v1/transactions/prepare` , `body { action, signerWallet, idempotencyKey }`) runs in order:

1. *Validate*. The action is one of the 41 kinds in the action catalogue, with exactly its fields. Amounts are canonical u64 decimal strings, hashes 64 lowercase hex characters, timestamps whole-second ISO UTC, keys 32 bytes, a voucher message 96 bytes and a quote message 169 bytes. `signerWallet` must be the session wallet.
2. *Deduplicate*. `(wallet, idempotencyKey)` is unique, and the request hash is SHA-256 of the request's canonical JSON. The same hash returns the stored preparation; a different hash returns `409 IDEMPOTENCY_CONFLICT` ; an expired preparation returns `409 DEADLINE_PASSED` with `details.prepareAgain: true` and asks for a new key.
3. *Refresh and check the deployment*. Projections are brought up to date (§11.2). The indexed `ProtocolConfig` must name the configured USDC mint and carry a genesis hash, a voucher domain and a positive freeze grace, and the RPC's genesis hash must equal it (in production, also `SOLANA_GENESIS_HASH`); otherwise `503 CONFIGURATION_REQUIRED` .

4. *Build.* `buildAction` checks the relationship the instruction needs (admin of the operator, session key of the vault, merchant of the line, arbiter of the escrow, upgrade authority for governance) and derives every PDA from the configured program ID. A voucher or quote is decoded, compared field by field and verified strictly before its `Ed25519` proof instruction is added. A `settle_tab` covers one line.
5. *Assemble.* A legacy transaction with the signer as fee payer and the latest blockhash opens with two `Compute Budget` instructions, a unit limit and a unit price (`TRANSACTION_COMPUTE_UNIT_PRICE_MICROLAMPORTS`, 1,000 by default), then takes the instructions and a final SPL Memo, `bonded:<preparedId>`, so that two identical intents prepared against one blockhash still have distinct messages and signatures. A quote-bound `pay_escrow` is already unique through its quote and payment receipt, so it omits the memo and keeps the room for a wallet's guards. Every signature-proof index is encoded counting the `Compute Budget` instructions ahead of it. The transaction must fit 1,232 bytes.
6. *Simulate and record.* The unsigned bytes are simulated with signature verification off, and the unit limit is then set to the units the simulation consumed, plus a fifth and 1,000 units. A program refusal returns the preparation with `simulation.allowed = false` and the mapped error, so a dashboard can show why. The `PREPARED` row stores the request hash, the response, the blockhash, its last valid block height and an expiry 120 seconds out.

The response carries `preparedId`, the action, the signer and its role, every account with its writable flag, the simulation, `expiresAt`, the unsigned `transactionBase64`, `messageHash` (SHA-256 of the message), `blockhash` and `lastValidBlockHeight`. For an allowed `settle_tab`, `file_claim` or `attest_line_voucher`, the API also keeps the merchant's voucher as private evidence for that merchant; a later preparation can raise its cumulative amount but never lower it.

Sign. The wallet signs exactly the prepared message. Because the preparation already carries its compute budget, a browser wallet keeps it and adds at most trailing guard instructions; §12.4 states exactly which additions the submit check admits. The agent SDK and the merchant worker add nothing; they rebuild the transaction from their own pinned configuration and compare it byte for byte before signing (§12.5).

Submit (`POST /v1/transactions/submit`, body `{ preparedId, signedTransactionBase64, idempotencyKey }`) checks before it relays:

1. The preparation belongs to the session wallet; the signed bytes are canonical base64, at most 1,232 bytes, preserve the prepared message (§12.4) and carry valid signatures.
2. The first submission fixes the submission key; later ones must repeat that key and byte-identical bytes, or receive `409 IDEMPOTENCY_CONFLICT`. A submitted, confirmed or failed preparation returns its status without a second broadcast.
3. A preparation past its expiry or its blockhash's last valid block height becomes `expired` and returns `409 DEADLINE_PASSED` with `details.prepareAgain = true`. A refused simulation cannot be submitted.
4. The signed bytes are simulated again; a refusal returns `409` with the mapped code.
5. The API computes the signature and, in one conditional update from `PREPARED`, stores it with the signed bytes and the key as `SUBMITTED`.
6. Only then does it broadcast. If the call fails or returns another signature and no receipt exists, it answers `503 INTERNAL_ERROR`, retryable, and asks for the identical bytes and key again.

Reserving the signature before broadcast makes a lost response harmless: if the RPC accepted the transaction, status finds the receipt by the stored signature; if not, the same signed bytes go out again without a second signature from the user.

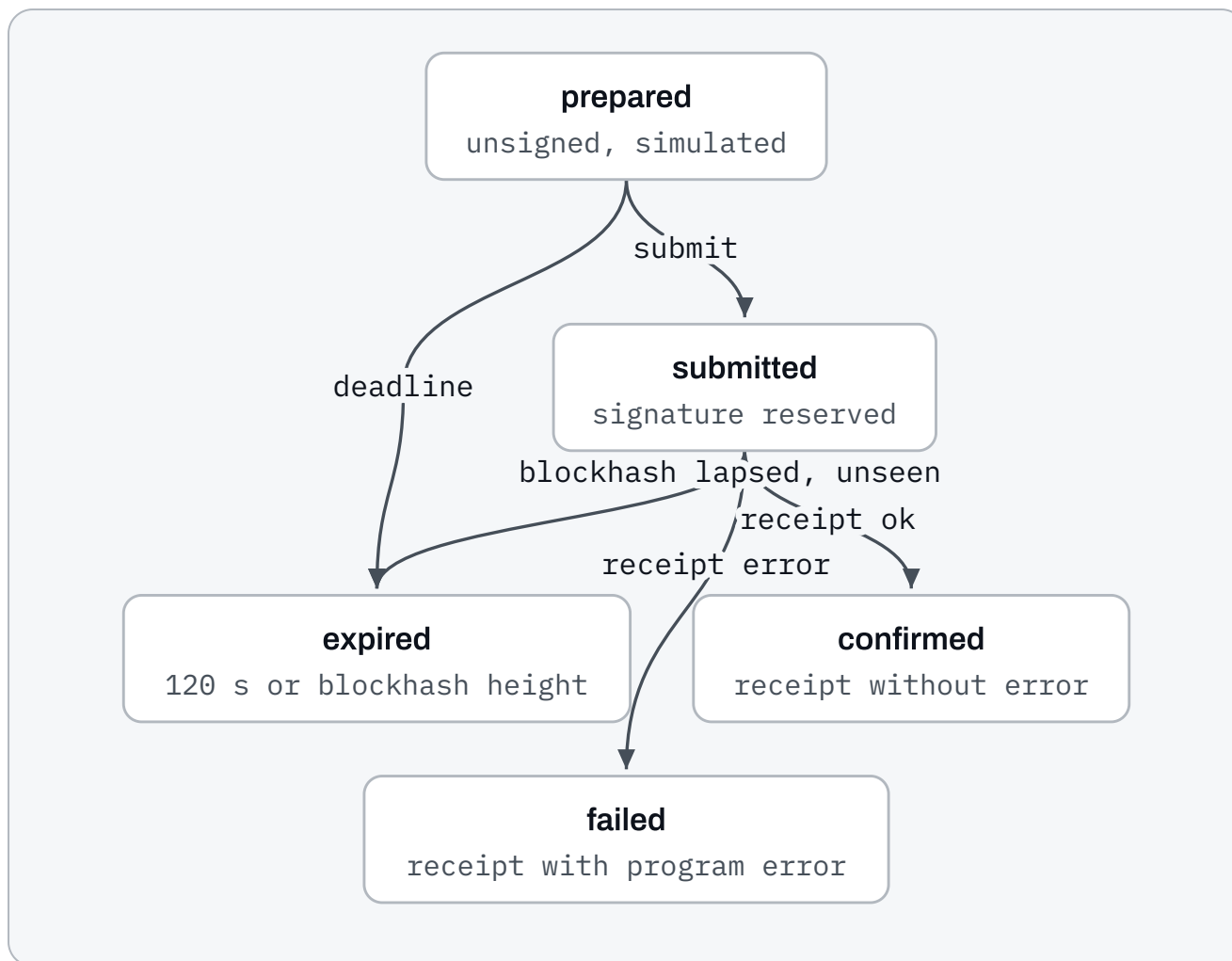


Figure 48. The five states of a prepared transaction and what moves each one.

GET /v1/transactions/:id , open only to the preparing wallet, drives these transitions. A receipt makes a submitted record **confirmed** or **failed** , and a confirmation triggers a fresh indexer pass so the next read already includes it. Without a receipt the record stays **submitted** until its blockhash lapses; then the API asks for the signature's status with full history search, up to three attempts. No record makes it **expired** , a recorded error makes it **failed** , and a recorded success keeps it **submitted** until the receipt arrives. If history cannot be read the API answers 503 and changes nothing, because a missing recent result does not prove that a payment never landed. Preparing a transaction emits no activity and no event.

Reads and live events

Live reads come from committed projections. Lists return { items, nextCursor } , 20 items by default and at most 100, and every read checks the session's identities against the resource.

Endpoint	Who may read	Returns
GET /v1/operators/:id	the operator's admin	operator, vaults, bond, lines, the 10 latest activities
GET /v1/operators/:id/pub lic	anyone	operator record, coverage, line count, total line limit

Endpoint	Who may read	Returns
GET /v1/vaults/:id	admin, session key, guardian, co-signer	vault and policy; rules and payments for admin and session key; approvals also for the co-signer
GET /v1/bonds/:id	admin	bond with coverage and pending withdrawal, lines, default events
GET /v1/lines , GET /v1/lines/:id	admin or the line's merchant; the session key with voucher fields redacted	line state, latest voucher, unsettled and settleable amounts
GET /v1/merchants/:id	the merchant	lines, public profiles of its operators, open claims, escrow jobs, settlement history
GET /v1/default-events/:id	admin, or a merchant holding a line on that bond	event and its claims
GET /v1/escrows/:id	admin, session key, merchant, arbiter	escrow, quote, deadlines and the actions this viewer may take
GET /v1/activity	scoped to an operator, vault, line or merchant the session owns	activity feed
GET /v1/protocol	anyone	voucher domain, token, fixed parameters, governed configuration

A line's `latestVoucher` is visible to the merchant holding it and to the operator only when the merchant shares it. An escrow's `allowedActions` follow from its state and chain time, narrowed to the roles the viewer holds.

Live events are server-sent events at `GET /v1/events?topics=...`, with 1 to 20 topics among `protocol`, `operator:<id>`, `vault:<id>`, `line:<id>`, `merchant:<id>`, `bond:<id>`, `default:<id>` and `escrow:<id>`. Topics are authorized when the stream opens and again on every poll, about once a second, after the session's identities are recomputed, so a wallet that loses a role loses the stream. Events come from the durable `ChainEvent` sequence: each `id` is a sequence number, and `Last-Event-ID` or `after` resumes from it. The server sends `retry: 2000`, at most 1,000 events per read and a heartbeat every 15 seconds. An `activity` event carries the record; an `invalidate` event names resources to refetch, and an activity the session may not see is reduced to an `invalidate` on its vault topic. Logout or expiry ends the stream.

One error model

Every response uses one envelope: `{ data, meta }` on success and `{ error: { code, message, retryable, details? }, meta }` on failure, with `meta.requestId` a fresh UUID also sent as `X-Request-Id`.

HTTP status	Codes	Retryable	Raised when
400, 413, 415	<code>INVALID_INPUT</code>	no	malformed JSON, unknown or invalid fields, a body over 64 KiB, a non-JSON content type
401	<code>UNAUTHORIZED</code>	no	no or expired session, an unknown or reused challenge, an invalid signature

HTTP status	Codes	Retryable	Raised when
403	FORBIDDEN	no	a disallowed origin, a missing or wrong CSRF token, a wallet without the required identity
404	NOT_FOUND	no	a resource that is not indexed
409	CONFLICT , IDEMPOTENCY_CONFLICT , DEADLINE_PASSED , INVALID_TRANSITION , SIMULATION_ONLY , mapped program codes	no, except an expired preparation (details.prepareAgain): prepare again under a new key	a reused key, an expired preparation, a refused simulation, a program refusal of the signed bytes
429	RATE_LIMITED	yes	the per-client request limit, with Retry-After
500, 503	INTERNAL_ERROR	yes	transport or database failure, an incomplete broadcast
503	CONFIGURATION_REQUIRED	no	the protocol is not initialized or indexed, or the RPC belongs to another cluster

Program refusals enter the same model. `programError` reads the Anchor log line `Error Code: <Name>` and maps it: `VaultFrozen` to `VAULT_FROZEN`, both window-budget errors to `WINDOW_BUDGET_EXCEEDED`, `PerTransactionCapExceeded` and `LineLimitExceeded` to `PER_TRANSACTION_CAP_EXCEEDED`, `ClaimWindowClosed` to `DEADLINE_PASSED`, `Unauthorized`, `MerchantNotAllowed` and `ActorNotAllowed` to `FORBIDDEN`, and `VoucherReplay`, `VoucherExpired`, `ApprovalRequired`, `ApprovalUsed`, `ApprovalExpired`, `CoverageTooLow` and `WithdrawalCoolingDown` to codes of the same names. An unmapped name becomes `INVALID_TRANSITION`. The name travels in `details.programCode`, and a program refusal is never retryable: the same bytes against the same state fail the same way.

11.2 The indexer and projections

The indexer turns chain state into rows the API can query and the workers can schedule from. Reading accounts directly cannot provide either history (closed claims, approvals and escrows leave only their events) or access control (a merchant's view joins several accounts and filters private evidence). Every row is a projection that the indexer can rebuild from the chain.

A pass is identified by a source key, `bonded:` followed by the first 56 hex characters of SHA-256 of `<programId>:<mint>`, so one deployment's cursor never advances another's. An idle pass costs two reads, the `Clock` sysvar and the newest program signature; if that signature is the one the cursor last observed, the pass only advances the cursor's slot and block time. Otherwise, and whenever a pass is forced after a confirmation, it runs in full:

1. Snapshot every program-owned account and list the program's successful signatures since the cursor, oldest first. List every vault token account's signatures too: a float payment is a top-level SPL `TransferChecked` signed by the session key as delegate and appears only there.
2. Fetch every receipt; if one is not yet available at the configured commitment, stop and retry rather than commit a partial view.

3. Snapshot again with the vault and bond token accounts (balance at byte 64, delegated amount at byte 121) and the vaults' freeze reserves, at or after the first snapshot's slot; stop if any receipt is newer.
4. Decode accounts by discriminator and events from `Program data`: lines, accepted only while `bonded` is the active invocation on the log's invoke stack.
5. In one database transaction under an advisory lock on the source key: skip if a later slot is already committed; upsert every decoded account; refuse the pass if a bond's recorded balance exceeds its SPL balance; mark rows of vanished accounts closed; insert each event once under `(signature, instructionIndex, eventIndex)`; record float payments and fundings from token transfers; rebuild claims, approvals and escrows from their event history; advance the cursor with the slot, the newest signature and the snapshot's block time.
6. After the commit, publish the new events.

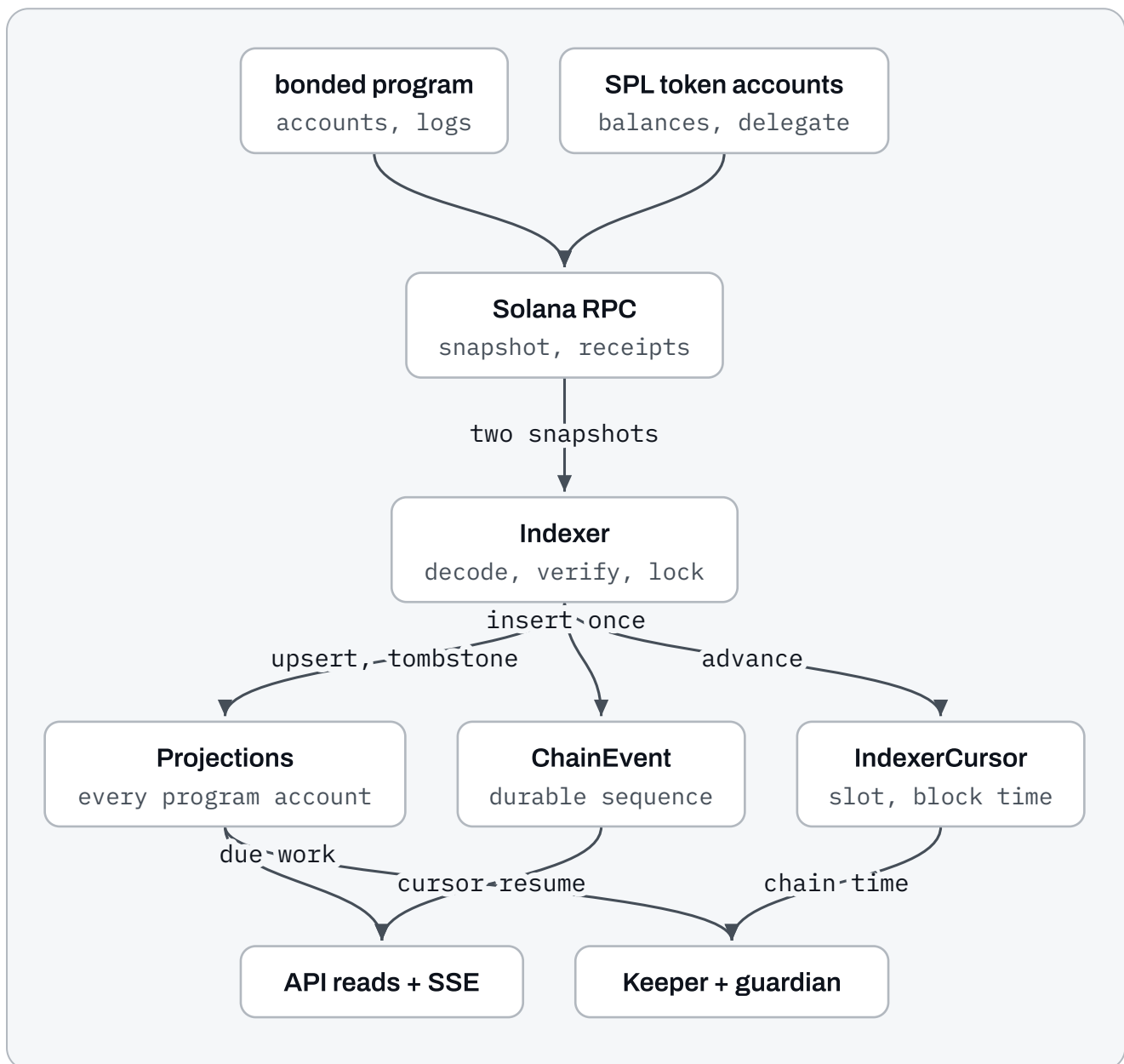


Figure 49. The data path from program accounts and logs to the reads, streams and jobs built on them.

A float payment is recorded only if it moves the configured six-decimal mint, in a positive amount, out of a token account the vault owns; a funding only if it comes from the operator's admin. A float transfer leaves the program's newest signature unchanged, so it is indexed by the next full pass, after an API confirmation or once anyone sends the next Bonded instruction, such as the vault's next refill. Account projections never depend on the event log: each pass rewrites every account row from the snapshot and closes vanished ones, so projections converge to the account state the RPC reports at the adapter's commitment (`confirmed` in the configured live service). Events and payments carry a commitment column (`CONFIRMED` , `FINALIZED` , `ORPHANED`), and every reader, from the activity feed to the guardian's window, excludes `ORPHANED` rows.

On top of the rows, the projection layer computes what a dashboard shows. Coverage uses the bond balance less any pending withdrawal against the sum of line limits. A line's `settleableNow` is zero while a default is active, the vault is frozen, the line or voucher has expired or the voucher's key is not authorized at the evidence time, and otherwise the unsettled amount when the vault holds it.

When the API runs as its own service (`src/server.ts`), only the worker reads the chain and writes projections. To get fresh state before a preparation or after a confirmation, the API writes a request ID into a refresh mailbox row (`refresh:` plus the source key) and polls every 200 milliseconds, for up to 12 seconds, until the worker records a refresh that started after the request. The worker checks the mailbox every second and otherwise reads the chain every `INDEXER_POLL_INTERVAL_MS` (30,000 by default, bounded to 1,000–3,600,000), so a quiet deployment costs two RPC reads per interval.

11.3 The keeper

Some transitions belong to no one: an escrow whose delivery deadline passed, a delivered job whose dispute window lapsed, a dispute the arbiter never resolved, a default whose claim window closed. The program makes each permissionless, so no party can hold funds by staying silent. The keeper executes them promptly. It holds no special authority: any merchant or operator can submit the same instructions. Each pass reconciles projections, reads chain time from the cursor and selects due work.

Due work	Selected when (chain time)	Instruction	What the program re-checks
Undelivered escrow	<code>FUNDED</code> and <code>deliverBy < now</code>	<code>refund_on_timeout</code>	state <code>FUNDED</code> , <code>now > deliver_by</code> ; principal and reserved fee return to the vault
Delivered, undisputed escrow	<code>DELIVERED</code> and <code>disputeUntil ≤ now</code>	<code>release_after_window</code>	state <code>DELIVERED</code> , <code>now ≥ dispute_until</code> ; pays a token account the merchant owns
Unresolved dispute	<code>DISPUTED</code> and <code>resolveUntil ≤ now</code>	<code>refund_after_resolution_deadline</code>	state <code>DISPUTED</code> , <code>now ≥ resolve_until</code> ; refunds the vault
Default with claims	not <code>PAID</code> and <code>windowEnd ≤ now</code>	<code>execute_default_payout</code> , one claim per transaction	<code>now ≥ window_end</code> ; a paid claim returns without effect
Default without claims	not <code>PAID</code> , <code>windowEnd ≤ now</code> , zero claims	<code>finalize_empty_default</code>	zero claims, <code>now ≥ window_end</code>

The other permissionless instructions, `refill_float` , `close_approval` and `close_freeze_checkpoint` , are open to the same keeper role through the API, and the agent SDK issues `refill_float` for its own vault.

The keeper's key has one permissioned job as well: it is the protocol's buyback crank (§10.7). Every five minutes (`BUYBACK_INTERVAL_MS` , 300,000) it reads the fee account; once that holds at least the configured minimum, it asks Jupiter for an exact-in quote from USDC to \$BONDED with at most 48 accounts (`BUYBACK_MAX_ACCOUNTS`) and 50 bps of slippage (`BUYBACK_SLIPPAGE_BPS`), and builds one versioned transaction over Jupiter's lookup tables and the protocol's own (`BUYBACK_LOOKUP_TABLES`): compute budget, `claim_buyback` with the quote's minimum, Jupiter's swap, `burn_buyback` . It simulates the transaction, sends it, and logs one line with the USDC spent, the \$BONDED burned and the signature. A failed quote waits for the next cycle. While no mint is configured it logs once that the buyback waits for a token mint and leaves the fees in place; on devnet, where Jupiter has no routes, it records the configuration and skips.

Running a crank twice must never pay twice, and a crashed keeper must never leave a payout stuck. Four mechanisms provide both.

- *Durable attempts.* Each escrow and each claim has its own attempt row, so one merchant's broken destination fails only that row and every other claim and escrow is still attempted.
- *Leases.* A worker takes a row with a conditional update that succeeds only when no lease is held or it has lapsed, and holds it for 60 seconds.
- *Resume before retry.* An attempt with a preparation reads its status first. If it is `submitted` , the keeper resubmits the stored signed bytes under their original key and never signs again, even when a replacement worker with a different key takes the job; if `confirmed` , the attempt is done. Only a missing, expired or failed preparation leads to a new one, keyed `keeper:<default>:<claim>:<attempt>` or `keeper:escrow:<escrow>:<attempt>` .
- *Program idempotence.* A paid claim returns without effect and payout closes the claim; release and refund close the escrow. A duplicate fails simulation at preparation, and the keeper never signs a refused simulation.

$$\text{retry_delay_ms} = \min(60\,000, 1\,000 \times 2^{\min(\text{attempts}, 6)})$$

`attempts` the attempt count stored on the row

Retries come 2, 4 and 8 seconds apart and settle at 60 seconds from the sixth attempt. A `KeeperSigner` supplies `wallet` , `signTransaction` and optionally a resolver naming a merchant-owned replacement destination; the program checks that destination's owner and mint, so the resolver can only choose among the merchant's own token accounts. Without a signer a pass only queues work. The keeper key exists only in the worker process, as `KEEPER_SECRET_KEY` .

11.4 The guardian worker

A guardian key can freeze a vault and nothing else (§12.1, §12.7); the guardian worker decides when. It applies exact integer rules to each unfrozen vault whose own or whose operator's guardian is the configured key, using the vault's committed payments other than tab settlements: float transfers, direct `pay` payments and escrow fundings. Tab settlements are excluded because the merchant chooses when to settle and how much cumulative debt a settlement covers. A freeze revokes the float delegate and stops every program payment from the vault.

Every rule value is operator configuration with no default: the window and baseline lengths, the minimum burst size, the minimum number of new counterparties, the spike floor, the spike multiplier in basis points (above 10,000) and the window cap.

$$\begin{aligned} \text{current} &= \text{payments with now} - \text{windowSeconds} < \text{at} \leq \text{now} \\ \text{baseline} &= \text{payments with now} - \text{windowSeconds} - \text{baselineSeconds} < \text{at} \leq \\ &\quad \text{now} - \text{windowSeconds} \\ \text{new_counterparty_burst} &\Leftrightarrow |\text{current}| \geq \text{minimumBurstPayments} \wedge \\ &\quad \text{new_merchants} \geq \text{minimumNewCounterparties} \\ \text{spend_spike} &\Leftrightarrow \text{spent} \geq \text{minimumSpikeSpend} \wedge \text{spent} \times \text{baselineSeconds} \\ &\quad \times 10\,000 > \text{before} \times \text{windowSeconds} \times \text{spikeMultiplierBps} \\ \text{window_spend_cap} &\Leftrightarrow \text{spent} > \text{maximumWindowSpend} \end{aligned}$$

<code>spent, before</code>	Σ gross amount of current and baseline payments, USDC base units
<code>new_merchants</code>	distinct merchants in current with no payment at or before <code>now - windowSeconds</code>
<code>now</code>	block time of the last committed indexer pass

The spike rule compares spending rates by cross-multiplying each window's total with the other window's length, so it stays exact in integers.

Example, with the local stack's rule values. The window is 60 seconds, the baseline 3,600 seconds, the spike floor 100 USDC (100,000,000 base units), the multiplier 50,000 basis points (five times) and the cap 500 USDC. A vault spent 36 USDC in the baseline hour and 150 USDC in the last minute. The spike test compares $150,000,000 \times 3,600 \times 10,000 = 5.4 \times 10^{15}$ with $36,000,000 \times 60 \times 50,000 = 1.08 \times 10^{14}$; the current rate is 250 times the baseline rate and above the floor, so `spend_spike` fires, while the cap does not.



Figure 50. The guardian worker freezes a vault after an alert and survives a lost response.

A job's ID is SHA-256 of the guardian, the vault and the triggering payment, so one alert never creates two jobs and an old alert cannot refreeze a vault after recovery. Every write is fenced to the job's lease. The worker stores the signed wire and signature before broadcasting and, after a lost response, finds the receipt or rebroadcasts the same bytes, never signing a second freeze. A dry run returns alerts without creating jobs or signing.

A guardian needs no SOL. The operator pre-funds the vault's freeze reserve, a system-owned PDA at `[b"freeze-reserve", vault]`, with `fund_freeze_reserve`, and the reserve pays each freeze checkpoint's rent; a separate sponsor pays the network fee. `createSponsoredGuardianExecutor` builds the real `freeze`, checks the RPC's and the indexed protocol's genesis hashes against its configuration, and accepts a signed transaction only if the message is unchanged, the sponsor is fee payer and exactly the guardian's and sponsor's signatures are present and valid. The keys stay in the caller's transport.

11.5 The merchant middleware

The merchant middleware lets a metered service bill an agent in arrears without a facilitator round trip. It answers an unpaid request with a `402` offering the `bonded-tab` scheme, verifies the agent's voucher against cached line state, meters and serves the request once, and hands settlement to a worker. The core is a `Workers-compatible Request / Response` handler with no Node imports; adapters mount it in Hono

(`tab.hono()`) and `Express ( expressBondedTab(tab, origin, serve) ,` before body parsers, with an exact origin and no trust in forwarded headers); `/postgres` supplies a store and `/worker` the settlement worker. The library never creates or loads a private key.

For a merchant that uses the deployed service as it is, `/node` wraps all of this in one call.

`createBondedMerchant({ apiUrl, authenticationOrigin, rpcUrl, programId, genesisHash, keypair, databaseUrl, price, admit })` signs in to the API with the merchant's key, reads lines through it, takes the line from the `X-Bonded-Line` header unless `lineFor` says otherwise, gives every timing threshold a default that satisfies the constraints below, and returns `hono()` , `express()` and `handle()` adapters, `start()` and `stop()` for the settlement and claims worker, and `quote()` , `deliver()` and `release()` for escrow jobs. The worker's signer authorizes each preparation with the agent SDK's own reconstruction (`authorizePreparedTransaction`), so the merchant signs only the action it asked for. The meter's three tables ship with the package as SQL and are created by the package's `bonded-merchant migrate` command, run with `DATABASE_URL` set (for example `npx --package https://bonded.cash/packages/bonded-merchant-middleware-0.1.0.tgz bonded-merchant migrate`).

Every threshold is explicit, and the constructor refuses combinations that would let a voucher expire before it can settle:

$$\begin{aligned} \text{voucherTtlSeconds} &> 2 \times \text{settlementHorizonSeconds} \\ \text{cacheMilliseconds} &\leq 1\,000 \times \text{settlementHorizonSeconds} \\ \text{serviceLeaseMilliseconds} &> \text{maximumServeMilliseconds} \\ \text{worker retry and lease} &< 1\,000 \times \text{settlementHorizonSeconds} \end{aligned}$$

`settlementHorizonSeconds` scheduler cadence plus confirmation time plus every retry allowance

A request passes these checks in order:

1. *Admission.* The host's `admit(request)` returns a stable authenticated `clientId` or refuses. Stored responses and leases are bound to that client, so a captured payment header cannot retrieve another client's response.
2. *Key, domain, digest.* An `Idempotency-Key` of 8–128 safe characters is required; the merchant's RPC must report the configured genesis hash; the request digest is SHA-256 of `[METHOD, URL without fragment, requestKey, SHA-256(body)]` with the body bounded by `maxRequestBytes` .
3. *Challenge.* Without `Payment-Signature` , the middleware prices the request, reads line state (cached, and re-read when the charge or outstanding amount reaches `rereadIncrement`) and computes the next voucher's cumulative amount, nonce and expiry. Credit pauses with `409 DEADLINE_NOT_REACHED` and settlement is queued when the line's expiry, a pending key switch or the newest outstanding voucher's expiry falls inside the horizon. A frozen vault, an inactive line or an outstanding amount above the line's effective limit is refused. Otherwise the answer is `402` with one `bonded-tab` offer, in the body and base64-encoded in `Payment-Required` .

4. *Payment*. `Payment-Signature` must name network `solana:<genesis hash>`, the same request key and line. A known key replays its stored response or answers busy, conflict or forbidden. A new voucher must carry this request's usage digest, match every offered field, be signed by the line's key effective now and expire after the horizon but no later than offered.
5. *Reserve, serve, persist*. Under an advisory lock on merchant and line, the store accepts the voucher only if its cumulative amount is exactly the previous base plus the charge and its nonce exactly one more, and queues settlement. The service runs with the request key, which it must honor for external side effects; its bounded response is stored and returned, and retries replay it without billing twice.

```
usage_digest = SHA-256(JSON[request_digest, charge, units])
cumulative_next = max(meter.cumulative, line.settled_cumulative,
                      line.retired_cumulative) + charge
nonce_next = max(meter.nonce, line.maximum_nonce) + 1
expiry = min(now + voucherTtlSeconds, line.expires_at)
```

`charge, units` positive u64 values returned by `price(request)`

Example. The local stack's example merchant charges 1,000 base units (\$0.001) per `GET /metered`. Twenty requests on a fresh line produce vouchers with nonces 1 to 20 and cumulative amounts 1,000 to 20,000. Each settlement pays only the delta since the previous one, so once the newest voucher settles the line's `settled_cumulative` is 20,000, however many intermediate settlements ran.

The settlement worker (`createMerchantWorker`) reads chain time, line state and defaults on each pass and keeps three kinds of job.

Job	Created when	Instruction	Deadline
Attest	a key switch is pending on the line	<code>attest_line_voucher</code> with the newest outgoing-key voucher	strictly before the switch's <code>effectiveAt</code>
Claim	a default on the bond is in its claim window	<code>file_claim</code> with the newest voucher eligible at the default	strictly before <code>window_end</code>
Settle	no unpaid default, and a stored voucher above the line's floors	<code>settle_tab</code> with the newest eligible voucher	before the voucher's and the line's expiry

The voucher is chosen by query: the newest stored voucher whose signer, expiry, nonce and cumulative amount fit the key authoritative at the relevant time, including outgoing-key attestations and freeze references, so a latest voucher that can no longer settle does not hide an older one that can. A job re-queues when a voucher with a higher nonce arrives. Each job runs under a lease: the worker prepares through the API, refuses a preview whose action differs from the expected one or whose simulation was refused, checks the 1,232-byte limit and calls the host's `sign(prepared, expectedAction)`, which must authorize the transaction against the merchant's own pins and state (the example merchant rebuilds the message and compares it byte for

byte). The signed wire is stored before submission and resubmitted identically after an ambiguous response. A single-voucher settlement prepared by the API, with its token-account guard and memo, measures 1,014 bytes and 82,363 compute units.

11.6 The agent SDK and the MCP server

The agent SDK is how an agent signs. It accepts a session signer and nothing else: a local 32-byte Ed25519 seed, or a thin transport to Turnkey, Crossmint or Coinbase that keeps provider credentials outside the SDK and whose every signature the SDK verifies strictly. Its central rule is that the preparing API is not trusted to say which program and cluster it serves: the caller pins `programId` and `genesisHash` independently, and `verifyNetwork()` compares both with the API's voucher domain, and the genesis hash also with the SDK's own RPC, before preparing, after preparing, before submitting and before signing any voucher or verifying any quote (§12.5).

Every SDK transaction takes one path: verify, prepare, verify again, require the prepared action to equal the requested one, check that the preparation is live with an allowed simulation, fits 1,232 bytes, hashes to its declared message hash and has the session key as fee payer and sole signer, rebuild the complete message from the SDK's own reads and compare it byte for byte, sign, verify once more and submit with the key `<idempotencyKey>:submit` . The SDK signs five instructions this way: `refill_float` , `pay` , `pay_escrow` , `accept` and `dispute` .

- *Vouchers.* `signVoucher` reads the line fresh, refuses a frozen, inactive or expired line, requires the signer to be the line key effective now, and advances nonce and cumulative amount past the chain's floors and every voucher it has issued, inside the effective limit and the line's expiry. A `voucherStore` restores issued vouchers across restarts; without one, an unexplained cumulative jump is refused.
- *Tabs.* `fetchWithTab(url, init, { maxCharge, requestKey, maxRetries, maxRequestBytes })` buffers a body once (1–16,777,216 bytes), refuses redirects, and before signing checks that the challenge names this URL, method, network, mint, merchant, request digest and usage digest, that the charge is positive and within `maxCharge` , and that the offered cumulative amount equals fresh line state plus the charge. A request key authorizes one voucher for life; retries resend it, and a stale challenge is refreshed at most twice. A durable `requestAuthorizationStore` saves each authorization before payment; without it, the in-memory cache stops at 1,024.
- *Float.* `refillFloat` books the consumed allowance and re-approves the delegate within the window budget; the agent's own `x402 exact` client then pays open-web endpoints with a top-level `TransferChecked` signed by the session key as delegate of the vault's token account. The SDK refills; it doesn't send float payments itself.
- *Escrow.* `describeEscrow` verifies the signed quote and returns principal, fee, total debit, intent hash, resolution duration and `timeoutOutcome: 'refund'` . `payEscrow` binds funding to `maximumTotalDebit` and the reviewed intent hash. A named arbiter must be in the caller's explicit `arbiterAllowlist` and differ from the merchant.

The MCP server exposes these operations to a model over stdio JSON-RPC (protocol 2025-11-25) and opens no HTTP port. Tool schemas reject unknown arguments, and failures report a code and message without keys or transport internals.

Tool	SDK method	Signs	Ceiling applied
<code>bonded_verify_net</code> <code>work</code>	<code>verifyNetwor</code> <code>k</code>	nothing	none

Tool	SDK method	Signs	Ceiling applied
<code>bonded_login</code>	<code>login</code>	the authentication challenge	none
<code>bonded_refill_float</code>	<code>refillFloat</code>	<code>refill_float</code>	the vault's on-chain window budget
<code>bonded_pay</code>	<code>pay</code>	<code>pay</code>	full amount reserved against per-call cap and window budget
<code>bonded_describe_escrow</code>	<code>describeEscrow</code>	nothing	none
<code>bonded_pay_escrow</code>	<code>payEscrow</code>	<code>pay_escrow</code>	total debit including fee reserved, passed as <code>maximumTotalDebit</code>
<code>bonded_escrow</code>	<code>escrow</code>	nothing	none
<code>bonded_accept_escrow</code>	<code>acceptEscrow</code>	<code>accept</code>	none; releases funds already escrowed
<code>bonded_dispute_escrow</code>	<code>disputeEscrow</code>	<code>dispute</code>	none
<code>bonded_fetch_tab</code>	<code>fetchWithTab</code>	one voucher	min(per-call cap, remaining window budget) reserved, passed as <code>maxCharge</code>

The operator sets the spending ceilings in configuration. `BONDED_MCP_PER_CALL_CAP`, `BONDED_MCP_WINDOW_BUDGET` and `BONDED_MCP_WINDOW_SECONDS` are positive base-unit integers in configuration; no tool argument can raise them, and `maxCharge` is not an argument at all. All spending tools share one budget, and each reservation is persisted before the SDK signs.

$$\text{window_start} = \lfloor \text{now} \div \text{windowSeconds} \rfloor \times \text{windowSeconds}$$

$$\text{reserve_pay} = \text{amount, if amount} \leq \text{perCallCap and } \Sigma \text{ window} + \text{amount} \leq \text{windowBudget}$$

$$\text{reserve_tab} = \min(\text{perCallCap}, \text{windowBudget} - \Sigma \text{ window})$$

$\Sigma \text{ window}$ reservations already recorded in the current window

A reservation is keyed by agent, tool and idempotency key and fingerprinted by the canonical arguments: an identical retry in the same window reuses it, changed arguments are refused, and failed or ambiguous operations keep their reservations. The CLI store writes under an exclusive cross-process lock with atomic replacement, stays closed after a crash until the operator resolves the lock, and holds at most 10,000 authorizations. The fetch tool reaches only origins in `BONDED_MERCHANT_ORIGINS`, returns at most 32 KiB and labels the body `untrusted_merchant_content`: nothing a merchant returns can authorize a tool call or change a ceiling. The CLI reads its session keypair and keeps its budget file at the absolute paths its operator

configures, refusing a path that resolves through a symlink to somewhere else and never reading the default Solana wallet; its arbiter and origin allowlists are explicit JSON arrays in which `[]` permits none, and `BONDED_MERCHANT_HEADERS` can add per-origin headers such as a merchant's API key.

12. Security design

Bonded's security rests on three layers, each of which works without trusting the one above it. The program decides every movement of USDC and checks every signer, limit, deadline and replay floor itself. The keys are separated so that each signs one narrow set of instructions. The services prepare, relay and crank but hold no key that a user's funds depend on, and every party that signs re-derives what it signs from its own configuration. This chapter describes each layer as built: what each key can sign, which checks run on every payment, how replay is closed, how a signed transaction is tied to what the wallet reviewed, and how bonds, freezes, services and upgrades are constrained.

12.1 Key separation

Bonded gives each role its own key, and each key the smallest set of instructions its job needs, so a compromised key exposes only its own role's instructions. The program enforces the separation in its account constraints: each instruction names the signer it accepts, and the account it changes stores the key that may sign for it.

Key	Stored as	Can sign	Cannot sign
Operator admin	<code>Operator.admin ; the operator PDA is [b"operator", admin]</code>	<code>create_operator , create_vault , set_policy , set_merchant_rule , set_passkey_credential , rotate_session_key , withdraw , fund_freeze_reserve , post_bond , open_line , resize_line , renew_line , rotate_line_key , request_withdrawal , finalize_withdrawal , rotate_operator_guardian , rotate_vault_guardian , freeze , unfreeze ; accept or dispute as principal</code>	<code>vouchers , pay , pay_escrow , settle_tab , file_claim , resolve , protocol configuration ; an immediate guardian, line-key or limit-decrease change; a bond withdrawal before unlock; a vault withdrawal while frozen</code>
Session key	<code>vault.policy.session_key ; copied to CreditLine.voucher_session_key at open_line</code>	<code>pay , pay_escrow , accept , dispute ; a top-level TransferChecked as delegate up to the float allowance; vouchers for lines whose key it is</code>	<code>policy, rules, lines, withdrawals, approvals for its own payments, unfreeze ; a payment to a merchant without a rule</code>
Co-signer	<code>vault.policy.cosigner : an Ed25519 key or a passkey credential PDA</code>	the 128-byte approval message, verified by the Ed25519 or secp256r1 precompile	any transfer; an approval past its expiry or the vault's policy revision
Guardian	<code>Operator.guardian or vault.policy.guardian</code>	<code>freeze</code>	<code>unfreeze , any transfer, any policy change</code>
Keeper	any key	<code>refill_float , execute_default_payout , finalize_empty_default , refund_on_timeout , release_after_window , refund_after_resolution_deadline , close_approval , close_freeze_checkpoint</code>	a payee of its choosing: each pays an account owned by the merchant (or vault) that account state names
Merchant	<code>CreditLine.merchant ; the quote signer</code>	<code>settle_tab , file_claim , attest_line_voucher , mark_delivered , close_claim , close_default_placeholder ; the 169-byte quote</code>	a settlement above the limit or without the line key's voucher; a claim after the window; arbitration of its own quote

Key	Stored as	Can sign	Cannot sign
Arbiter	<code>Escrow.arbiter</code> , from the signed quote	<code>resolve</code> , strictly before <code>resolve_until</code>	any action on an escrow that is not disputed, or after the deadline
Protocol authority	the current loader-v3 upgrade authority in <code>ProgramData</code>	<code>initialize_protocol</code> , <code>update_protocol</code> , <code>configure_escrow_fee</code> , <code>configure_escrow_resolution</code> , <code>configure_buyback</code>	any vault, bond, line or escrow; the fee treasury; a tab fee above the immutable maximum; an open line's fee or a funded escrow's deadline
Buyback crank	<code>ProtocolConfig.buyback_crank</code> , set by <code>configure_buyback</code> <code>k</code>	<code>claim_buyback</code> and <code>burn_buyback</code> , only together in one transaction	USDC kept rather than swapped; a swap below the quoted minimum; any other instruction
Upgrade authority	<code>ProgramData</code> of the <code>bonded</code> program	a program upgrade	anything else directly (§12.9)

Several limits are structural. The admin is fixed for the life of its operator account, because the account's address derives from the admin key; the admin can be a Squads smart-account PDA, so an operator keeps its existing multisig [15]. A guardian replacement, a line voucher-key replacement and a limit decrease each take effect only 72 hours (`DECREASE_DELAY`) after the admin schedules them, and `set_policy` refuses a policy whose guardian differs from the current one, so the delay cannot be bypassed. Rotating the session key revokes the float delegate at once but leaves each line's voucher key on its own delayed schedule, so a merchant extending credit knows which key will be authoritative and has 72 hours to attest its outstanding vouchers before a switch (see §6 and §7). Because a precompile verifies the co-signer's approval message, whoever pays for the `approve_payment` transaction gains none of the co-signer's authority.

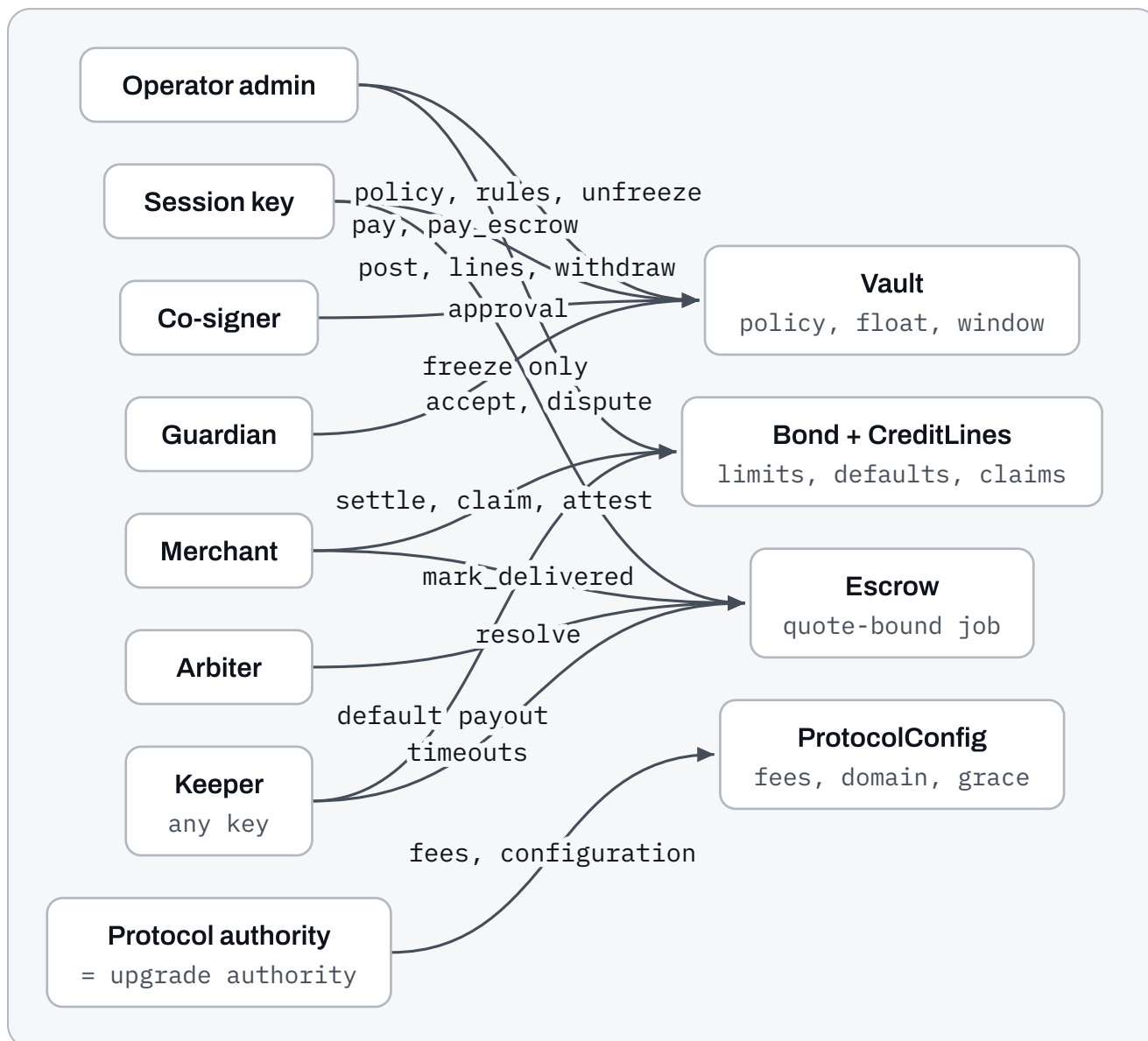


Figure 51. Each key reaches only the accounts and instructions its role needs.

12.2 What the program enforces on every payment

"Spent outside policy" is a statement about where USDC can go. A vault's USDC sits in a classic SPL token account whose authority is the vault PDA, `[b"vault", operator, agent_id]`. Only the program can sign for that PDA, so USDC leaves the vault only through an instruction the program implements, plus one SPL allowance the program itself sets.

Exit	Instruction	Token authority	What bounds it
Float spend	top-level SPL <code>TransferChecked</code>	session key as delegate	the delegated allowance, set by <code>refill_float</code> to at most <code>min(float_cap, window budget - window spent)</code>
Direct payment	<code>pay</code>	vault PDA, by CPI	merchant rule, per-transaction cap, merchant and vault window budgets, approval above threshold, optional signed quote
Escrow funding	<code>pay_escrow</code>	vault PDA	the same checks on principal plus fee, with a signed quote required

Exit	Instruction	Token authority	What bounds it
Tab settlement	<code>settle_tab</code>	vault PDA	the merchant's line, the line key's voucher, $\text{delta} \leq \text{line limit}$
Owner withdrawal	<code>withdraw</code>	vault PDA	the admin's signature; refused while frozen

Escrow refunds and the surplus of a release return to the vault's own token account, and fees go only to the program-owned `fees` PDA. Every USDC transfer is a `TransferChecked` of `ProtocolConfig.usdc_mint` with 6 decimals; Token-2022 accounts are not accepted. A merchant's destination must be a token account the merchant owns: `pay` requires `merchant_token.owner == rule.merchant`, `settle_tab` requires the merchant as token authority, and an escrow release checks the destination's owner and mint.

`pay` and `pay_escrow` share one function, `book_spend`, which applies the policy in a fixed order: the vault is not frozen; the amount is positive; the window is reconciled; the rule's merchant is the paid merchant; the amount is within the per-transaction cap; a quote is present if the rule requires one; the merchant's window budget holds; the vault's window budget holds; and above the co-sign threshold, an approval for exactly this payment exists and is consumed.

$$\begin{aligned} \text{consumed} &= \text{last_approved} - \text{delegated_amount} \\ \text{window_spent} &\leftarrow (\text{window rolled over} ? 0 : \text{window_spent}) + \text{consumed} \\ \text{amount} &\leq \text{rule.per_transaction_cap} \\ \text{rule.window_spent} + \text{amount} &\leq \text{rule.window_budget} \\ \text{window_spent} + \text{amount} + \text{delegated_amount} &\leq \text{policy.window_budget} \\ \text{amount} > \text{cosign_threshold} &\Rightarrow \text{approval unused} \wedge \text{now} < \text{expiry} \wedge \\ &\text{revision} = \text{policy_revision} \wedge (\text{intent, merchant, amount}) \text{ match} \\ \text{float allowance} &= \min(\text{float_cap}, \text{window_budget} - \text{window_spent}) \end{aligned}$$

<code>delegated_amount</code>	float allowance the session key has not yet spent, read from the SPL token account
<code>last_approved</code>	the allowance granted at the last refill

Two choices in this arithmetic close the gap between lanes. Reconciliation books the float already spent before a new payment is judged, so float spending counts against the same window; and the check counts the outstanding allowance as already spent, so the float plus every program payment can never exceed the window budget even if the agent spends its whole allowance immediately afterward. A policy needs a nonzero session key, a positive window and `float_cap ≤ window_budget`. A policy change, a session-key rotation, a withdrawal and a freeze each reconcile the window and revoke the delegate, and a policy change increments

`policy_revision` , invalidating every unused approval. Every program payment also binds an intent hash over the vault, merchant, amount, nonce, quote hash and lane, which the program recomputes from the instruction's own fields; its `PaymentReceipt` at `[b"payment", vault, intent]` can exist only once (§12.3).

Example, with policy values in USDC. A vault has a float cap of 20, a window of 86,400 seconds, a window budget of 200, a co-sign threshold of 100, and one merchant rule with a cap of 150 per transaction and 400 per window. The agent has spent 3, 6 and 25 at that merchant and holds a full 20-USDC float allowance. A payment of 4,000 to an unknown address names a `MerchantRule` account at `[b"rule", vault, merchant]` that does not exist, and the program refuses it before any amount check. A payment of 4,000 to the allowed merchant, a decimal slip, fails `PerTransactionCapExceeded` . A 120-USDC batch job fails `ApprovalRequired` until the co-signer approves exactly that intent; with the approval it passes the window check, $34 + 120 + 20 = 174 \leq 200$, and the approval closes as it is consumed.

Tabs and escrow add their own invariants. A settlement pays only $\text{delta} = \text{voucher.cumulative} - \text{max}(\text{settled_cumulative}, \text{retired_cumulative})$, at most the line's limit, for a voucher signed by the line's key at the evidence time, unexpired then and no later than the line's expiry, on a line belonging to the signing merchant. An escrow funding debits $\text{amount} + \lfloor \text{amount} \times \text{escrow_fee_bps} \div 10\,000 \rfloor$ through `book_spend` and requires the merchant's unexpired quote for this vault, merchant, amount and lane, whose arbiter, if it names one, isn't the merchant. No instruction moves a vault's USDC to an address the operator's policy did not admit.

12.3 Replay protection

Every message a Bonded key signs carries a domain tag and is consumed by a state change that cannot happen twice. The tag ties a signature to one program on one cluster, so a voucher signed for devnet, for a test deployment or for another program can never settle against the production program.

$\text{tag}(\text{prefix}) = \text{first 8 bytes of SHA-256}(\text{prefix} \parallel \text{program_id} \parallel \text{cluster_genesis_hash})$

`voucher_domain = tag("Bonded voucher v2\0")`, stored in `ProtocolConfig` at initialization

`program_id`, `cluster_genesis_hash` raw 32-byte values, not base58 text

Signed message	Size	Domain	What makes a second use fail
Voucher	96 bytes	<code>Bonded voucher v2\0</code>	nonce must exceed <code>maximum_nonce</code> , cumulative must exceed <code>max(settled, retired)</code>
Quote	169 bytes	<code>Bonded quote v1\0</code>	<code>QuoteReceipt</code> at <code>[b"quote", vault, quote_hash]</code> is marked used; the escrow PDA <code>[b"escrow", vault, quote_hash]</code> is created once; the quote expires

Signed message	Size	Domain	What makes a second use fail
Approval v2	128 bytes	Bonded approval v2\0	Approval at [b"approval", vault, intent] is created once, used once and closed on use; it expires, dies with the policy revision and cannot be created once the intent has a receipt
Payment intent	32-byte hash of a 121-byte preimage	Bonded payment intent v1\0	PaymentReceipt at [b"payment", vault, intent] is created once
Passkey assertion	authenticator data 37–128 bytes, client data ≤ 512 bytes	challenge = base64url(SHA-256(approval message))	the signature counter must advance (or stay zero on both sides); origin and RP ID hashes must match
Sign-in challenge	11 lines	Bonded off-chain authentication v1	deleted when used; 300 seconds by default
Transaction	≤ 1,232 bytes	blockhash and memo bonded:<preparedId>	one preparation per wallet and idempotency key; one signed wire per preparation

Vouchers are cumulative, so replay protection is a comparison rather than a list of spent nonces. Each voucher states the total owed on the line, and the line stores the highest nonce and cumulative amount already settled. A settlement sets both; a default payout raises `retired_cumulative` and `maximum_nonce` to the claimed voucher's, so debt resolved by a default cannot be settled again after renewal; and a refilled claim must exceed the claim it replaces in both nonce and cumulative amount. *Example:* a line has settled nonce 20 at cumulative 20,000 base units. Replaying nonce 15 at 15,000 fails `VoucherReplay`; a new voucher with nonce 25 at 26,000 pays exactly the delta, 6,000.

A proof cannot be borrowed from elsewhere in the transaction. Through the instructions `sysvar` the program requires the `Ed25519` instruction immediately before the calling `Bonded` instruction, at the index the caller passed, with zero accounts, exactly 208 data bytes and the fixed offset header `[1,0,48,0,255,255,16,0,255,255,112,0,96,0,255,255]`, which points every offset into the proof instruction itself. One signature therefore verifies exactly the 96 bytes the program decodes; foreign instruction references, other offsets, multiple signatures and CPI wrappers are refused. Quotes and approvals follow the same rule with their own lengths. Off chain, the API binds each preparation to one request hash and one signed wire (§11.1), the middleware binds each request key to one client, request digest and voucher (§11.5), and the SDK and MCP budget bind each request key to one authorization (§11.6).

12.4 Transaction integrity between server and wallet

The API builds transactions and the wallet signs them, so the bytes broadcast must be the bytes the wallet reviewed, apart from additions that cannot move funds. The server prepares, the wallet signs what it reviews, and `submit` compares the signed message with the stored preparation before anything reaches the network. Browser wallets such as Phantom may add compute-budget instructions and Lighthouse assertions before signing; the comparison admits exactly these and nothing else:

1. The signed message may be legacy or version 0 but may not use address lookup tables, and re-serializing it must reproduce the submitted bytes exactly.
2. The recent blockhash, the fee payer (the signed-in wallet) and the set of required signers are unchanged.
3. No account is writable in the signed message unless it was writable in the prepared one.

4. Every prepared instruction appears in its original order with the same program ID, data and accounts, including each account's signer and writable flags.
5. The only extras are Compute Budget instructions with no accounts, anywhere, and Lighthouse instructions (`L2TExMFKdjpN9kozasaurPirfHy9P8sbXoAN1qA3S95`) after the last prepared instruction.
6. Every required signature verifies over the signed message.

Each rule closes a path. A compute-budget instruction without accounts can raise the fee or compute limit but cannot name a token account. A Lighthouse instruction is an assertion that fails the transaction if state differs from what the wallet expected, and trailing position runs it after Bonded's instructions. The writable-set rule stops any addition from writing an account the preparation did not already write, and the lookup-table rule keeps the account set to the static keys that were reviewed.

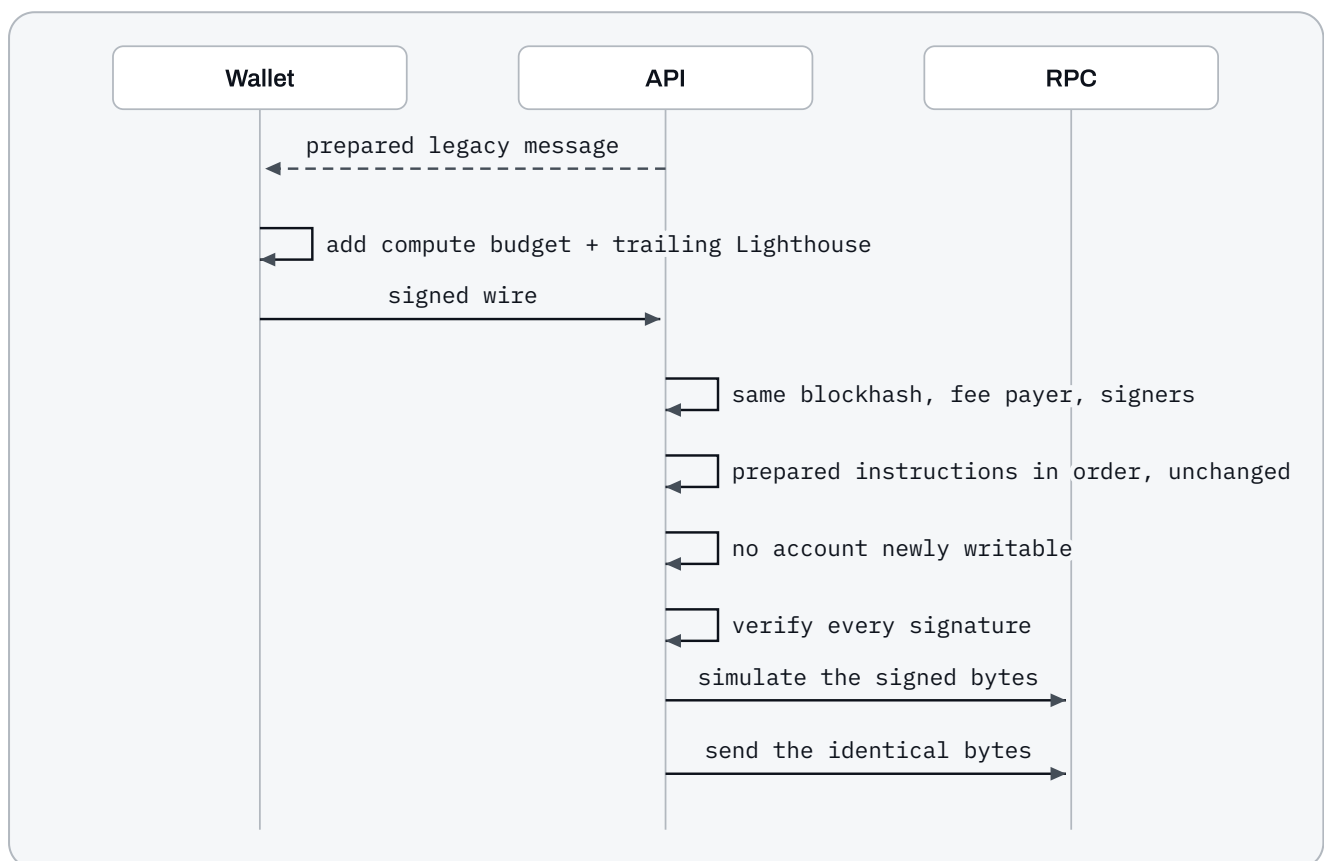


Figure 52. A browser wallet's additions are admitted only if every prepared instruction survives unchanged.

Admission by the comparison is not the last check: `submit` simulates the signed bytes before reserving the signature, so an addition the program rejects is refused before broadcast with the program's error. For instructions that verify a signature through the instructions `sysvar( settle_tab , file_claim , attest_line_voucher , approve_payment , approve_payment_passkey , pay with a quote and pay_escrow )`, the program requires the proof at the index the builder encoded, immediately before the Bonded instruction. The preparation therefore opens with its own Compute Budget instructions, counted into that index: a wallet that finds a compute budget keeps it rather than inserting one, and its Lighthouse guards trail the Bonded instruction, so the index holds. An addition that did move the proof would fail the re-simulation with `InvalidPrecompile` before broadcast. The merchant worker, the keeper and the agent SDK, which sign these instructions, add nothing; the SDK requires the preparation to hash to its declared message hash, to have the session key as fee payer and sole signer, and to match a message it rebuilt itself.

12.5 The SDK's independent pins

Comparing a signed message with its preparation guards the transaction in transit. The agent SDK guards the step before it as well: it checks the preparation itself, so a preparing API pointed at another program, another cluster or accounts the agent did not intend gets no signature. It trusts nothing the API says about the deployment.

- *Pins.* The operator configures `programId` and `genesisHash` independently. `verifyNetwork()` requires the API's voucher domain to name the pinned program and both the API's genesis hash and the SDK's own RPC `getGenesisHash()` to equal the pinned one, and recomputes the domain tag from the pins. It runs before preparing, after preparing, before submitting, and before signing a voucher or verifying a quote.
- *Re-derivation.* Every PDA in the expected transaction derives from the pinned program: protocol, fees, vault, rule, approval, payment receipt, quote receipt, escrow, and each associated token account from owner, token program and mint.
- *Own reads.* Through its own RPC the SDK reads `ProtocolConfig` and requires the pinned program as owner, its discriminator, the API's mint at byte 41, the derived fees PDA at byte 73 and the pinned genesis hash at bytes 126–158. It reads the vault and checks that its PDA re-derives from the stored operator and agent seed, that its token account is the derived associated account and that its session key is the signer. Approvals, escrows and payment receipts are read the same way, so rent recipients come from chain state rather than from the API.
- *Reconstruction.* From these values the SDK builds the complete expected transaction (the merchant's associated-account creation where the API adds it, the exact Ed25519 quote proof, the one requested Bonded instruction and the final memo) and compares the serialized message byte for byte. An extra program or transfer, a changed account or a changed amount fails the comparison.

Every other signer follows the same rule. The MCP server refuses to start unless its pins equal the SDK's. The merchant worker's `sign(prepared, expectedAction)` must authorize against the merchant's own pins and state; the merchant kit does so with the SDK's own reconstruction. The guardian executor checks the RPC's and the indexed protocol's genesis hashes before preparing, signing and submitting. And escrow review carries the pin through time: `payEscrow` binds the intent hash computed at review, which includes the fee, so a fee change before signing changes the intent and is refused.

12.6 Bond safety by construction

A bond protects merchants only if the operator cannot withdraw it ahead of the default that would claim it. Bonded makes that ordering impossible to break rather than detectable: every deadline that could need the bond falls before the bond's unlock time. §6 and §7 describe the bond, its lines and the default process; this section collects the rules that keep them in order.

$$\text{unlock_at} = \max(\text{now}, \text{longest_line_expiry}) + 259\,200 + \text{withdrawal_margin_seconds}$$

$$(\text{bond.balance} - \text{pending_withdrawal}) \times 10\,000 \geq \text{reserved_lines} \times \text{minimum_coverage_bps}$$

$$\text{payout} = \text{eligible, if } \Sigma \text{ eligible} \leq \text{pool; otherwise } \lfloor \text{pool} \times \text{eligible} \div \Sigma \text{ eligible} \rfloor$$

$$\text{eligible} = \min(\max(\text{voucher} - \text{settled_snapshot}, 0), \text{line_limit_snapshot})$$

259 200	CLAIM_WINDOW, 72 hours, a program constant
withdrawal_margin_seconds	configuration, 86 400 in the production parameter files
longest_line_expiry	high-water mark of every line expiry on the bond, never lowered
pool	the bond balance snapshotted at the first payout of the default

The reasoning runs one way. A voucher's expiry cannot exceed its line's, and no line outlives `longest_line_expiry`, so that is the last moment a merchant can present a voucher. If the vault is short then, a default opens and accepts claims for 72 hours. The bond therefore cannot be needed after `longest_line_expiry + 72 h`, and a withdrawal unlocks a configured margin later. Opening or renewing a line raises `longest_line_expiry` and pushes a pending unlock later, so a request cannot race ahead of new credit.

- *Coverage counts pending withdrawals.* Opening, raising and renewing lines check coverage against the balance less any pending withdrawal, and a withdrawal request must leave coverage intact.
- *Defaults supersede withdrawals.* A default opened by `settle_tab` clears the pending withdrawal; while it is active, no withdrawal can be requested or finalized and no line opened, raised or renewed.
- *Freezes lock collateral.* While any of the operator's vaults is frozen and until its recourse tail ends, withdrawals and matured decreases cannot be applied (§12.7).
- *Decreases wait.* A limit decrease applies only 72 hours after it is scheduled and only while collateral is unlocked; until then the full limit stays reserved. A claim snapshots the full limit when a freeze reference applies or collateral was locked when the default opened, and otherwise the limit effective at its reference time.
- *Claims close before payout.* `file_claim` requires `now < window_end`; `execute_default_payout` requires `now ≥ window_end` and snapshots the pool and eligible total once, so every claim is paid from the same snapshot, in full or pro rata, with rounding dust left in the bond.
- *Beta caps.* `post_bond` refuses a bond above 25,000,000,000 base units (\$25,000) or a protocol total above 250,000,000,000 (\$250,000).

Example, with the production margin of 86,400 seconds. An operator's longest line expires on day 30. A withdrawal requested on day 0 unlocks on day 30 + 3 + 1 = day 34. If on day 10 the operator opens a line expiring on day 45, the pending unlock moves to day 49. If a merchant opens a default on day 33, the request is cleared and the claims are paid first.

12.7 The freeze path

A freeze stops a compromised or misbehaving agent. It must act in one transaction, be cheap to trigger, and never become a way for an operator to keep merchants from their recourse.

`freeze` accepts the operator's admin, the operator's guardian or the vault's guardian. It books the float already spent, revokes the session key's SPL delegate and zeroes `last_approved`. When the vault was not already frozen it also creates a `FreezeCheckpoint` at `[b"freeze", vault, generation]` recording the freeze time, the generation, its episode and the operator's default count, with rent from the vault's freeze reserve or else from the signer; starts a new episode if the previous recourse tail has ended; and counts the vault as frozen on its operator. From then on `book_spend`, `refill_float` and `withdraw` refuse with `VaultFrozen`, and no line can be opened, renewed or raised.

Merchants keep their recourse. A voucher is judged at the checkpoint's freeze time, so a voucher valid when the vault froze stays valid however long the freeze lasts. Once the vault's cumulative frozen time in the episode reaches `freeze_grace_seconds`, a merchant's `settle_tab` opens a default regardless of the vault's balance, and the bond pays through the ordinary claim process (see §6 and §7).

$$\text{freeze_grace_seconds} \geq \text{MIN_FREEZE_GRACE_SECONDS} = 345\,600 \text{ (96 h)}$$
$$96 \text{ h} = 72 \text{ h guardian replacement delay} + 24 \text{ h response margin}$$
$$\text{recourse_until} = \text{unfreeze_time} + \text{freeze_grace_seconds} + 259\,200$$

`freeze_grace_seconds` configuration, 345 600 in the production parameter files

The floor keeps a guardian freezing in bad faith from pushing an operator into a forced default: the admin can schedule a new guardian, which takes 72 hours and can be applied while the vault is frozen, and still has 24 hours to unfreeze. Frozen time accumulates across an episode: an unfreeze adds the completed interval to `episode_frozen_seconds`, a refreeze continues from it and unfrozen time never counts, so rapid refreezes neither skip nor restart the grace.

`unfreeze` is the admin's alone. It finalizes the checkpoint, sets the vault's recourse tail to the unfreeze time plus the grace plus the 72-hour claim window, and raises the operator's `recourse_lock_until` to at least that. Until the tail ends the operator cannot request or finalize a bond withdrawal, apply a matured limit decrease or renew a line, so the collateral behind the frozen debt stays in place.

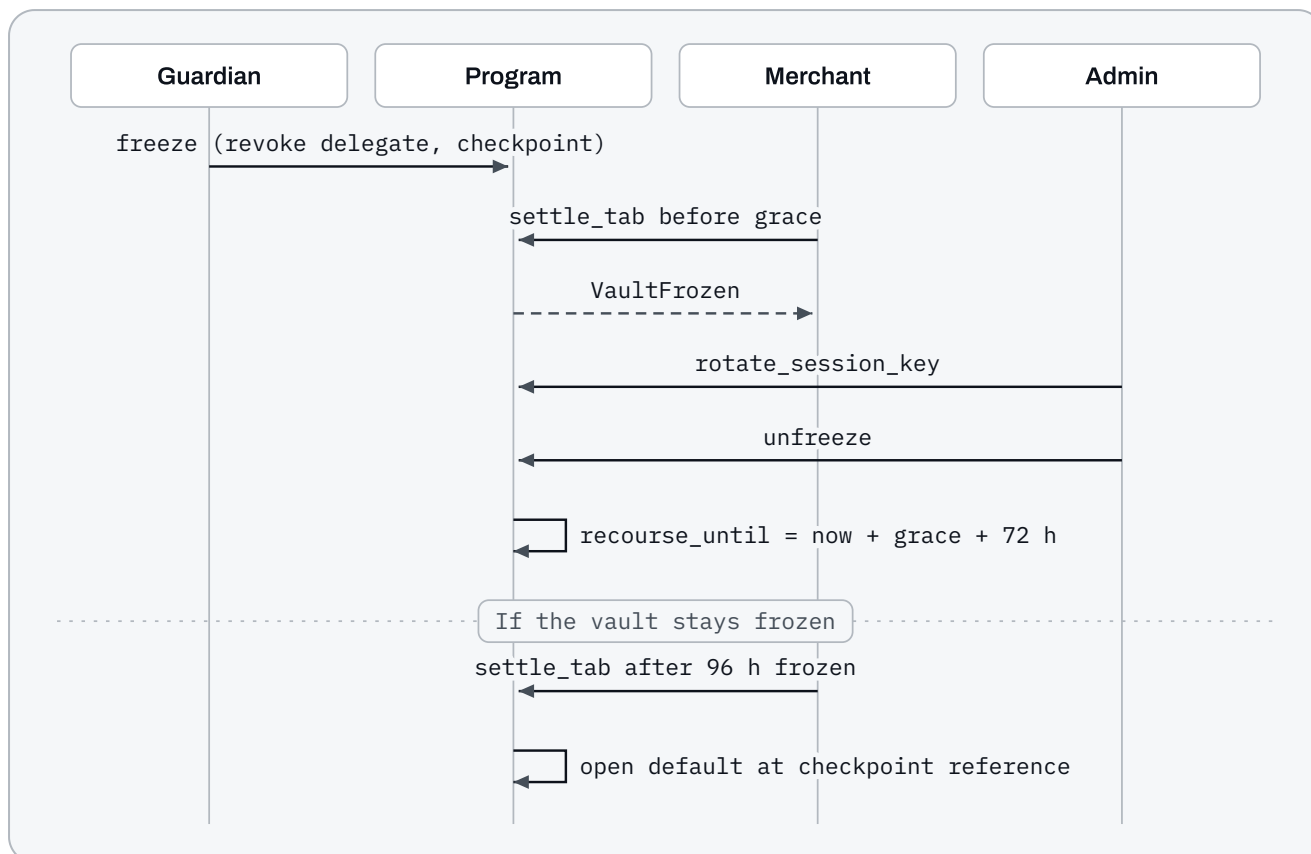


Figure 53. A guardian freeze, the grace that protects the operator, and the recourse that protects merchants.

Example, with the production grace of 345,600 seconds. The guardian freezes a vault at hour 0 after a spending spike, and a merchant's settlement at hour 10 is refused. The admin rotates the session key and unfreezes at hour 30, so `episode_frozen_seconds` becomes 108,000 and the recourse tail runs to hour 30 + 96 + 72 = 198. Had the vault stayed frozen, a settlement at hour 96 would have opened a default judged at the hour-0 checkpoint.

12.8 Service security

Each service holds only the keys and data its job requires and checks what it relies on. The API reads projections and relays signed bytes: compromising its process yields no key that moves a user's funds, and a wallet or SDK that verifies its transaction (§12.4, §12.5) signs nothing the API alters.

Control	Setting	Value
Request rate	<code>RATE_LIMIT_PER_MINUTE</code> per client, 60-second window, <code>/healthz</code> exempt	120 by default, 1–10,000; 429 with <code>Retry-After</code>
Beta intake rate	<code>BETA_APPLICATION_RATE_LIMIT_PER_HOUR</code> per client	10 by default
Client address	the socket address; the first <code>X-Forwarded-For</code> address only with <code>TRUST_PROXY=true</code>	<code>false</code> by default
CORS	<code>CORS_ORIGINS</code> , exact origins without paths or wildcards	a disallowed <code>Origin</code> is refused with 403; requests without <code>Origin</code> are allowed
Cookie	<code>bonded_session</code> , <code>HttpOnly</code> , <code>SameSite=Lax</code> , <code>Path=</code>	Secure in production; HMAC-SHA-256 tag with a <code>SESSION_SECRET</code> of at least 32 characters

Control	Setting	Value
Sessions and challenges	<code>SESSION_TTL_SECONDS</code> , <code>CHALLENGE_TTL_SECONDS</code>	3,600 s and 300 s by default
CSRF	<code>X-CSRF-Token</code> on every non- <code>GET</code> request after sign-in	32 random bytes, constant-time comparison
Request bodies	JSON only	at most 64 KiB, otherwise <code>413</code> ; a non-JSON type gets <code>415</code>
Response headers	every response	<code>X-Content-Type-Options: nosniff</code> , <code>Cache-Control: no-store</code> , <code>X-Request-Id</code>

CORS responses echo only an allowed origin with credentials and `Vary: Origin` , allow `GET` , `POST` , `DELETE` and `OPTIONS` with `Content-Type` , `X-CSRF-Token` , `X-Bonded-Fixture` and `Last-Event-ID` , cache preflights for 600 seconds and expose only `X-Request-Id` . Fixture variants and actors are refused in live mode, and the local faucet exists only in the local stack against a loopback RPC.

Logs carry no secrets. The API writes one JSON line per request with the request ID, route template and status, never wallet addresses, query strings, cookies, bodies or signatures, and logs errors by event name. In production the RPC adapter logs only read-method names, never endpoints or payloads; the deploy script redacts RPC URLs from errors, since provider URLs can embed keys; and the MCP CLI never echoes key bytes, file contents or transport errors.

Keys live only where they are used. The API loads no admin, session, merchant, guardian, sponsor or keeper key, and every fee payer is the signing user's wallet. The keeper key exists only in the worker, as `KEEPER_SECRET_KEY` , and signs only permissionless instructions, which pay accounts owned by the party that state names, and, as the buyback crank, the buyback (\$10.7). Guardian and sponsor keys stay in the operator's signing transport, the merchant key in the merchant host's signer, and the session key in the agent's signer; the MCP CLI reads it from the path its host configures and refuses symlinks and anything but an ordinary file, and the deploy script refuses the default Solana wallet.

The database is a projection and is not trusted for money. Amounts are `Decimal(20, 0)` with u64 range constraints, an indexer pass that sees a bond's recorded balance above its SPL balance refuses to commit, and every signer acting on projected state either re-checks it through its own RPC or relies on the program to refuse a stale action at simulation.

12.9 Upgrade authority and verified builds

The upgrade authority is the root of trust above the program: it can replace the code that enforces everything in this chapter. Bonded binds every other governed power to it and makes its custody a governance step that needs no code change.

At launch the deployer's key is the upgrade authority. The deploy script requires a dedicated deployer keypair, checks the RPC's genesis hash against the selected cluster's (devnet

`EtWTRABZaYq6iMfeYKouRu166VU2xqa1wcaWoxPkrZBG` , mainnet-beta `5eykt4UsFv8P8NJdTREpY1vzqKqZKvdpKuc147dw2N9d`), requires `--confirm-mainnet` for mainnet, and deploys through a persistent buffer with the deployer as upgrade authority.

The protocol authority is not a separate key. `initialize_protocol` , `update_protocol` , `configure_escrow_fee` , `configure_escrow_resolution` and `configure_buyback` each require the signer to be the `upgrade_authority_address` of the program's actual ProgramData account, verified at execution, and record the signer as `ProtocolConfig.authority` . There is no bootstrap key and no first-caller path. Bonded's governance moves this authority from the deployer to a Squads multisig [15] with a 7-day timelock, paired with verified builds. Because every governed instruction reads ProgramData at execution, transferring the loader's upgrade authority is the whole migration: from that transaction on, every upgrade and every protocol change needs the multisig's approval and waits out its timelock.

The protocol authority's own power is bounded by values it cannot change later. The maximum tab fee is fixed at initialization, at most 10,000 basis points, and the current fee cannot exceed it; the maximum escrow fee is fixed by the first `configure_escrow_fee` . Each line keeps the tab fee in force when it opened, through renewals, and each escrow the resolution duration in force when funded, so changes apply only to new lines and jobs. No governed instruction moves USDC: the `fees` PDA pays out only to the configured crank, inside a transaction that swaps all of it for \$BONDED and burns the result (§10.7). The production parameter files set a 10-basis-point tab fee, waived, under a 100-basis-point maximum, and a 25-basis-point escrow fee under a 100-basis-point maximum.

A verified build ties the deployed bytes to reviewed source. The program builds from locked Cargo sources with a pinned toolchain (Anchor 0.31.1, Solana platform-tools v1.52) and its program ID fixed at build time. After deployment the deploy script hashes the ProgramData bytes after the 45-byte header and stops before initialization unless the SHA-256 equals the hash of the locally built `.so` . Anyone can repeat the comparison by rebuilding the tagged source with the same toolchain, and under the multisig the same comparison applies to a proposed upgrade's buffer before approval.

13. Infrastructure and deployment

A Bonded deployment is one program on a Solana cluster and four services around it: the site, the API, the worker and a Postgres database. The program is deployed once from an operator's machine with the Solana toolchain; the services are Node and run on Railway. No service holds a user's key, and the only signing key any of them holds is the keeper's, in the worker alone.

13.1 What runs where

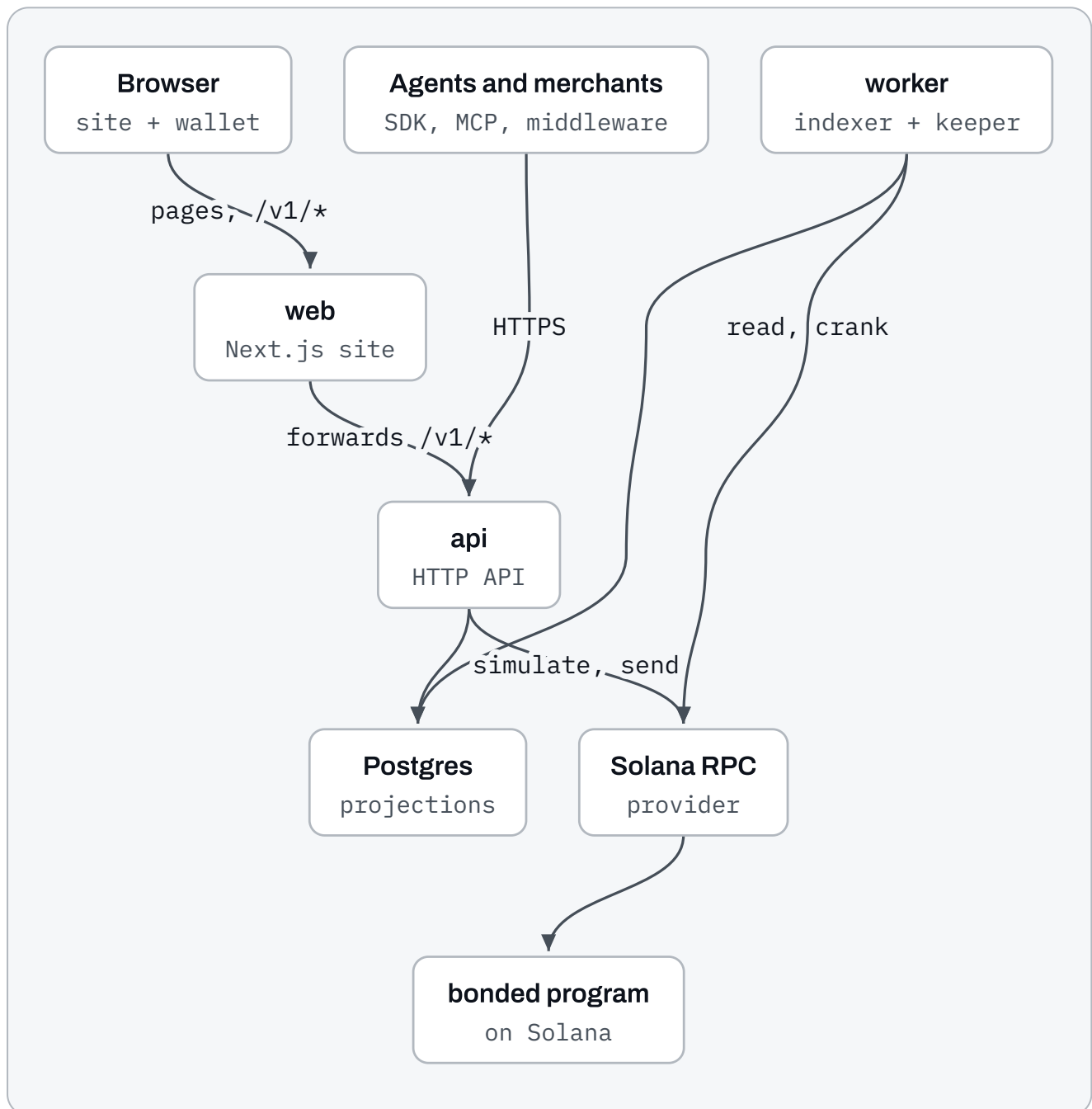


Figure 54. A deployment: four Railway services, the program on Solana, and the clients that use them.

Service	Source folder	Runs	Public domain	Holds
web	web/	the site: landing, app, docs, papers; forwards the browser's <code>/v1/*</code> to the API	yes	no key
api	backend/	the HTTP API: sign-in, reads, prepare and submit, live events	yes	the session secret
worker	backend/	the indexer and the keeper	no	the keeper key
Postgres	Railway template	the chain projections, prepared and submitted transactions, live events, beta applications	no (private)	data

The API and the worker share one codebase and one build, and differ only in their start command. Splitting them keeps the signing key out of the process that answers the internet, and lets the indexer run on its own cadence however many requests the API serves.

13.2 The services

The site

The site is a Next.js application. Its pages are rendered at build time where they can be (the landing, the docs, the papers) and in the browser where they act (the app). The browser never calls the API's domain: it calls `/v1/*` on the site's own origin, and a route on the site forwards each request to the API, passing the method, body, `Cookie`, `X-CSRF-Token`, `Origin` and `Last-Event-ID` headers through and returning `Set-Cookie` and the response unchanged. Server-sent events stream through the same route unbuffered.

Forwarding through the site's origin makes the session cookie first-party wherever the two services are hosted: it is set by, and sent to, the site's own domain. The site also serves the package tarballs that agent developers and merchants install, and the two papers as PDFs.

The API

The API is a Node HTTP server. In production it binds all interfaces on the port Railway injects, applies its Postgres migrations at start (`prisma migrate deploy` , which needs the private network and so can't run at build time), and serves:

- wallet sign-in with HMAC-authenticated session cookies (`Secure` , `HttpOnly` , `SameSite=Lax` , `Path=`) and a CSRF token per session;
- reads from the projections;
- prepare → submit for every action: it builds the transaction, simulates it, and after the wallet signs, accepts the signed bytes only if the prepared instructions are unchanged and in order, with nothing added but `ComputeBudget` instructions and trailing Lighthouse guards, the same blockhash, fee payer and signers, and no account made newly writable; it then simulates the signed bytes and relays them;
- live events over server-sent events, fed from committed database events with a 15-second heartbeat.

It holds no signing key. Cross-origin browser access is limited to `CORS_ORIGINS` , the site's exact origin; forwarded client addresses are used for rate limits only when `TRUST_PROXY` is set. The development faucet, fixture sign-in and local connection details are absent from the production profile.

The worker

The worker runs two loops over the same database.

- **The indexer** reads the program's accounts and transactions from RPC into Postgres. Idle, it costs two reads a cycle: the cluster clock, then the program's newest signature with a minimum context slot. Only when that signature has changed does it read the program's accounts and the new transactions, decode them and commit the projection. The cycle is `INDEXER_POLL_INTERVAL_MS`, 30 seconds by default; an API request that needs fresh state can ask the worker for a refresh through Postgres. Every real RPC read is logged as one JSON line naming the method, never the keyed URL.
- **The keeper** finds due work in the projection and executes it with the keeper key: default payouts after a claim window, escrow refunds after a missed deadline, releases after a dispute window, and refunds after a missed resolution deadline; as the configured crank it also runs the fee buyback (\$10.7). Every other crank is permissionless in the program, so the keeper's key authorizes nothing but paying the network fee and any rent; a job that runs twice is refused the second time by the program's own state checks.

The guardian worker is not part of the deployment: each operator runs its own, with its own thresholds and signing adapter, against the same API.

13.3 Deploying the program

The program is built and deployed from the operator's machine with Node 22, the Solana CLI 3.x and the SBF toolchain; Railway never needs Rust. One command builds, deploys, verifies and initializes:

```
npm run program:deploy -- --cluster devnet --deployer .deploy-keys/deployer.json --rpc-url "$DEVNET_RPC_URL"
npm run program:deploy -- --cluster mainnet-beta --confirm-mainnet --deployer .deploy-keys/deployer.json --rpc-url "$MAINNET_RPC_URL"
```

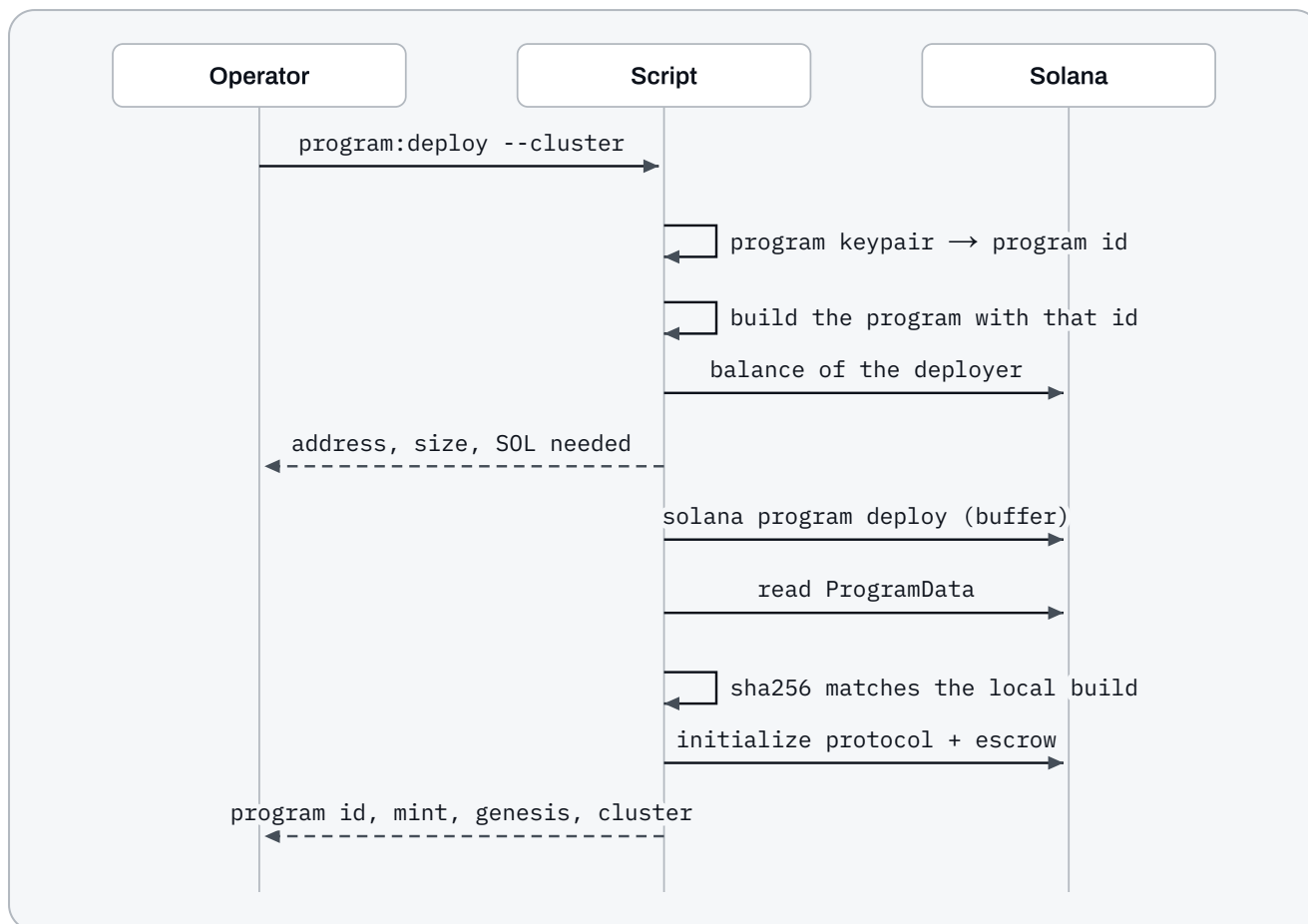


Figure 55. One deploy command: build with the program's own id, deploy, verify the bytes, initialize, print the public values.

- Keys.** The deployer is a keypair the operator chooses and funds; the script never reads a default CLI wallet. On first run it creates the program keypair and a fixed buffer keypair under `.deploy-keys/<cluster>/` (excluded from every upload), or uses ones supplied with `--program-keypair` and `--buffer-keypair`.
- Build.** The program is compiled with the id of that program keypair, so `declare_id!` and the deployed address agree, into `target/deploy/<cluster>/`.
- Preflight.** Before sending anything the script prints the deployer's full address, the program's size and the SOL it needs: about one times the rent of the program's bytes, plus the fee allowance of the parameter file (0.05 SOL). It stops if the balance is short; the Solana CLI repeats its own exact check.
- Deploy.** `solana program deploy --use-rpc` writes the program through the fixed buffer with an explicit compute-unit price (1,000 micro-lamports by default) and the deployer as upgrade authority. An interrupted deploy resumes from the same buffer, and the buffer's rent returns to the deployer when it completes. The first deploy sizes the program account to the executable exactly; the CLI extends it on a later, larger upgrade.
- Verify.** The script reads the deployed ProgramData and compares the SHA-256 of its executable bytes with the local build.
- Initialize.** It creates the protocol configuration from the cluster's parameter file (§14.2), the fee account, the escrow fee and the escrow resolution setting, and checks the fee account; with a \$BONDED mint in the parameter file it also configures the buyback (§10.7). Run again, it verifies every value and changes nothing; a mismatch stops it rather than govern an existing deployment with different values.

7. **Public values.** It prints the values the services need, `BONDED_PROGRAM_ID` , `USDC_MINT` , `SOLANA_GENESIS_HASH` , `SOLANA_CLUSTER` , `SERVICE_PROFILE` and `SERVICE_MODE` , as a block to paste into the API's variables. These are public; the same program id and genesis hash are the pins every SDK, middleware and MCP server is configured with.

A `--dry-run` performs steps 1 and 2 and prints the plan without contacting a cluster.

Authority. The deployer keeps the upgrade authority, and with it the protocol authority (§9.8, §12.9). Moving it to a Squads multisig is one `solana program set-upgrade-authority` transaction the operator sends after verifying the multisig's authority address; from then on every upgrade and every protocol change goes through the multisig.

13.4 Configuration

Each service reads its configuration from the environment. Values that cross services are Railway reference variables, so a domain or a database URL is written once.

Variable	Service	Value
<code>DATABASE_URL</code>	api, worker	<code>\${{Postgres.DATABASE_URL}}</code>
<code>SOLANA_CLUSTER</code>	api; worker references it	<code>devnet</code> or <code>mainnet-beta</code>
<code>SOLANA_RPC_URL</code>	api; worker references it	the RPC provider's HTTPS URL for that cluster (secret when keyed)
<code>BONDED_PROGRAM_ID</code> , <code>USDC_MINT</code> , <code>SOLANA_GENESIS_HASH</code>	api; worker references them	the public block from §13.3
<code>SESSION_SECRET</code>	api	32 random bytes, hex
<code>CORS_ORIGINS</code>	api	<code>https://\${{web.RAILWAY_PUBLIC_DOMAIN}}</code>
<code>TRUST_PROXY</code>	api	<code>true</code>
<code>KEEPER_SECRET_KEY</code>	worker	the keeper keypair, 64 bytes as a JSON array; its public key is also the buyback crank
<code>BUYBACK_LOOKUP_TABLES</code>	worker	the lookup table printed when the buyback is configured
<code>BUYBACK_INTERVAL_MS</code> , <code>BUYBACK_SLIPPAGE_BPS</code> , <code>BUYBACK_MAX_ACCOUNTS</code>	worker	300,000, 50, 48 (optional)
<code>INDEXER_POLL_INTERVAL_MS</code>	worker	30,000
<code>NODE_ENV</code> , <code>SERVICE_PROFILE</code> , <code>SERVICE_MODE</code>	api, worker	<code>production</code> , <code>production</code> , <code>live</code>
<code>BONDED_API_URL</code>	web	<code>https://\${{api.RAILWAY_PUBLIC_DOMAIN}}</code> : where the site forwards <code>/v1/*</code>
<code>NEXT_PUBLIC_BONDED_API_URL</code>	web	the same: the public API address the docs show
<code>SITE_URL</code>	web	<code>https://bonded.cash</code> , the site's own domain

`PORT` is injected by Railway and never set by hand. The keeper key is the only secret that signs; it is a fresh keypair with no other authority, funded with a little SOL for fees and rent, and its balance is watched. The session secret authenticates cookies. The RPC URL is secret only because providers put their key in it.

Service settings are entered in Railway's dashboard, which applies them to uploads from the CLI as well:

Setting	api	worker	web
Uploaded from	<code>backend/</code>	<code>backend/</code>	<code>web/</code>
Build	<code>npm run build</code>	<code>npm run build</code>	<code>npm run build</code>
Start	<code>npm start</code>	<code>npm run start:worker</code>	<code>npm start</code>
Health check	<code>/healthz</code>	—	<code>/healthz</code>
Restart	on failure, 3 retries	on failure, 3 retries	on failure, 3 retries

Railway builds with Railpack on Node 22.23.3 and npm 10.9.9; the backend pins them (`engines` , `packageManager` , `RAILPACK_NODE_VERSION`) and the web service sets `RAILPACK_NODE_VERSION` , and each and its lockfile installs cleanly with that npm. Each upload is filtered by the service's `.railwayignore` , which excludes keys, real environment files, dependencies, build output and native artifacts, and is checked by listing exactly what it uploads.

13.5 Checking a deployment

- `GET /healthz` on the API answers 200 with a status: `waiting` while the database is reachable but the program isn't deployed or initialized yet, `ok` once the indexed protocol is ready. A waiting deployment is an intended state, not an error.
- `GET /v1/protocol` reports the cluster, the mint and the voucher domain (program id and genesis hash) once indexed. The docs' service values read it live.
- Signing in on the site sets the session cookie on the site's own domain; an action from the app goes `prepare` → `wallet` → `submit` → `confirmed` and appears in the activity feed within one indexer cycle.
- The worker's logs show one line per RPC read with its method; an idle deployment reads twice per cycle.

One API replica serves the beta. Sign-in challenges and rate-limit counters live in the API process; the cookie itself is authenticated by the secret, and a restart asks users to sign in again.

13.6 Upgrades

A program upgrade is the same command with the same program keypair: the script builds, deploys through the buffer and verifies the bytes, and the initialization step confirms the configuration is unchanged. Because the protocol authority is read from the program's `ProgramData` at execution, an upgrade needs no configuration migration. The services redeploy independently of the program; each reads the program id and mint from its environment.

14. Parameters

Every number that governs Bonded's behaviour lives in one of three places. **Program constants** are compiled into the program: changing one is a program upgrade. **Protocol configuration** is written to the `ProtocolConfig` account when the deployment is initialized, and a few fields can later be changed by the protocol authority within ceilings the configuration itself sets. **Policy** belongs to each operator and each vault, chosen by the operator per account. This chapter lists all three with the values a deployment starts with.

14.1 Program constants

Constant	Value	Where it applies
<code>CLAIM_WINDOW</code>	259,200 s (72 h)	from a default event's opening to the first payout; part of every bond withdrawal's cooldown; the tail after a freeze episode
<code>DECREASE_DELAY</code>	259,200 s (72 h)	a line limit decrease, a line voucher-key rotation, and an operator or vault guardian rotation take effect this long after they are requested
<code>DISPUTE_WINDOW_SECONDS</code>	172,800 s (48 h)	after an escrow job is marked delivered, the time the payer has to dispute it
<code>MIN_FREEZE_GRACE_SECONDS</code>	345,600 s (96 h)	the shortest freeze grace a deployment may configure: 72 h to replace a guardian plus 24 h to respond
<code>COVERAGE_DEFAULT_BPS</code>	10,000 (100 %)	the minimum coverage the API gives a new operator unless it asks for another; <code>create_operator</code> itself takes an explicit value
<code>COVERAGE_FLOOR_BPS</code>	3,300 (33 %)	the lowest minimum coverage an operator may choose
<code>BOND_CAP</code>	25,000,000,000 base units (\$25,000)	the most one bond may hold
<code>TOTAL_CAP</code>	250,000,000,000 base units (\$250,000)	the most all bonds of a deployment may hold together
<code>VOUCHER_DOMAIN_PREFIX</code>	Bonded voucher v2\0 (18 bytes)	the prefix hashed with the program id and genesis hash into every voucher's 8-byte domain tag

Wire-format constants that follow from the program's layouts:

Quantity	Value
Voucher	96 bytes: 8 domain, 32 line, 8 cumulative, 8 nonce, 8 expiry, 32 usage digest; integers little-endian
Ed25519 proof instruction for one voucher	208 bytes: 16-byte offsets header, 64-byte signature, 32-byte key, 96-byte message
Escrow quote message	169 bytes, in a 281-byte proof instruction
Approval message	128 bytes, version 2, bound to the vault's policy revision
Legacy transaction limit	1,232 bytes (Solana)
USDC	6 decimals; amounts are integer base units throughout

14.2 Protocol configuration

Written once by the initialization step (§13.3) from the deployment's parameter file. Devnet and mainnet-beta start from the same values; only the mint differs.

Field	Initial value	Changeable after initialization
<code>usdc_mint</code>	mainnet-beta EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v ; devnet 4zMMC9srt5Ri5X14GAgXhaHii3GnPAEERYPJgZJDncDU	no
<code>tab_fee_bps</code>	10	yes, by <code>update_protocol</code> , up to <code>maximum_tab_fee_bps</code>
<code>tab_fee_waived</code>	true (the beta waiver)	yes, by <code>update_protocol</code>
<code>maximum_tab_fee_bps</code>	100	no
<code>escrow_fee_bps</code>	25	yes, by <code>configure_escrow_fee</code> , up to its maximum
<code>maximum_escrow_fee_bps</code>	100	fixed once set
<code>escrow_resolution_seconds</code>	259,200 (72 h)	yes, by <code>configure_escrow_resolution</code> ; applies to jobs funded afterwards
<code>withdrawal_margin_seconds</code>	86,400 (24 h)	no
<code>freeze_grace_seconds</code>	345,600 (96 h)	no
<code>voucher_domain</code>	derived from the program id and the cluster's genesis hash	no
<code>authority</code>	the program's upgrade authority, read live from its ProgramData account	follows the upgrade authority
<code>fee_treasury</code>	the program-owned USDC account at seeds <code>[b"fees"]</code>	no
<code>buyback_token_mint</code>	unset until the \$BONDED mint is configured	yes, by <code>configure_buyback</code>
<code>buyback_crank</code>	the worker's keeper key	yes, by <code>configure_buyback</code>
<code>buyback_minimum_usdc</code>	10,000,000 (10 USDC)	yes, by <code>configure_buyback</code>

Two fee rules hold for every value the authority can set:

$$\text{tab_fee_bps} \leq \text{maximum_tab_fee_bps} \leq 10\,000$$

$$\text{escrow_fee_bps} \leq \text{maximum_escrow_fee_bps} \leq 10\,000$$

`maximum_tab_fee_bps` fixed at initialization, so no later update can raise the tab fee past it

`maximum_escrow_fee_bps` fixed the first time the escrow fee is configured

Each credit line copies the tab fee rate in force when it is opened (0 while the waiver is on) and keeps it for its life, so an update never changes the terms of a line already open. Each escrow job records its fee and its resolve duration when it is funded.

14.3 Operator and vault policy

Chosen by the operator, per account, within the constants above.

Parameter	Set by	Bounds	Meaning
<code>minimum_coverage_bps</code>	<code>create_operator</code>	3,300 – 65,535	the coverage every line opening and bond withdrawal must keep
guardian (operator)	<code>create_operator</code> ; <code>rotate_operator_guardian</code> (72 h)	any key or none	may freeze every vault of the operator
<code>session_key</code>	<code>create_vault</code> , <code>rotate_session_key</code>	one key	the only key that spends from the vault
<code>float_cap</code>	<code>create_vault</code> , <code>set_policy</code>	$\leq$ <code>window_budget</code>	the SPL Token allowance the session key holds
<code>window_seconds</code>	<code>create_vault</code> , <code>set_policy</code>	> 0	the length of the spending window
<code>window_budget</code>	<code>create_vault</code> , <code>set_policy</code>	$\geq$ <code>float_cap</code>	what float spending, direct payments and escrow may total per window
cosigner, <code>cosign_threshold</code>	<code>create_vault</code> , <code>set_policy</code>	Ed25519 key or passkey credential	payments above the threshold need an approval
guardian (vault)	<code>create_vault</code> ; <code>rotate_vault_guardian</code> (72 h)	any key or none	may freeze this vault
merchant rule	<code>set_merchant_rule</code>	per merchant	per-transaction cap, window budget, quote required
line limit, expiry	<code>open_line</code> , <code>resize_line</code> , <code>renew_line</code>	coverage at the minimum; expiry after now	what the agent may owe one merchant, and until when

14.4 Service configuration

The API and the worker read their own settings from the environment (§13.4). The ones that shape behaviour a user sees:

Setting	Effect
SESSION_TTL_SECONDS	how long a signed-in session lasts
CHALLENGE_TTL_SECONDS	how long a sign-in challenge may be signed
RATE_LIMIT_PER_MINUTE	requests per client per minute before RATE_LIMITED
CORS_ORIGINS	the site origins that may call the API from a browser; a sign-in message names the request's own allowed origin, or the first one when the request carries none
indexer cadence	how often the worker reads the chain, and so how quickly reads follow a confirmed transaction
TRANSACTION_COMPUTE_UNIT_PRICE_MICROLAMPORTS	the priority fee the API attaches to the transactions it prepares (1,000 by default)
BUYBACK_INTERVAL_MS , BUYBACK_SLIPPAGE_BPS , BUYBACK_MAX_ACCOUNTS	how often the crank looks at the fee account (5 minutes), the slippage its quote allows (50 bps), and the route's account cap (48)

None of these changes what the program accepts: they bound how the services around it behave.

14.5 Derived quantities

A few values follow from the parameters above and appear throughout the paper.

$$\text{coverage_bps} = \lfloor (\text{bond.balance} - \text{pending withdrawal}) \times 10\,000 \div \Sigma \text{line.limit} \rfloor$$

$$\text{unlock_at} = \max(\text{now}, \text{longest_line_expiry}) + \text{CLAIM_WINDOW} + \text{withdrawal_margin_seconds}$$

$$\text{tab_fee} = \lfloor (\text{voucher.cumulative} - \text{floor}) \times \text{line.tab_fee_bps} \div 10\,000 \rfloor$$

$$\text{escrow_fee} = \lfloor \text{quote.amount} \times \text{escrow_fee_bps} \div 10\,000 \rfloor$$

$$\text{payout} = \min(\text{eligible}, \lfloor \text{bond.balance} \times \text{eligible} \div \Sigma \text{eligible} \rfloor \text{ when } \Sigma \text{eligible} > \text{bond.balance})$$

longest_line_expiry	the latest expiry of any line the bond has backed
eligible	min(voucher.cumulative – floor, line.limit) for each claim
floor	max(settled_cumulative, retired_cumulative), the line's settled total or the voucher total an earlier paid claim retired

With the initial configuration, an operator whose longest line expires in 30 days can withdraw its bond 30 days + 72 h + 24 h after asking at the earliest; a \$100.00 settlement on a line opened after the waiver ends pays a \$0.10 fee; a \$340.00 escrow job debits the vault \$340.85.

Appendix A. Instruction reference

This appendix lists all 47 instructions of the `bonded` program, grouped by area as in §9.4. For each it gives the Borsh arguments, the accounts in the order the instruction expects them, the checks in the order they run, and the effect.

A.1 Conventions and summary

Account flags: **s** signer, **w** writable, **init** created by the instruction (fails if it exists), **init?** created when missing, **close** closed by the instruction, **opt** optional. An absent optional account is passed as the program ID. Seeds are written as `[b"prefix", key, ...]`; integers inside seeds are little endian.

Every account listed is validated by Anchor before the handler runs: PDA seeds and bump, owner and discriminator, `has_one` relations, fixed addresses and signer flags. A failure there raises Anchor's own constraint error, not a Bonded code. The checks below list the explicit constraints that carry a Bonded error, then the handler's checks. "Collateral unlocked" means `operator.frozen_vault_count == 0` and `now > operator.recourse_lock_until` (else `FrozenRecourseActive`). "Coverage" means `available × 10 000 ≥ reserved × minimum_coverage_bps` (else `CoverageTooLow`). "Reconcile" is the window step shared by policy instructions:

$$\begin{aligned} \text{periods} &= \lfloor (\text{now} - \text{window_started_at}) \div \text{window_seconds} \rfloor \\ \text{window_started_at} &+= \text{periods} \times \text{window_seconds}; \text{window_spent} = 0 \text{ when} \\ &\quad \text{periods} > 0 \\ \text{window_spent} &+= \text{last_approved} - \text{delegated_amount} \\ \text{last_approved} &= \text{delegated_amount} \end{aligned}$$

reconcile fails with `InvalidConfiguration` if `now < window_started_at` or `delegated_amount > last_approved`

Instruction data is the 8-byte discriminator, the first 8 bytes of SHA-256 of `global:<name>`, followed by the arguments.

Instruction	Discriminator	Data bytes	Signer
<code>initialize_protocol</code>	<code>bce9fc6a8692ca5b</code>	53	upgrade authority
<code>update_protocol</code>	<code>ce19da726d294aad</code>	11	upgrade authority
<code>configure_escrow_fee</code>	<code>37da52f7688371d8</code>	12	upgrade authority
<code>configure_escrow_resolution</code>	<code>a63f609565786c04</code>	12	upgrade authority
<code>configure_buyback</code>	<code>ff699821c0dd4a18</code>	48	upgrade authority
<code>claim_buyback</code>	<code>c7fff3ecc7acc98d</code>	24	buyback crank
<code>burn_buyback</code>	<code>ed2201f3acdcf937</code>	8	buyback crank

Instruction	Discriminator	Data bytes	Signer
create_operator	9128ee4bb5fc3b0b	42	admin
post_bond	a897ca77a33a93f7	16	admin
open_line	636882c680221bd0	24	admin
resize_line	315323428eae4966	16	admin
renew_line	f311bd16f0754fc5	16	admin
rotate_line_key	9421e8c83077cd3f	40	admin
attest_line_voucher	fa57d99dc12860af	10	merchant
settle_tab	94d00a8c35043961	18	merchant
file_claim	bbfe280d92dfe661	10	merchant
execute_default_payout	fb04d6a7b541d894	8	none
finalize_empty_default	9919f19461c2298a	8	none
close_claim	2ab1a523d5b3d313	8	merchant
close_default_placeholder	5ad9c064ea73fdd2	8	merchant
request_withdrawal	fb5579cd38c90cb1	16	admin
finalize_withdrawal	b257ce44c9baa4e8	8	admin
create_vault	1dedf7d0c1523687	164	admin
set_policy	28850c9debca0284	132	admin
rotate_session_key	8d1c89eadd8c0716	40	admin
withdraw	b712469c946da122	16	admin
set_merchant_rule	df76fa36f030a08c	25	admin
refill_float	4c161ce5f9b5d637	8	none
pay	7712d841c0757adc	89 or 91	session key, payer
approve_payment	157bc38b6b8d22bb	98	payer
approve_payment_passkey	754e08e478d1aee2	106 + both byte vectors	payer
set_passkey_credential	247e93281dd57128	137	admin
close_approval	8632e5cf43a194cb	8	any
pay_escrow	cdd27b87710061f4	82	session key
mark_delivered	f076bc8e40556b12	40	merchant
dispute	d85c8092ca558749	8	session key or admin
accept	419646d885066b04	8	session key or admin
release_after_window	c3d1827cac1405f9	8	any

Instruction	Discriminator	Data bytes	Signer
resolve	f696ecce6c3f3a0a	9	arbiter
refund_on_timeout	9ae2da9966b9ca02	8	any
refund_after_resolution_deadline	bce76df53ba37bd4	8	any
freeze	ff5bcf54fbc2fe3f	8	admin or a guardian
unfreeze	85a044fd50e8daf7	8	admin
fund_freeze_reserve	f605ab8199982e47	16	admin
close_freeze_checkpoint	39b4efaec81d2396	8	none
rotate_operator_guardian	18beed245dc0262a	40	admin
rotate_vault_guardian	66c36bbb4695921c	40	admin

A.2 Protocol

The five governed instructions take `program` (the Bonded program account) and `program_data` (its loader-v3 ProgramData). Two constraints apply to both, each failing with `Unauthorized`: the program's ProgramData address equals `program_data`, and `program_data.upgrade_authority_address` equals the signer.

`initialize_protocol`

Arguments: `tab_fee_bps`: u16, `tab_fee_waived`: bool, `withdrawal_margin_seconds`: u32, `maximum_tab_fee_bps`: u16, `freeze_grace_seconds`: u32, `cluster_genesis_hash`: [u8; 32].

Accounts: `authority` (s,w) · `protocol` (w,init, [b"protocol"]) · `mint` · `fee_treasury` (w,init token account at [b"fees"], mint `mint`, `authority` `protocol`) · `program` · `program_data` · `token_program` · `system_program`.

Checks: ProgramData constraints; `maximum_tab_fee_bps` $\leq$ 10 000, `tab_fee_bps` $\leq$ `maximum_tab_fee_bps`, `withdrawal_margin_seconds` $>$ 0 and `freeze_grace_seconds` $\geq$ 345 600 (`InvalidConfiguration`); `mint.decimals` == 6 (`InvalidMint`).

Effect: writes every field; `authority` is the signer; `voucher_domain` = SHA-256("Bonded voucher v2\0" || `program ID` || `genesis`)[0..8]; the escrow options stay empty. No event.

`update_protocol`

Arguments: `tab_fee_bps`: u16, `tab_fee_waived`: bool. Accounts: `authority` (s) · `protocol` (w, [b"protocol"]) · `program` · `program_data`.

Checks: ProgramData constraints; `tab_fee_bps` $\leq$ `maximum_tab_fee_bps` (`InvalidConfiguration`).

Effect: sets the fee and waiver for lines opened from now on; `authority` becomes the signer; emits `ProtocolUpdated`.

`configure_escrow_fee`

Arguments: `fee_bps`: u16, `maximum_fee_bps`: u16. Accounts: as `update_protocol`.

Checks: ProgramData constraints; `maximum_fee_bps` $\leq$ 10 000, `fee_bps` $\leq$ `maximum_fee_bps`, and the stored maximum is unset or equal to `maximum_fee_bps` (`InvalidConfiguration`).

Effect: `escrow_fee_bps = Some(fee_bps)` , `maximum_escrow_fee_bps = Some(maximum_fee_bps)` ;
`authority` synced; emits `ProtocolUpdated` .

configure_escrow_resolution

Arguments: `resolve_seconds: u32` . Accounts: `as update_protocol` .

Checks: `ProgramData` constraints; `resolve_seconds > 0` (`InvalidConfiguration`).

Effect: `escrow_resolve_seconds = Some(resolve_seconds)` for escrows funded from now on; `authority` synced; emits `ProtocolUpdated` .

configure_buyback

Arguments: `minimum_usdc: u64` , `crank: Pubkey` . Accounts: `protocol (w, [b"protocol"])` ·
`authority (s)` · `program` · `program_data` · `token_mint` (the \$BONDED mint, owned by `token_program`)
· `token_program` (SPL Token or Token-2022).

Checks: `ProgramData` constraints; `minimum_usdc > 0` and a non-zero `crank` (`InvalidConfiguration`); no
buyback pending (`BuybackPending`); the mint carries only metadata-pointer and token-metadata extensions
(`BuybackInvalidTokenMint`).

Effect: stores `buyback_token_mint` , `buyback_crank` and `buyback_minimum_usdc` ; `authority` synced;
emits `BuybackConfigured` .

claim_buyback

Arguments: `amount: u64` , `minimum_output: u64` . Accounts: `protocol (w)` · `fee_treasury (w,`
`[b"fees"])` · `crank (s)` · `crank_usdc (w, the crank's USDC associated account)` · `crank_tokens` (the
crank's \$BONDED associated account) · `mint (USDC)` · `token_program` · `instructions (sysvar)`.

Checks: a mint and crank are configured (`BuybackNotConfigured`); the signer is the crank (`Unauthorized`);
nothing pending (`BuybackPending`); `amount ≥ buyback_minimum_usdc` , `minimum_output > 0`
(`InvalidAmount`); `amount ≤ the fee account` (`BuybackInsufficientFees`); both crank accounts are the
canonical associated accounts (`BuybackInvalidTokenMint`); the instruction is top-level and a matching
`burn_buyback` follows later in the transaction (`BuybackMissingBurn`).

Effect: `TransferChecked` of `amount` from the fee account to `crank_usdc` , signed by the protocol PDA;
records `pending_buyback` (amount, minimum output, the two balances before, this instruction's index).

burn_buyback

Arguments: none. Accounts: `protocol (w)` · `crank (s)` · `crank_usdc` · `token_mint (w)` · `crank_tokens`
(w) · `token_program` (the mint's owner) · `instructions (sysvar)`.

Checks: the signer is the crank (`Unauthorized`); the mint is the configured one
(`BuybackInvalidTokenMint`); the pending claim sits earlier in this transaction with the same amount,
minimum and accounts (`BuybackMissingClaim`); `crank_usdc` holds no more than before the claim
(`BuybackUsdcNotSpent`); the \$BONDED gain is at least the minimum (`BuybackOutputTooLow`).

Effect: burns the whole gain from `crank_tokens` under the mint's token program, clears `pending_buyback` ,
and emits `BuybackBurned` .

A.3 Operator, bond and lines

create_operator

Arguments: guardian: Pubkey (zero for none), minimum_coverage_bps: u16 .

Accounts: admin (s,w) · protocol · usdc_mint (address protocol.usdc_mint) · operator (w,init, [b"operator", admin]) · bond (w,init, [b"bond", operator]) · bond_token (w,init?, ATA of bond for the mint) · token_program · associated_token_program · system_program .

Checks: minimum_coverage_bps $\geq$ 3 300 (CoverageTooLow).

Effect: Operator with admin , guardian , created_at = now ; Bond with its operator, token account and minimum; emits CoreEvent 1.

post_bond

Arguments: amount: u64 .Accounts: admin (s) · operator ([b"operator", admin] , has_one admin) · protocol (w) · bond (w, [b"bond", operator]) · mint · source (w, token account of the mint, authority admin) · bond_token (w, address bond.token_account , authority bond) · token_program .

Checks: amount > 0 (InvalidAmount); bond.balance + amount $\leq$ 25 000 000 000 and total_bond_balance + amount $\leq$ 250 000 000 000 (BetaCapExceeded).

Effect: TransferChecked from source , signed by the admin; raises bond.balance and total_bond_balance ; emits CoreEvent 4.

open_line

Arguments: limit: u64 , expires_at: i64 .Accounts: admin (s,w) · operator · protocol · bond (w) · vault ([b"vault", operator, agent_id] , has_one operator) · merchant (any address; does not sign) · line (w,init, [b"line", bond, merchant]) · system_program .

Checks: vault not frozen (VaultFrozen); no active default (DefaultActive); limit > 0 and expires_at > now (InvalidAmount); coverage with available = balance - pending withdrawal and reserved = reserved_lines + limit .

Effect: reserved_lines += limit ; longest_line_expiry = max(longest_line_expiry, expires_at) ; a pending withdrawal's unlock_at rises to at least max(longest_line_expiry, now) + 259 200 + withdrawal_margin_seconds . The line records the vault, the merchant, voucher_session_key = vault.policy.session_key , the limit and expiry, opened_default_sequence = bond.default_sequence , opened_at = now , freeze_generation_at_open = vault.freeze_generation and tab_fee_bps (0 when waived). Emits CoreEvent 5.

resize_line

Arguments: limit: u64 .Accounts: admin (s) · operator · bond (w) · line (w, [b"line", bond, line.merchant] , has_one bond) · vault (has_one operator ; line.vault == vault , else Unauthorized).

Checks: no active default (DefaultActive). A decrease that repeats the pending value: collateral unlocked, then now $\geq$ pending_limit.effective_at (DecreaseDelayNotReached). Any other decrease: none. An increase: vault not frozen (VaultFrozen); for any non-decrease, the line is in the current epoch and unexpired (LineExpired), and coverage holds with reserved - line.limit + limit .

Effect: a new decrease is scheduled at `now + 259 200 ( CoreEvent 6)`; a mature repeated decrease or a non-decrease applies, adjusts `reserved_lines` and clears the pending decrease (`CoreEvent 7`). Resizing to the current limit cancels a pending decrease.

renew_line

Arguments: `expires_at: i64 .Accounts: admin (s) · operator · protocol · bond (w) · vault · line (w, has_one bond , has_one vault )`.

Checks: `vault not frozen ( VaultFrozen )`; no active default (`DefaultActive`); collateral unlocked; `expires_at > now and expires_at ≥ line.expires_at ( InvalidAmount )`; coverage with the current `reserved_lines` .

Effect: raises `longest_line_expiry` and a pending withdrawal's unlock as `open_line` does; sets `expires_at` and `opened_default_sequence = bond.default_sequence` . Opening time, first freeze generation, fee, limit, keys and replay floors are kept. Emits `CoreEvent 17`.

rotate_line_key

Arguments: `session_key: Pubkey .Accounts: admin (s) · operator · bond · line (w, has_one bond )`.

Checks: `session_key` is not zero (`InvalidConfiguration`); a pending key equal to `session_key` before its `effective_at ( KeyRotationDelayNotReached )`; when scheduling, `session_key` differs from the current key (`InvalidConfiguration`).

Effect: a matured pending key is first materialized (`previous_voucher_key = voucher_session_key , rotated_at = effective_at , voucher_session_key = pending`). If `session_key` was that pending key, the rotation is applied (`CoreEvent 19`); otherwise `session_key` is scheduled at `now + 259 200 ( CoreEvent 18)`. No freeze, recourse or default lock applies.

attest_line_voucher

Arguments: `ed25519_ix_index: u16 .Accounts: merchant (s,w) · protocol · line ( [b"line", line.bond, merchant] , has_one merchant ) · voucher_attestation (w, init?, [b"voucher-attestation", line, line.voucher_session_key, pending effective_at] ) · instructions (sysvar) · system_program .`

Checks: a key rotation is pending (`InvalidConfiguration`); `now < effective_at ( AttestationNoticeClosed )`; the voucher proof (as in `settle_tab`); signer equals `line.voucher_session_key ( InvalidPrecompile )`; `0 < expiry ≤ line.expires_at ( VoucherExpired )`; nonce above `maximum_nonce` and cumulative above `max(settled_cumulative, retired_cumulative) ( VoucherReplay )`; cumulative minus that floor $\leq$ `limit ( LineLimitExceeded )`; an existing record is not lowered in nonce or amount (`VoucherReplay`).

Effect: records the outgoing key, `valid_from = line.rotated_at` , the switch time, the voucher's nonce and cumulative amount and `attested_at` . No tokens move. Emits `CoreEvent 26`.

A.4 Settlement and defaults

settle_tab

Arguments: `default_sequence: u64 (the bond's default_sequence + 1 )`, `ed25519_ix_index: u16 (the proof's index, always the current index – 1)`.

Accounts: `merchant (s,w) · protocol · operator (w, [b"operator", operator.admin]) · vault (has_one operator) · vault_token (w, address vault.token_account, authority vault) · bond (w, has_one operator) · line (w, [b"line", bond, merchant], has_one bond, vault, merchant) · default_event (w, init?, [b"default", bond, default_sequence], paid by the merchant) · merchant_token (w, authority merchant) · fee_treasury (w, address protocol.fee_treasury) · mint · instructions · token_program · system_program · freeze_checkpoint (the line's reference checkpoint) · optional remaining account: the VoucherAttestation of an outgoing key.`

Checks, in order:

1. Reference checkpoint: an empty account must be the PDA for generation `max(freeze_generation_at_open + 1, freeze_episode_start_generation)`; a filled one must be a program-owned checkpoint of this vault at its own PDA, from that generation up to the current one, in the current episode (`InvalidFreezeCheckpoint`). It yields a reference time when it froze no later than now and the vault is frozen or within `recourse_until`.
2. Proof: the current instruction is Bonded's top-level instruction and the proof sits at index - 1, is the Ed25519 program with no accounts, 208 bytes and the canonical header (`InvalidPrecompile`); the message has the stored domain tag and names this line (`InvalidVoucher`).
3. Key: on an unfrozen vault, the signer is the key effective now, or an attested outgoing key within its nonce and amount ceilings, and the voucher is valid now; otherwise the signer must be authorized at the freeze reference, which becomes the validation time. A frozen vault with no reference fails (`VaultFrozen`); an unauthorized signer fails (`InvalidPrecompile`); a malformed attestation fails (`InvalidVoucherAttestation`).
4. Frozen vault: accumulated frozen seconds in the episode $\geq$ `freeze_grace_seconds` (`VaultFrozen`).
5. No active default (`DefaultActive`); `default_sequence == bond.default_sequence + 1` (`InvalidDefault`).
6. The line is in the current epoch and the validation time precedes `line.expires_at` (`LineExpired`).
7. Validation time $<$ `voucher.expiry` $\leq$ `line.expires_at` (`VoucherExpired`).
8. Nonce and cumulative both advance past the line floors (`VoucherReplay`).
9. On an unfrozen vault with collateral unlocked, a matured pending decrease is applied first.
10. $\text{delta} = \text{cumulative} - \max(\text{settled_cumulative}, \text{retired_cumulative}) \leq \text{limit}$ (`LineLimitExceeded`).

Effect: if the vault is frozen or `vault_token.amount < delta`, a default opens. The bond's `default_sequence` and `active_default_sequence` take the new sequence, a pending withdrawal is cleared, the event records `opened_at = now`, `eligibility_at`, the merchant as rent payer, `collateral_locked_at_open` and `window_end = now + 259 200`, `operator.defaults_count` rises, and `CoreEvent 9` is emitted; no tokens move. Otherwise `fee = [delta × line.tab_fee_bps ÷ 10 000]`; `delta - fee` goes to the merchant and `fee` to the fee account, both signed by the vault PDA; `settled_cumulative` and `maximum_nonce` advance; `operator.tabs_settled` rises; `CoreEvent 8` is emitted; the placeholder `default_event` closes to the merchant.

file_claim

Arguments: `ed25519_ix_index: u16`.

Accounts: `merchant (s,w) · protocol · bond ([b"bond", bond.operator]) · vault ([b"vault", bond.operator, agent_id] ; vault.operator == bond.operator ,else Unauthorized) · line ([b"line", bond, merchant] , has_one bond , merchant , vault) · default_event (w, [b"default", bond, sequence] , has_one bond) · claim (w,init?, [b"claim", default_event, merchant]) · instructions · system_program · freeze_checkpoint · activity_checkpoint (a checkpoint, or the program ID when no freeze applies) · optional remaining VoucherAttestation .`

Checks, in order:

1. Activity witness, unless it is the program ID: a program-owned checkpoint of this vault at its PDA, generated after the line opened and no later than the current generation, frozen at or before the default opened, with `defaults_opened_at_freeze < sequence` , and still frozen or with `recourse_until ≥ opened_at` (`InvalidFreezeCheckpoint`).
2. With a witness, `freeze_checkpoint` is validated within the witness's episode, with a generation no later than the witness's and `defaults_opened_at_freeze < sequence` (`InvalidFreezeCheckpoint`); its freeze time, if not after the opening, is the freeze reference.
3. `opened_at > 0` , `now < window_end` , no payout snapshot taken (`ClaimWindowClosed`).
4. The event is the bond's active default, and the line belongs to the epoch just before it (`InvalidDefault`).
5. The voucher proof (`InvalidPrecompile` , `InvalidVoucher`).
6. The signer is authorized at the opening time or at the freeze reference (`InvalidPrecompile` ; `InvalidVoucherAttestation`).
7. The voucher and line are unexpired at the reference that authorized it (`VoucherExpired`).
8. Nonce and cumulative advance past the line floors (`VoucherReplay`).
9. A refiling: the claim is unpaid and both nonce and cumulative exceed the filed voucher (`VoucherReplay`).

Effect: a first filing raises `claim_count` and snapshots `settled_snapshot` (the line floor) and `line_limit_snapshot` (the stored limit when a freeze reference applies or collateral was locked at opening, otherwise the limit effective at the reference). Every filing stores the voucher amount, nonce and expiry, sets `eligible_amount = min(max(cumulative - settled_snapshot, 0), line_limit_snapshot)` and `filed_at` , replaces its share of `claims_total` , and emits `ClaimFiled` and `CoreEvent 10`.

execute_default_payout

Arguments: none. Accounts: `protocol (w) · bond (w) · bond_token (w, address bond.token_account , authority bond) · default_event (w, has_one bond) · claim (w,close to merchant , [b"claim", default_event, claim.merchant] , has_one default_event , line) · line (w, [b"line", bond, claim.merchant]) · merchant (w, address claim.merchant ; does not sign) · merchant_token (w, owned by claim.merchant ,else Unauthorized) · mint · token_program .`

Checks: `now ≥ window_end` and `opened_at > 0` (`ClaimWindowOpen`); an already paid claim transfers nothing; the event is active and unpaid (`InvalidDefault`); `eligible ≤ eligible total` (`InvalidAmount`).

Effect: the first payout sets `payout_pool = bond.balance` and `eligible_total_snapshot = claims_total` . The claim receives its full eligibility when the total fits the pool, else `[pool × eligible ÷ total]` , from the bond token account signed by the bond PDA. `bond.balance` and `total_bond_balance` fall; the claim is marked paid; the line's `retired_cumulative` and `maximum_nonce` rise to the claimed voucher; `CoreEvent 11` is emitted. After the last claim the event is marked paid, the bond's active default clears and `CoreEvent 12` carries the total paid. The claim closes to the merchant.

finalize_empty_default

Arguments: none. Accounts: `bond (w) · default_event (w, has_one bond)`.

Checks: `opened_at > 0` and `now ≥ window_end (ClaimWindowOpen)`; `claim_count == 0` (`InvalidDefault`); an already finalized event is a no-op; the event is active (`InvalidDefault`).

Effect: records the pool snapshot, marks the event paid, clears the bond's active default; emits `CoreEvent 12` with amount 0.

close_claim

Arguments: none. Accounts: `merchant (s,w) · default_event · claim (w, close to merchant, has_one default_event, merchant, line; must be paid, else InvalidDefault) · line (maximum_nonce ≥ claim.voucher_nonce and retired_cumulative ≥ claim.voucher_amount, else VoucherReplay)`.

Effect: closes a retained paid claim; rent to the merchant. No event.

close_default_placeholder

Arguments: none. Accounts: `merchant (s,w) · bond · default_event (w, close to merchant, has_one bond; opened_at == 0 and rent_payer == merchant, else InvalidDefault)`.

Effect: closes a retained placeholder; rent to the merchant. No event.

request_withdrawal

Arguments: `amount: u64`. Accounts: `admin (s) · operator · protocol · bond (w)`.

Checks: collateral unlocked; no active default (`DefaultActive`); no pending request (`WithdrawalExists`); `0 < amount ≤ balance` (`InvalidAmount`); coverage with `balance - amount`.

Effect: stores `{amount, requested_at = now, unlock_at = max(now, longest_line_expiry) + 259 200 + withdrawal_margin_seconds}`; emits `CoreEvent 13`.

finalize_withdrawal

Arguments: none. Accounts: `admin (s) · operator · protocol (w) · bond (w) · bond_token (w, authority bond) · destination (w, token account of the mint) · mint · token_program`.

Checks: collateral unlocked; no active default (`DefaultActive`); a request exists (`NoWithdrawal`); `now ≥ unlock_at` (`WithdrawalCoolingDown`); coverage with `balance - amount`.

Effect: `TransferChecked` to `destination` signed by the bond PDA; `balance` and `total_bond_balance` fall; the request clears; emits `CoreEvent 14`.

A.5 Vault and payments

`PolicyArgs` is `session_key: Pubkey, float_cap: u64, window_seconds: u32, window_budget: u64, cosigner: Pubkey` (zero for none, or a `PasskeyCredential PDA`), `cosign_threshold: u64, guardian: Pubkey` (zero for none). A valid policy has a nonzero session key, `window_seconds > 0` and `float_cap ≤ window_budget` (`InvalidConfiguration`).

create_vault

Arguments: `agent_id: [u8; 32]` , `policy: PolicyArgs` .Accounts: `admin (s,w)` · `operator ( has_one admin )` · `protocol` · `mint` · `vault (w,init, [b"vault", operator, agent_id] )` · `vault_token (w,init?, ATA of vault )` · `token_program` · `associated_token_program` · `system_program` .

Checks: a valid policy. Effect: stores the vault with `window_started_at = now` ; emits `CoreEvent 2`.

set_policy

Arguments: `policy: PolicyArgs` .Accounts: `admin (s)` · `operator (w)` · `vault (w)` · `vault_token (w, authority vault )` · `token_program` · `protocol` .

Checks: a valid policy; `policy.guardian` equals the current guardian (`GuardianRotationRequired`); reconcile.

Effect: books consumed float, revokes the delegate, sets `last_approved = 0` , increments `policy_revision` and replaces the policy; emits `CoreEvent 3`.

rotate_session_key

Arguments: `session_key: Pubkey` .Accounts: as `set_policy` .

Checks: `session_key` is not zero (`InvalidConfiguration`); reconcile.

Effect: revokes the delegate, sets `last_approved = 0` and the new session key; credit lines keep their own voucher keys and `policy_revision` is unchanged; emits `CoreEvent 24`.

withdraw

Arguments: `amount: u64` .Accounts: `admin (s)` · `operator` · `protocol` · `vault (w)` · `vault_token (w)` · `destination (w, token account of the mint)` · `mint` · `token_program` .

Checks: vault not frozen (`VaultFrozen`); `amount > 0` (`InvalidAmount`); reconcile.

Effect: revokes the delegate, sets `last_approved = 0` , transfers `amount` to `destination` signed by the vault PDA; emits `CoreEvent 25`.

set_merchant_rule

Arguments: `per_transaction_cap: u64` , `window_budget: u64` , `quote_required: bool` .Accounts: `admin (s,w)` · `operator` · `vault` · `merchant` · `rule (w,init?, [b"rule", vault, merchant] )` · `system_program` .

Checks: account constraints only. Effect: a new rule records the vault, the merchant and the vault's current window start; the cap, budget and quote flag are overwritten; emits `CoreEvent 21`.

refill_float

Arguments: none. Accounts: `vault (w)` · `vault_token (w, authority vault )` · `session (address vault.policy.session_key ; does not sign)` · `token_program` .

Checks: vault not frozen (`VaultFrozen`); reconcile; `window_spent ≤ window_budget` (`WindowBudgetExceeded`).

Effect: SPL `Approve` of the session key for `min(float_cap, window_budget - window_spent)` , signed by the vault PDA; `last_approved` becomes that allowance; emits `CoreEvent 20`.

pay

Arguments: amount: u64 , nonce: u64 , quote_hash: [u8; 32] (zero when unquoted), intent_hash: [u8; 32] , quote_ix_index: Option<u16> .

Accounts: payer (s,w) · session (s; equals vault.policy.session_key , else Unauthorized) · protocol · mint · vault (w) · vault_token (w) · rule (w, [b"rule", vault, rule.merchant] , has_one vault) · merchant_token (w, owned by rule.merchant , else Unauthorized) · approval (w, opt, [b"approval", vault, intent_hash]) · approval_rent_payer (w, opt) · payment_receipt (w, init, [b"payment", vault, intent_hash]) · quote_receipt (w, opt, init?, [b"quote", vault, quote_hash]) · instructions · token_program · system_program .

Checks, in order:

1. Quoted: a proof index is given (QuoteRequired); the proof is canonical and 281 bytes (InvalidPrecompile); the 169-byte message has the quote tag, names this vault and rule.merchant , is signed by that merchant, has lane 0, this amount and an unexpired expires_at (InvalidQuote); its SHA-256 equals quote_hash and its nonce equals nonce (InvalidQuote); quote_receipt is present (InvalidQuote) and unused (PaymentReplay). Unquoted: no proof index and no quote_receipt (InvalidQuote).
2. intent_hash equals the payment intent of (vault, merchant, amount, nonce, quote hash, lane 0) (ApprovalMismatch).
3. Spend: vault not frozen (VaultFrozen); amount > 0 (InvalidAmount); reconcile; the rule's merchant (MerchantNotAllowed); amount ≤ per_transaction_cap (PerTransactionCapExceeded); a quote when the rule requires one (QuoteRequired); the merchant window, reset when the vault window moved (MerchantWindowBudgetExceeded); window_spent + amount + delegated_amount ≤ window_budget (WindowBudgetExceeded).
4. With a co-signer and amount > cosign_threshold : an approval is present (ApprovalRequired), unused (ApprovalUsed), unexpired (ApprovalExpired), of the current policy revision (ApprovalPolicyChanged) and binds this intent, merchant and amount (ApprovalMismatch).
5. A consumed approval's rent receiver is its recorded payer (ApprovalMismatch).

Effect: updates both windows; transfers amount to merchant_token signed by the vault PDA; writes the receipt (lane 0, outcome 1) and marks the quote used; emits PaymentCommitted and CoreEvent 23; closes a consumed approval to its payer.

approve_payment

Arguments: intent_hash: [u8; 32] , merchant: Pubkey , amount: u64 , expiry: u64 , ed25519_ix_index: u16 , policy_revision: u64 .

Accounts: payer (s,w) · protocol · vault · approval (w, init, [b"approval", vault, intent_hash]) · instructions · system_program · payment_receipt ([b"payment", vault, intent_hash]).

Checks: no payment receipt exists for the intent (PaymentReplay); policy_revision is current (ApprovalPolicyChanged); amount > 0 and now < expiry ≤ i64::MAX (InvalidAmount); a canonical 240-byte proof (InvalidPrecompile); a co-signer is set, signed the proof, and the message equals the 128-byte approval v2 bytes (ApprovalMismatch).

Effect: creates the Approval with the payer as rent payer; emits ApprovalCreated and CoreEvent 22.

approve_payment_passkey

Arguments: as `approve_payment` , with `precompile_ix_index: u16` in place of the Ed25519 index, then `authenticator_data: Vec<u8>` , `client_data_json: Vec<u8>` , `policy_revision: u64` .

Accounts: `payer (s,w) · protocol · vault · credential (w, [b"passkey", vault, credential_id_hash] , has_one vault ) · approval (w,init) · instructions · system_program · payment_receipt` .

Checks: receipt, revision and amount as `approve_payment` ; `vault.policy.cosigner` is this credential PDA (`ApprovalMismatch`); authenticator data of 37–128 bytes and client data of at most 512 bytes; client data is one JSON object without duplicate keys, with `type webauthn.get` , `challenge` equal to the unpadded base64url of SHA-256 over the approval v2 bytes, an origin whose SHA-256 is `allowed_origin_hash` , and `crossOrigin` absent or false; the authenticator data starts with `rp_id_hash` , has UP and UV set and AT clear; its counter is zero with a stored zero, or above the stored counter (all `InvalidPasskey`); the secp256r1 proof at index – 1 has the canonical header, this key and the message `authenticator_data || SHA-256(client_data_json)` (`InvalidPrecompile`).

Effect: stores the new counter; creates the `Approval` with the credential PDA as approver; emits `ApprovalCreated` and `CoreEvent 22`.

set_passkey_credential

Arguments: `credential_id_hash: [u8; 32]` , `public_key: [u8; 33]` , `rp_id_hash: [u8; 32]` , `allowed_origin_hash: [u8; 32]` . Accounts: `admin (s,w) · operator · vault · credential (w,init?, [b"passkey", vault, credential_id_hash] ) · system_program` .

Checks: the key starts with 0x02 or 0x03 (`InvalidPasskey`); an existing credential has the identical key and hashes (`InvalidPasskey`), and is then left unchanged.

Effect: stores the credential with `sign_count = 0` . No event.

close_approval

Arguments: none. Accounts: `caller (s) · vault · approval (w,close to rent_payer , has_one vault ) · rent_payer (w,address approval.rent_payer)` .

Checks: `now ≥ expiry` , or the approval's revision differs from the vault's (`ApprovalStillValid`).

Effect: emits `ApprovalClosed` ; rent returns to the original payer.

A.6 Escrow

pay_escrow

Arguments: `amount: u64` (quote principal), `quote_hash: [u8; 32]` , `intent_hash: [u8; 32]` , `ed25519_ix_index: u16` .

Accounts: `session_key (s,w;equals vault.policy.session_key ,else Unauthorized) · protocol · mint · vault (w) · vault_token (w) · rule (w, [b"rule", vault, merchant] , has_one vault , merchant) · merchant · escrow (w,init, [b"escrow", vault, quote_hash]) · escrow_token (w,init?, ATA of escrow) · approval (w,opt) · approval_rent_payer (w,opt) · payment_receipt (w,init) · quote_receipt (w,init?) · instructions · token_program · associated_token_program · system_program` .

Checks, in order: escrow fee and maximum set, fee within maximum, resolution duration set and positive (`InvalidConfiguration`); `fee_amount = [amount × fee_bps ÷ 10 000]` and `total_debit = amount + fee_amount` ; the quote proof in lane 1 for this vault, merchant and principal, unexpired (`InvalidPrecompile` , `InvalidQuote`); its hash equals `quote_hash` (`QuoteMismatch`); the arbiter is not the merchant (`MerchantCannotArbitrate`); `deliver_by` fits i64 (`InvalidQuote`) and `now ≤ deliver_by` (`DeadlinePassed`); `intent_hash` is the intent of (vault, merchant, total debit, quote nonce, quote hash, lane 1) (`ApprovalMismatch`); the spend and approval checks of `pay` on `total_debit` ; the quote receipt is unused (`PaymentReplay`); a consumed approval's rent receiver (`ApprovalMismatch`).

Effect: marks the quote used; writes the receipt (lane 1, outcome 0); creates the `Escrow` in state `Funded` with the fee snapshot, resolution duration, specification hash, deadline, arbiter and the session key as rent payer; moves `total_debit` into the escrow token account signed by the vault PDA; emits `PaymentCommitted` and an `EscrowEvent` carrying the 169-byte quote and the merchant signature; closes a consumed approval.

mark_delivered

Arguments: `deliverable_hash: [u8; 32]` .Accounts: `merchant (s) · escrow (w, [b"escrow", escrow.vault, escrow.quote_hash] , has_one merchant )` .

Checks: `state Funded ( StateTransitionNotAllowed )` ; `now ≤ deliver_by ( DeadlinePassed )` .

Effect: `delivered_at = now` , `dispute_until = now + 172 800` , the deliverable hash, state `Delivered` ; emits `EscrowEvent` .

dispute

Arguments: none.Accounts: `actor (s) · operator (address vault.operator ) · vault ( has_one operator ) · escrow (w, has_one vault )` .

Checks: the actor is the session key or the admin (`ActorNotAllowed`); state `Delivered` (`StateTransitionNotAllowed`); an arbiter is named (`ArbiterRequired`); `now < dispute_until` (`DeadlinePassed`).

Effect: state `Disputed` , `disputed_at = now` , `resolve_until = now + resolve_seconds` ; emits `EscrowEvent` .

Terminal escrow instructions

`accept` , `release_after_window` , `resolve` , `refund_on_timeout` and `refund_after_resolution_deadline` share one account list: `actor (s) · operator (address vault.operator) · vault · mint · protocol · escrow (w, close to rent_payer , has_one vault) · escrow_token (w, address escrow.token_account , authority escrow) · merchant_token (w; checked only on release) · vault_token (w, address vault.token_account) · fee_treasury (w, address protocol.fee_treasury , authority protocol) · payment_receipt (w, address escrow.payment_receipt) · rent_payer (w, address escrow.rent_payer) · token_program` .

Each first runs its own checks:

Instruction	Arguments	Checks	Outcome
<code>accept</code>	<code>none</code>	<code>actor is the session key or admin (ActorNotAllowed)</code> ; state <code>Delivered (StateTransitionNotAllowed)</code>	<code>release</code>

Instruction	Arguments	Checks	Outcome
<code>release_after_window</code>	none	state <code>Delivered</code> ; <code>now ≥ dispute_until</code> (<code>DeadlineNotReached</code>)	release
<code>resolve</code>	<code>release:</code> bool	state <code>Disputed</code> ; arbiter named (<code>ArbiterRequired</code>); actor is the arbiter (<code>ActorNotAllowed</code>); <code>now < resolve_until</code> (<code>DeadlinePassed</code>)	release if true, refund if false
<code>refund_on_timeout</code>	none	state <code>Funded</code> ; <code>now > deliver_by</code> (<code>DeadlineNotReached</code>)	refund
<code>refund_after_resolution_deadline</code>	none	state <code>Disputed</code> ; <code>resolve_until > 0</code> and <code>now ≥ resolve_until</code> (<code>DeadlineNotReached</code>)	refund

Then the shared routine checks that the escrow token account holds at least `amount + fee_amount` (`TokenAccountMismatch`). A release also checks that `merchant_token` is a classic SPL account of the mint owned by `escrow.merchant` (`TokenAccountMismatch`), pays `amount` to it and `fee_amount` to the fee account, and returns any surplus to the vault. A refund returns the whole escrow balance to the vault. Both close the escrow token account and the `Escrow` to the rent payer, set state `Released` or `Refunded`, write outcome 1 or 2 and `completed_at` into the receipt, and emit `EscrowEvent` .

A.7 Freeze and guardians

freeze

Arguments: none. Accounts: `actor` (s,w) · `operator` (w, [b"operator", operator.admin]) · `vault` (w, has_one operator) · `vault_token` (w, authority vault) · `token_program` · `freeze_checkpoint` (w, [b"freeze", vault, generation] : the current generation when frozen, the next one otherwise) · `system_program` · `freeze_reserve` (w, [b"freeze-reserve", vault] , system-owned).

Checks: the actor is the admin, the operator guardian or the vault guardian (`Unauthorized`); reconcile; on a first freeze, the checkpoint is the derived PDA, system-owned and empty (`InvalidFreezeCheckpoint`), and its rent comes from the reserve or, when the reserve is short, from the actor (`FreezeReserveInsufficient`).

Effect: books consumed float and revokes the delegate. On a first freeze: the generation rises; the checkpoint is funded, allocated (89 bytes), assigned to the program and written with the freeze time, `defaults_opened_at_freeze = operator.defaults_count` and its episode; a new episode starts when none exists or the previous recourse has ended; the vault is frozen and `operator.frozen_vault_count` rises. A reserve that would be left with a nonzero balance below the rent-exempt minimum moves entirely to the checkpoint. Always: `last_approved = 0` and `CoreEvent` 15.

unfreeze

Arguments: none. Accounts: `admin` (s) · `operator` (w) · `vault` (w) · `vault_token` (w) · `token_program` · `protocol` · `freeze_checkpoint` (w).

Checks, when frozen: the checkpoint is the current generation's PDA, program-owned, for this vault and generation, and not yet finalized (`InvalidFreezeCheckpoint`).

Effect: adds the elapsed frozen interval to `episode_frozen_seconds` ; sets `recourse_until = now + freeze_grace_seconds + 259 200` on the vault and the checkpoint and `unfrozen_at = now` on the checkpoint; raises `operator.recourse_lock_until` to it; lowers `frozen_vault_count` ; unfreezes; emits `CoreEvent` 16. The float stays revoked until `refill_float` .

fund_freeze_reserve

Arguments: amount: u64 (lamports). Accounts: admin (s,w) · operator · vault · freeze_reserve (w, system-owned) · system_program .

Checks: amount > 0 (InvalidAmount). Effect: system transfer from the admin to the reserve. No event.

close_freeze_checkpoint

Arguments: none. Accounts: operator · vault (has_one operator) · freeze_checkpoint (w, close to freeze_reserve , [b"freeze", vault, checkpoint.generation] , has_one vault) · freeze_reserve (w).

Checks: for a checkpoint of an older episode, now > freeze_episode_started_at + 259 200 ; for the current episode, the vault is unfrozen and now > recourse_until + 259 200 (FreezeCheckpointInUse).

Effect: closes the checkpoint; its rent returns to the reserve. No event.

rotate_operator_guardian and rotate_vault_guardian

Arguments: guardian: Pubkey (zero removes the guardian). Accounts: admin (s) · operator (w for the operator variant) · vault (w; vault variant only).

Checks: when guardian equals the pending change, now ≥ effective_at (GuardianRotationDelayNotReached); otherwise guardian differs from the current one (InvalidConfiguration).

Effect: a matured repeat applies the change; anything else schedules it, replacing a different pending change, at now + 259 200 . Emits GuardianRotated with applied and effective_at . Neither freeze nor default blocks a rotation.

Appendix B. Account layouts

This appendix gives the byte layout of every account the `bonded` program owns, in field order, from `state.rs`, `policy.rs`, `passkey.rs` and `escrow.rs`. Sizes are the allocated `8 + INIT_SPACE`.

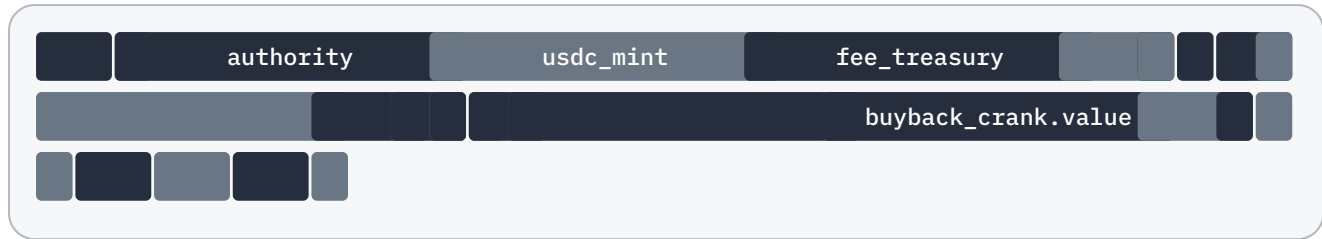
B.1 Conventions

- Bytes 0–7 are the account discriminator, the first 8 bytes of SHA-256 of `account: <Name>`. Byte 8 is the PDA bump. There is no version byte.
- Integers are little endian. `Pubkey` and hashes are raw 32-byte values. Amounts are `u64` base units; times are `i64` Unix seconds; `bool` is one byte, 0 or 1.
- An `Option<T>` is a one-byte tag followed by `T` when the tag is 1. Allocation reserves the full `T`, but Borsh writes an absent value as the tag alone, so every later field moves up by the size of `T`. The offsets below assume every option is present; each section states the shift. Readers decode sequentially.

Account	Discriminator	Bytes	Options
<code>ProtocolConfig</code>	<code>cf5bfa1c98b3d7d1</code>	286	6
<code>Operator</code>	<code>db1fbc91458bcc75</code>	150	1
<code>Bond</code>	<code>e08030fbb6f66fc4</code>	141	2
<code>Vault</code>	<code>d308e82b02987577</code>	351	1
<code>MerchantRule</code>	<code>2a0a799c2db7f464</code>	106	0
<code>Approval</code>	<code>e90999310bde3b82</code>	162	0
<code>PaymentReceipt</code>	<code>a8c6d1043ceb7e6d</code>	211	0
<code>QuoteReceipt</code>	<code>835d19278ef7d944</code>	138	0
<code>PasskeyCredential</code>	<code>1529c8778f2d212c</code>	174	0
<code>CreditLine</code>	<code>dce2cd18dc978168</code>	302	3
<code>VoucherAttestation</code>	<code>903228d186a9fe77</code>	145	0
<code>DefaultEvent</code>	<code>be17805109f01023</code>	156	0
<code>Claim</code>	<code>9b4616b07bd7f666</code>	170	0
<code>FreezeCheckpoint</code>	<code>57d10106b6f2f938</code>	89	0
<code>Escrow</code>	<code>1fd57bbbba16da9b</code>	400	0

The fee account at `[b"fees"]` and the vault, bond and escrow token accounts are standard 165-byte SPL Token accounts, and the freeze reserve is a system account with no data.

B.2 Protocol



286 bytes · one cell per 4 bytes

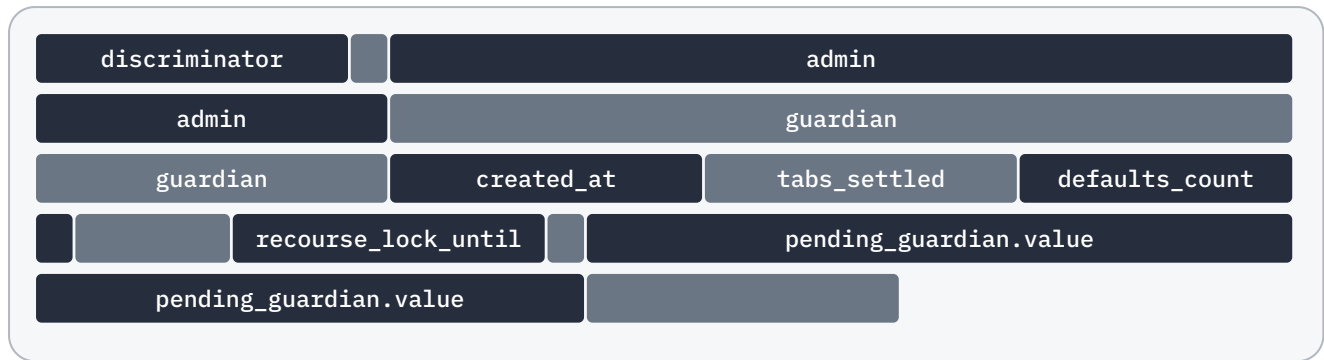
Offset	Bytes	Field	Meaning
0	8	discriminator	cf5bfa1c98b3d7d1
8	1	bump	PDA bump of [b"protocol"]
9	32	authority	upgrade authority at the last governed call
41	32	usdc_mint	classic SPL mint, 6 decimals
73	32	fee_treasury	the [b"fees"] token account
105	8	total_bond_balance	u64, sum of booked bond balances
113	2	tab_fee_bps	u16, snapshotted by new lines
115	1	tab_fee_waived	bool, new lines take 0 bps
116	4	withdrawal_margin_seconds	u32, > 0
120	2	maximum_tab_fee_bps	u16, ≤ 10 000, immutable
122	4	freeze_grace_seconds	u32, ≥ 345 600, immutable
126	32	cluster_genesis_hash	raw genesis hash
158	8	voucher_domain	SHA-256 tag of the voucher v2 domain
166	1	escrow_fee_bps.tag	0 until configure_escrow_fee
167	2	escrow_fee_bps.value	u16
169	1	maximum_escrow_fee_bps.tag	fixed once set
170	2	maximum_escrow_fee_bps.value	u16, ≤ 10 000
172	1	escrow_resolve_seconds.tag	0 until configured
173	4	escrow_resolve_seconds.value	u32, > 0
177	1	buyback_token_mint.tag	0 until configure_buyback
178	32	buyback_token_mint.value	the \$BONDED mint
210	1	buyback_crank.tag	0 until configure_buyback
211	32	buyback_crank.value	the one key that runs buybacks
243	8	buyback_minimum_usdc	u64, the smallest claim
251	1	pending_buyback.tag	1 only inside a buyback transaction

Offset	Bytes	Field	Meaning
252	8	pending_buyback.amount	u64, USDC claimed
260	8	pending_buyback.minimum_output	u64, least \$BONDED gain
268	8	pending_buyback.token_balance_before	u64
276	8	pending_buyback.usdc_balance_before	u64
284	2	pending_buyback.claim_instruction_index	u16

Figure 56. The 286-byte ProtocolConfig account, with every option present.

Freshly initialized, all five options are absent and the serialized data ends at byte 180: tag bytes at 166 to 170, the buyback minimum (zero until configured) at 171 to 178, and the pending-buyback tag at 179.

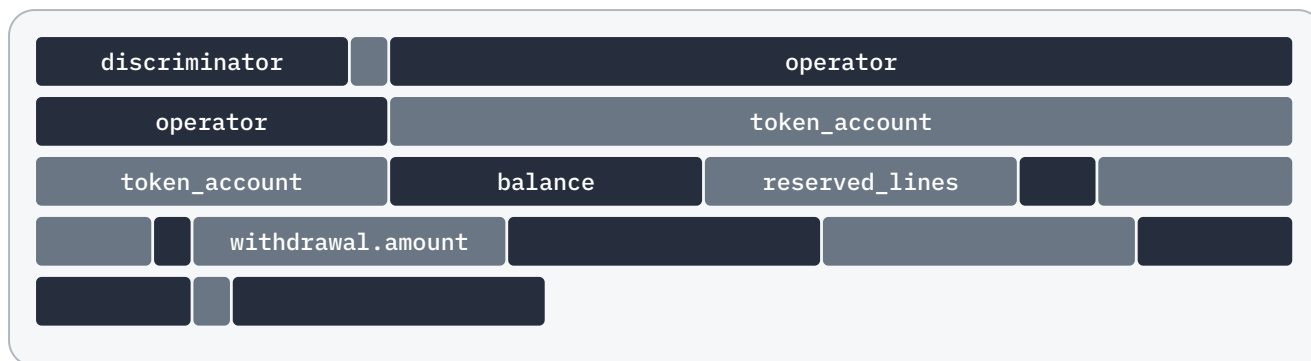
B.3 Operator and bond



150 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	db1fbc91458bcc75
8	1	bump	PDA bump of [b"operator", admin]
9	32	admin	signs every operator instruction
41	32	guardian	operator-level guardian; zero for none
73	8	created_at	i64
81	8	tabs_settled	u64, healthy settlements
89	8	defaults_count	u64, defaults opened
97	4	frozen_vault_count	u32, vaults frozen now
101	8	recourse_lock_until	i64, latest unfreeze tail end
109	1	pending_guardian.tag	1 while a rotation is scheduled
110	32	pending_guardian.value	replacement guardian
142	8	pending_guardian.effective_at	i64, now + 72 h when scheduled

Figure 57. The 150-byte Operator account, with a pending guardian change present.



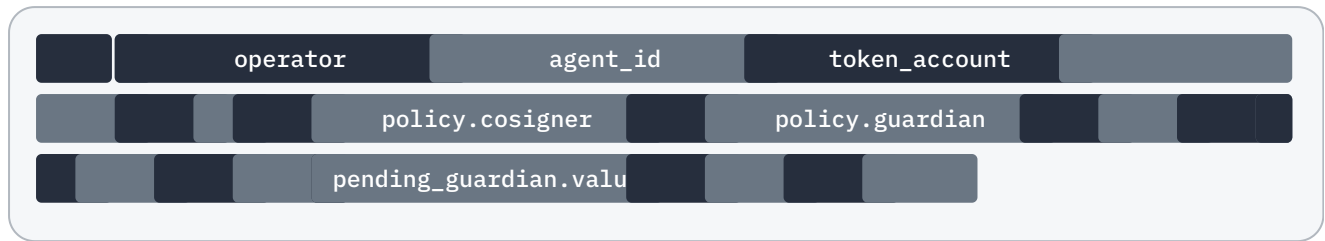
141 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	e08030fbb6f66fc4
8	1	bump	PDA bump of [b"bond", operator]
9	32	operator	the Operator PDA
41	32	token_account	ATA of the bond PDA
73	8	balance	u64, booked collateral
81	8	reserved_lines	u64, Σ line limits
89	2	minimum_coverage_bps	u16, $\geq 3\ 300$
91	8	longest_line_expiry	i64, high-water mark
99	1	withdrawal.tag	1 while a request is pending
100	8	withdrawal.amount	u64
108	8	withdrawal.requested_at	i64
116	8	withdrawal.unlock_at	i64
124	8	default_sequence	u64, last default opened
132	1	active_default_sequence.tag	1 during a default
133	8	active_default_sequence.value	u64

Figure 58. The 141-byte Bond account, with a pending withdrawal and an active default present.

Without a pending withdrawal, `default_sequence` starts at byte 100 and `active_default_sequence` at 108.

B.4 Vault and policy accounts



351 bytes · one cell per 4 bytes

Offset	Bytes	Field	Meaning
0	8	discriminator	d308e82b02987577
8	1	bump	PDA bump of [b"vault", operator, agent_id]
9	32	operator	the Operator PDA
41	32	agent_id	32-byte seed
73	32	token_account	ATA of the vault PDA
105	32	policy.session_key	the agent's key
137	8	policy.float_cap	u64
145	4	policy.window_seconds	u32, > 0
149	8	policy.window_budget	u64, ≥ float_cap
157	32	policy.cosigner	Ed25519 key or credential PDA; zero for none
189	8	policy.cosign_threshold	u64
197	32	policy.guardian	vault-level guardian; zero for none
229	8	window_started_at	i64, anchored window start
237	8	window_spent	u64, booked spend this window
245	8	last_approved	u64, allowance at the last approve or reconcile
253	1	frozen	bool
254	8	frozen_at	i64, start of the current freeze
262	8	recourse_until	i64, end of the unfreeze tail
270	8	freeze_generation	u64, count of freezes
278	8	policy_revision	u64, bumped by set_policy
286	1	pending_guardian.tag	1 while a rotation is scheduled
287	32	pending_guardian.value	replacement guardian
319	8	pending_guardian.effective_at	i64
327	8	freeze_episode_start_generation	u64
335	8	freeze_episode_started_at	i64

Offset	Bytes	Field	Meaning
343	8	episode_frozen_seconds	u64, completed frozen intervals

Figure 59. The 351-byte Vault account, with its embedded PolicyArgs and a pending guardian present.

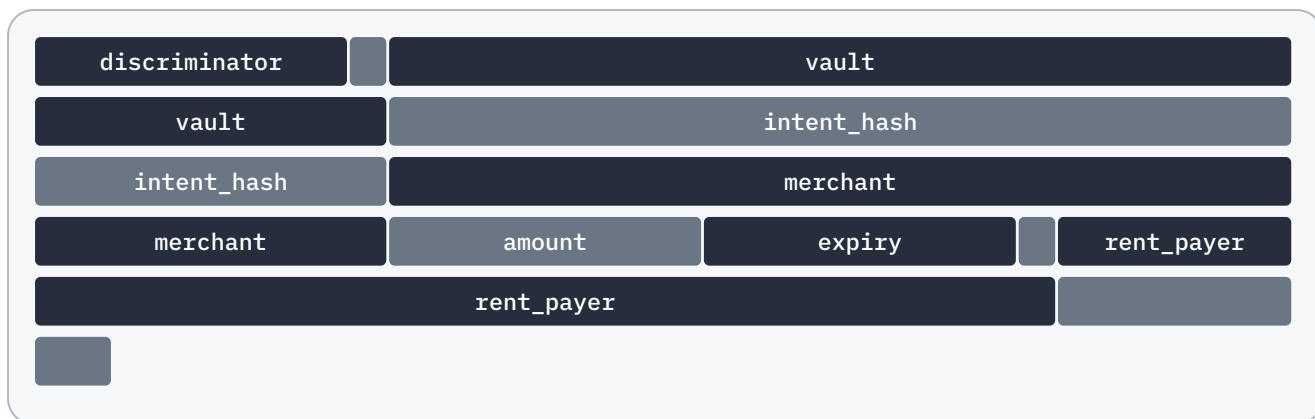
Without a pending guardian, the three episode fields start at bytes 287, 295 and 303.



106 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	2a0a799c2db7f464
8	1	bump	PDA bump of [b"rule", vault, merchant]
9	32	vault	the Vault PDA
41	32	merchant	allowlisted merchant wallet
73	8	per_transaction_cap	u64
81	8	window_budget	u64, per merchant per window
89	8	window_spent	u64, valid when window_started_at matches the vault
97	1	quote_required	bool
98	8	window_started_at	i64, the window the counter belongs to

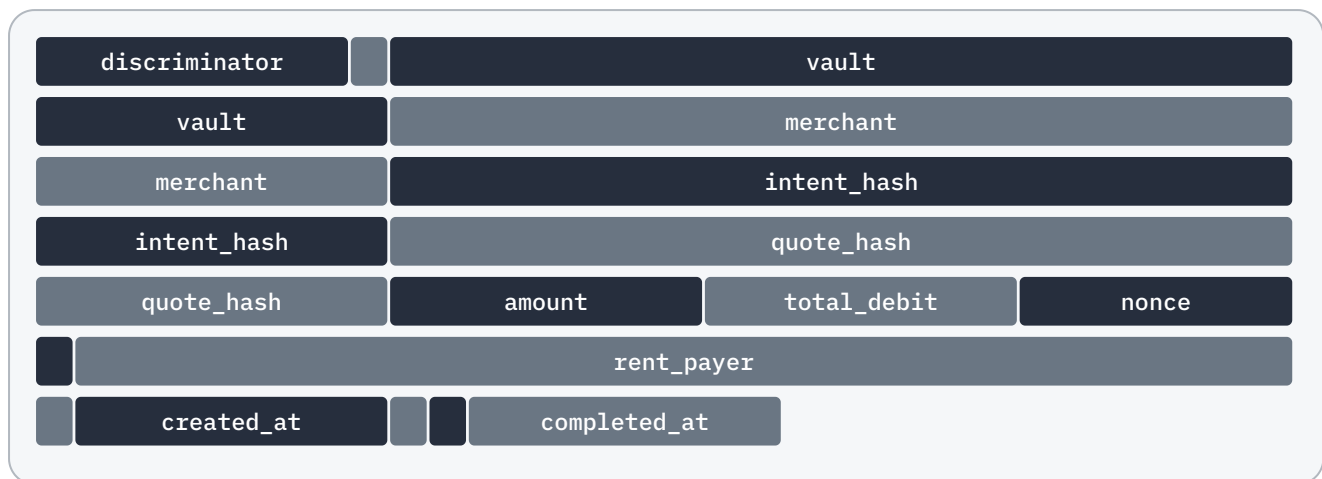
Figure 60. The 106-byte MerchantRule account.



162 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	e90999310bde3b82
8	1	bump	PDA bump of [b"approval", vault, intent_hash]
9	32	vault	the Vault PDA
41	32	intent_hash	SHA-256 payment intent
73	32	merchant	the bound merchant
105	8	amount	u64, the bound total debit
113	8	expiry	i64, exclusive
121	1	used	bool
122	32	rent_payer	receives the rent on close
154	8	policy_revision	u64, the revision it was signed under

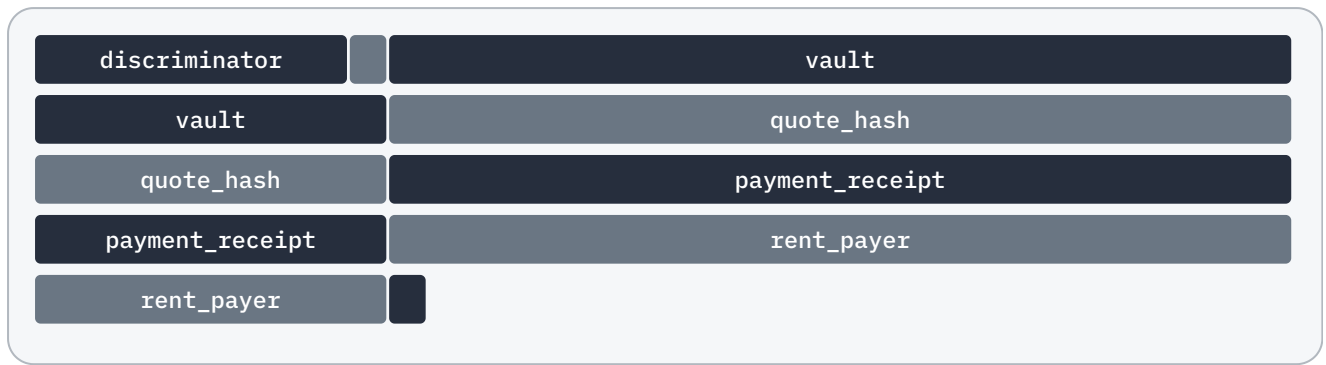
Figure 61. The 162-byte Approval account.



211 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	a8c6d1043ceb7e6d
8	1	bump	PDA bump of [b"payment", vault, intent_hash]
9	32	vault	the Vault PDA
41	32	merchant	payee
73	32	intent_hash	the consumed intent
105	32	quote_hash	zero for an unquoted pay
137	8	amount	u64, principal
145	8	total_debit	u64, principal plus escrow fee
153	8	nonce	u64, payment or quote nonce
161	32	rent_payer	payer of this receipt
193	8	created_at	i64
201	1	lane	0 pay, 1 escrow
202	1	outcome	0 pending escrow, 1 merchant paid, 2 refunded
203	8	completed_at	i64, 0 while pending

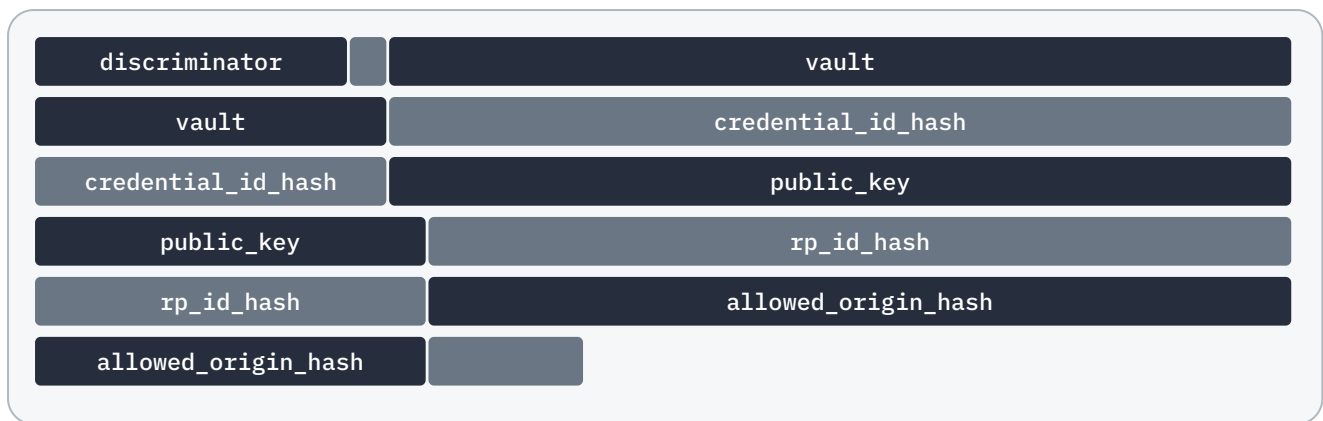
Figure 62. The 211-byte PaymentReceipt account.



138 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	835d19278ef7d944
8	1	bump	PDA bump of [b"quote", vault, quote_hash]
9	32	vault	the Vault PDA
41	32	quote_hash	SHA-256 of the 169-byte quote
73	32	payment_receipt	the receipt that used it
105	32	rent_payer	payer of this receipt
137	1	used	bool, always 1 once written

Figure 63. The 138-byte QuoteReceipt account.

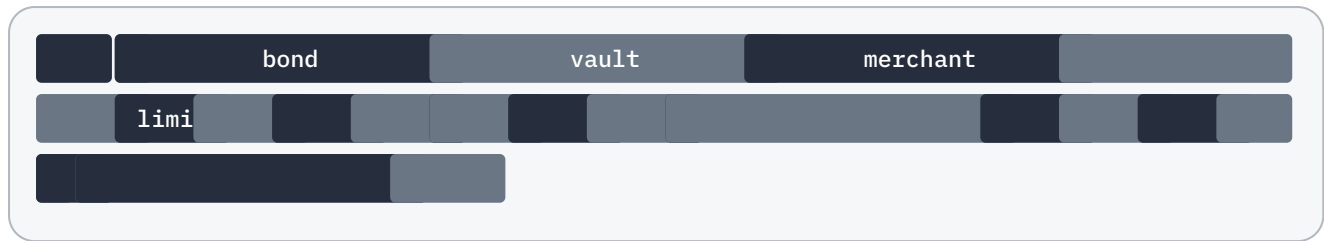


174 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	1529c8778f2d212c
8	1	bump	PDA bump of [b"passkey", vault, credential_id_hash]
9	32	vault	the Vault PDA
41	32	credential_id_hash	SHA-256 of the WebAuthn credential ID
73	33	public_key	compressed P-256 key, 0x02 or 0x03 prefix
106	32	rp_id_hash	SHA-256 of the relying-party ID
138	32	allowed_origin_hash	SHA-256 of the allowed origin
170	4	sign_count	u32, last authenticator counter

Figure 64. The 174-byte PasskeyCredential account.

B.5 Lines and defaults

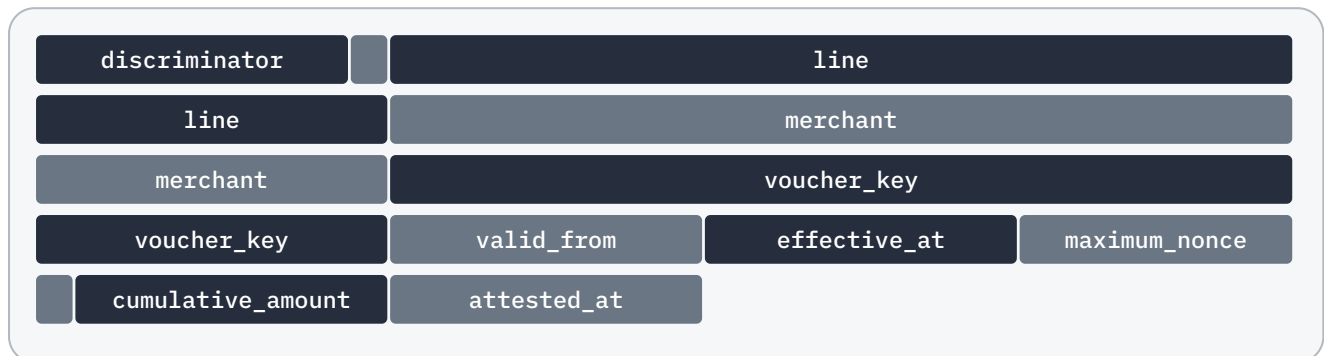


302 bytes · one cell per 4 bytes

Offset	Bytes	Field	Meaning
0	8	discriminator	dce2cd18dc978168
8	1	bump	PDA bump of [b"line", bond, merchant]
9	32	bond	the Bond PDA
41	32	vault	the vault that pays this tab
73	32	merchant	line holder
105	32	voucher_session_key	current voucher key
137	8	limit	u64, applied limit
145	8	settled_cumulative	u64, paid by the vault
153	8	maximum_nonce	u64, highest settled or claimed nonce
161	8	expires_at	i64
169	1	pending_limit.tag	1 while a decrease is scheduled
170	8	pending_limit.value	u64
178	8	pending_limit.effective_at	i64
186	8	opened_default_sequence	u64, epoch of the line
194	1	pending_voucher_key.tag	1 while a key switch is scheduled
195	32	pending_voucher_key.value	incoming key
227	8	pending_voucher_key.effective_at	i64, switch time
235	8	retired_cumulative	u64, debt resolved by a default
243	8	opened_at	i64, first opening
251	8	freeze_generation_at_open	u64
259	2	tab_fee_bps	u16, immutable
261	1	previous_voucher_key.tag	1 after a materialized switch
262	32	previous_voucher_key.value	outgoing key
294	8	rotated_at	i64, last switch time

Figure 65. The 302-byte CreditLine account, with all three options present.

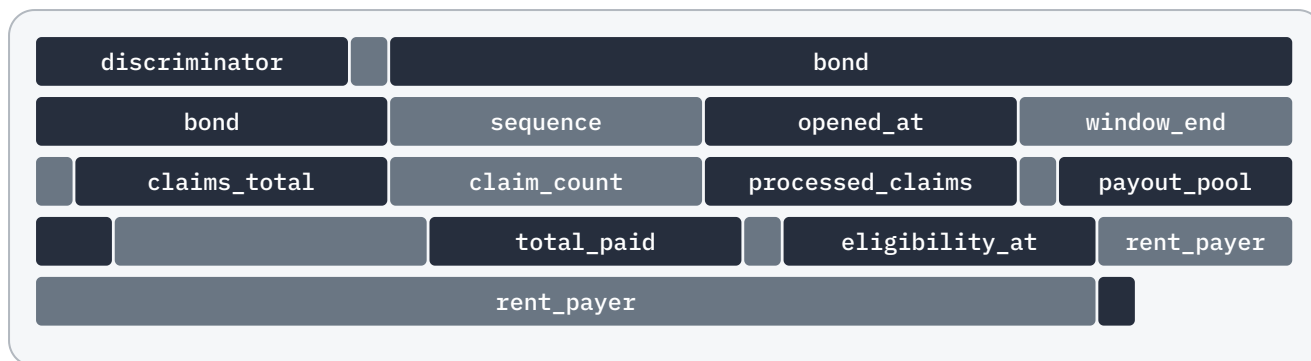
An absent `pending_limit` moves every later field up by 16 bytes, an absent `pending_voucher_key` by 40 more, and an absent `previous_voucher_key` moves `rotated_at` up by 32. A line with no pending changes and no rotation serializes in 214 bytes.



145 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	903228d186a9fe77
8	1	bump	PDA bump of [b"voucher-attestation", line, key, effective_at]
9	32	line	the CreditLine PDA
41	32	merchant	the attesting line holder
73	32	voucher_key	the outgoing key
105	8	valid_from	i64, when that key became current
113	8	effective_at	i64, the switch time
121	8	maximum_nonce	u64, ceiling for old-key vouchers
129	8	cumulative_amount	u64, ceiling for old-key vouchers
137	8	attested_at	i64

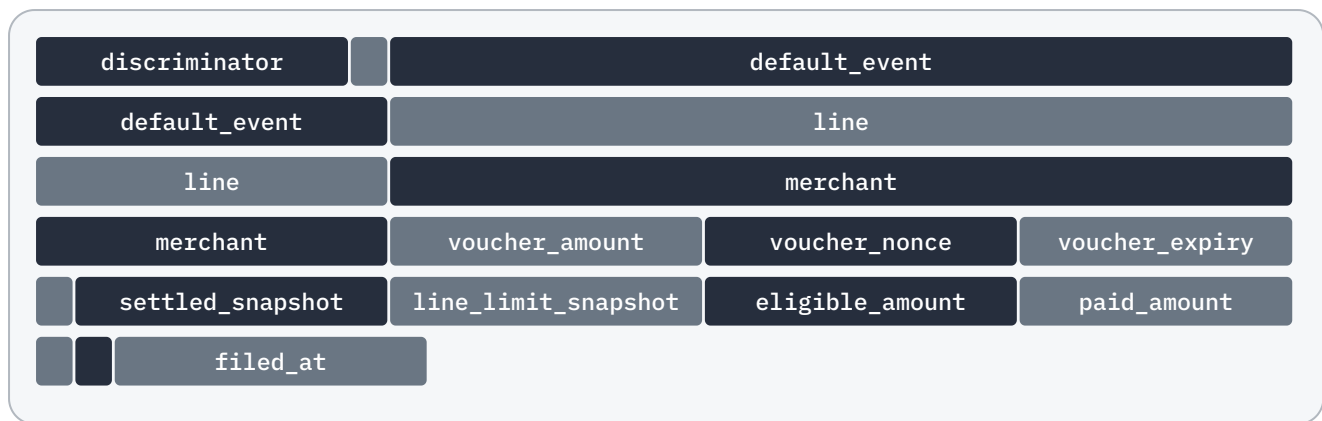
Figure 66. The 145-byte VoucherAttestation account.



156 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	be17805109f01023
8	1	bump	PDA bump of [b"default", bond, sequence]
9	32	bond	the Bond PDA
41	8	sequence	u64
49	8	opened_at	i64, 0 for a placeholder
57	8	window_end	i64, opened_at + 72 h
65	8	claims_total	u64, Σ eligible of filed claims
73	8	claim_count	u64
81	8	processed_claims	u64
89	1	snapshot_taken	bool, set by the first payout
90	8	payout_pool	u64, bond balance at the first payout
98	8	eligible_total_snapshot	u64, claims_total at the first payout
106	8	total_paid	u64
114	1	paid	bool, all claims processed
115	8	eligibility_at	i64, validation time that opened it
123	32	rent_payer	merchant that paid the rent
155	1	collateral_locked_at_open	bool

Figure 67. The 156-byte DefaultEvent account.

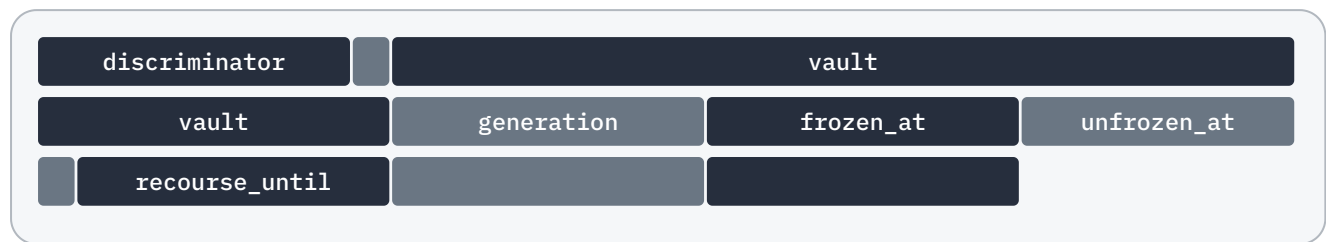


170 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	9b4616b07bd7f666
8	1	bump	PDA bump of [b"claim", default_event, merchant]
9	32	default_event	the DefaultEvent PDA
41	32	line	the CreditLine PDA
73	32	merchant	claimant
105	8	voucher_amount	u64, cumulative of the filed voucher
113	8	voucher_nonce	u64
121	8	voucher_expiry	u64, as on the wire
129	8	settled_snapshot	u64, line floor at first filing
137	8	line_limit_snapshot	u64
145	8	eligible_amount	u64
153	8	paid_amount	u64
161	1	paid	bool
162	8	filed_at	i64, latest filing

Figure 68. The 170-byte Claim account.

B.6 Freeze



89 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	discriminator	57d10106b6f2f938
8	1	bump	PDA bump of [b"freeze", vault, generation]
9	32	vault	the Vault PDA
41	8	generation	u64, the vault's freeze count at this freeze
49	8	frozen_at	i64
57	8	unfrozen_at	i64, 0 while the freeze lasts
65	8	recourse_until	i64, written once by unfreeze
73	8	defaults_opened_at_freeze	u64, operator defaults_count
81	8	episode_start_generation	u64

Figure 69. The 89-byte FreezeCheckpoint account.

B.7 Escrow



400 bytes · one cell per 4 bytes

Offset	Bytes	Field	Meaning
0	8	discriminator	1fd57bbba16da9b
8	1	bump	PDA bump of [b"escrow", vault, quote_hash]
9	32	vault	the paying Vault PDA
41	32	quote_hash	SHA-256 of the merchant quote
73	32	token_account	ATA of the escrow PDA
105	32	merchant	payee named in the quote
137	8	amount	u64, principal
145	8	fee_amount	u64, reserved, earned on release
153	2	fee_bps	u16, snapshot at funding
155	32	specification_hash	from the quote
187	8	deliver_by	i64, inclusive delivery deadline
195	8	delivered_at	i64, 0 until delivery
203	8	dispute_until	i64, delivered_at + 48 h
211	32	deliverable_hash	set by mark_delivered
243	32	arbiter	zero for none
275	1	state	0 Funded, 1 Delivered, 2 Disputed, 3 Released, 4 Refunded
276	32	payment_receipt	the PaymentReceipt PDA
308	32	quote_receipt	the QuoteReceipt PDA
340	32	rent_payer	the funding session key
372	8	created_at	i64
380	4	resolve_seconds	u32, snapshot at funding
384	8	disputed_at	i64, 0 unless disputed
392	8	resolve_until	i64, disputed_at + resolve_seconds

Figure 70. The 400-byte Escrow account.

States 3 and 4 appear only in the final `EscrowEvent` and in the receipt's outcome, because the account closes in the same instruction that sets them.

Appendix C. Error codes

C.1 Program errors

The program defines two error enums. `BondedError` takes Anchor's default base, so its variants are 6000–6052 in declaration order; `EscrowError` is declared with `offset = 7000`, so its variants are 7000–7007. A failed instruction returns `Custom(code)` and logs `Error Code: <Name>` and `Error Message: <message>`. Existing numbers never change: new variants are appended at the end of the enum.

Errors from account validation that runs before any handler (a wrong PDA, a failed `has_one`, an account that already exists for `init`, a missing signer) are Anchor's or the System Program's own errors, not these codes. The last column is the API code a client receives for each program error (C.2).

Code	Name	Message	Raised by	API code
6000	Unauthenticated	This signer is not authorized.	<code>freeze</code> (signer is not admin or a guardian); the five governed instructions (not the upgrade authority); <code>pay</code> , <code>pay_escrow</code> (not the session key); <code>pay</code> , <code>execute_default_payout</code> (merchant token owner); <code>resize_line</code> (line of another vault); <code>file_claim</code> (vault of another operator)	FORBIDDEN
6001	InvalidMint	The classic SPL mint must have six decimals.	<code>initialize_protocol</code>	INVALID_INPUT
6002	InvalidConfiguration	The protocol or policy configuration is invalid.	<code>initialize_protocol</code> , <code>update_protocol</code> , <code>configure_escrow_fee</code> , <code>configure_escrow_resolution</code> ; <code>create_vault</code> , <code>set_policy</code> (policy); <code>rotate_session_key</code> , <code>rotate_line_key</code> , both guardian rotations (unchanged key); <code>attest_line_voucher</code> (no pending rotation); <code>pay_escrow</code> (escrow unconfigured); the window reconcile in <code>refill_float</code> , <code>pay</code> , <code>pay_escrow</code> , <code>set_policy</code> , <code>rotate_session_key</code> , <code>withdraw</code> , <code>freeze</code>	CONFIGURATION_REQUIRED
6003	InvalidAmount	The amount is invalid.	<code>claim_buyback</code> , <code>post_bond</code> , <code>open_line</code> , <code>renew_line</code> , <code>request_withdrawal</code> , <code>withdraw</code> , <code>fund_freeze_reserve</code> , <code>pay</code> , <code>pay_escrow</code> , <code>approve_payment</code> , <code>approve_payment_passkey</code> , <code>execute_default_payout</code>	INVALID_INPUT
6004	ArithmeticOverflow	Checked arithmetic exceeded its supported range.	any checked addition or subtraction on balances, counters or timestamps	INVALID_INPUT
6005	CoverageTooLow	Coverage would fall below the operator minimum.	<code>create_operator</code> (minimum below 3,300 bps), <code>open_line</code> , <code>resize_line</code> , <code>renew_line</code> , <code>request_withdrawal</code> , <code>finalize_withdrawal</code>	COVERAGE_TOO_LOW
6006	BetaCapExceeded	The beta bond cap would be exceeded.	<code>post_bond</code>	INVALID_INPUT
6007	DefaultActive	A default event is active for this bond.	<code>open_line</code> , <code>resize_line</code> , <code>renew_line</code> , <code>settle_tab</code> , <code>request_withdrawal</code> , <code>finalize_withdrawal</code>	INVALID_TRANSITION

Code	Name	Message	Raised by	API code
6008	InvalidDefault	The requested default event is invalid.	settle_tab , file_claim , execute_default_payout , finalize_empty_default , close_claim , close_default_placeholder	INVALID_TRANSITION
6009	LineExpired	The credit line is expired or retired by a default.	resize_line , settle_tab	DEADLINE_PASSED
6010	DecreaseDelayNotReached	The line limit decrease is still cooling down.	resize_line	DEADLINE_NOT_REACHED
6011	VaultFrozen	The vault is frozen.	open_line , resize_line , renew_line , settle_tab , refill_float , pay , pay_escrow , withdraw	VAULT_FROZEN
6012	VoucherExpired	The voucher is expired or outlives the credit line.	settle_tab , file_claim , attest_line_voucher	VOUCHER_EXPIRED
6013	VoucherReplace	The voucher nonce or cumulative amount does not advance state.	settle_tab , file_claim , attest_line_voucher , close_claim	VOUCHER_REPLACE
6014	LineLimitExceeded	The voucher exceeds the credit line limit.	settle_tab , attest_line_voucher	PER_TRANSACTION_CAP_EXCEEDED
6015	InvalidPrecompile	The immediately preceding Ed25519 instruction must use canonical self-contained offsets.	settle_tab , file_claim , attest_line_voucher , approve_payment , approve_payment_passkey , pay with a quote , pay_escrow	INVALID_INPUT
6016	InvalidVoucher	The voucher domain or credit line does not match.	settle_tab , file_claim , attest_line_voucher	INVALID_INPUT
6017	ClaimWindowClosed	The default claim window is closed.	file_claim	DEADLINE_PASSED
6018	ClaimWindowOpen	The default claim window is still open.	execute_default_payout , finalize_empty_default	DEADLINE_NOT_REACHED
6019	WithdrawalExists	A withdrawal request already exists.	request_withdrawal	CONFLICT
6020	NoWithdrawal	No withdrawal request exists.	finalize_withdrawal	NOT_FOUND

Code	Name	Message	Raised by	API code
6021	WithdrawalCoolingDown	The bond withdrawal is still cooling down.	finalize_withdrawal	WITHDRAWAL_COOLING_DOWN
6022	FrozenRecourseActive	Frozen-vault recourse still protects this operator's collateral.	resize_line (applying a decrease), renew_line , request_withdrawal , finalize_withdrawal	VAULT_FROZEN
6023	KeyRotationDelayNotReached	The credit-line voucher key rotation is still cooling down.	rotate_line_key	DEADLINE_NOT_REACHED
6024	WindowBudgetExceeded	The vault window budget would be exceeded.	refill_float , pay , pay_escrow	WINDOW_BUDGET_EXCEEDED
6025	MerchantNotAllowed	This merchant is not allowed by the vault.	pay , pay_escrow	FORBIDDEN
6026	MerchantTransactionCapExceeded	The merchant per-transaction cap would be exceeded.	pay , pay_escrow	PER_TRANSACTION_CAP_EXCEEDED
6027	QuoteRequired	A valid merchant-signed quote is required.	pay	QUOTE_REQUIRED
6028	MerchantWindowBudgetExceeded	The merchant window budget would be exceeded.	pay , pay_escrow	WINDOW_BUDGET_EXCEEDED
6029	ApprovalRequired	A co-signer approval is required.	pay , pay_escrow	APPROVAL_REQUIRED
6030	ApprovalUsed	This approval has already been used.	pay , pay_escrow	APPROVAL_USED
6031	ApprovalExpired	This approval has expired.	pay , pay_escrow	APPROVAL_EXPIRED
6032	ApprovalMismatch	The approval does not bind this exact payment.	pay , pay_escrow (intent hash, approval fields, rent receiver); approve_payment (co-signer or message); approve_payment_passkey (credential is not the co-signer)	INVALID_INPUT
6033	InvalidQuote	The quote domain, fields, signature or expiry is invalid.	pay , pay_escrow	INVALID_INPUT

Code	Name	Message	Raised by	API code
6034	PaymentReplacement	The payment intent or quote has already been consumed.	pay , pay_escrow , approve_payment , approve_payment_passkey	CONFLICT
6035	InvalidPasskey	The passkey assertion, origin, RP ID, flags or counter is invalid.	approve_payment_passkey , set_passkey_credential	INVALID_INPUT
6036	InvalidFreezeCheckpoint	The freeze checkpoint does not match this credit line.	freeze , unfreeze , settle_tab , file_claim	INVALID_INPUT
6037	GuardianRotationRequired	Guardian changes require the delayed rotation instruction.	set_policy	INVALID_TRANSITION
6038	GuardianRotationDelayNotReached	The guardian rotation is still cooling down.	rotate_operator_guardian , rotate_vault_guardian	DEADLINE_NOT_REACHED
6039	ApprovalPolicyChanged	The approval belongs to a previous policy revision.	approve_payment , approve_payment_passkey , pay , pay_escrow	CONFLICT
6040	ApprovalStillValid	An unexpired approval for the current policy cannot be closed.	close_approval	DEADLINE_NOT_REACHED
6041	FreezeCheckpointInUse	This freeze checkpoint can still be used by a merchant claim.	close_freeze_checkpoint	DEADLINE_NOT_REACHED
6042	FreezeReserveInsufficient	Pre-fund the freeze reserve or provide a rent-funded freezing signer.	freeze	INSUFFICIENT_FUNDS
6043	AttestationNoticeClosed	A voucher can be attested only before its scheduled key switch.	attest_line_voucher	DEADLINE_PASSED
6044	InvalidVoucherAttestation	The voucher attestation does not match this line or key epoch.	settle_tab , file_claim	FORBIDDEN

Code	Name	Message	Raised by	API code
6045	BuybackNotConfigured	Configure a token mint and crank before claiming protocol fees.	claim_buyback	— (the worker's crank)
6046	BuybackPending	A buyback claim is already pending.	configure_buyback , claim_buyback	—
6047	BuybackMissingBurn	A matching burn must follow the fee claim in this transaction.	claim_buyback	—
6048	BuybackMissingClaim	A matching fee claim must precede this burn in the same transaction.	burn_buyback	—
6049	BuybackOutputTooLow	The bought token balance gain is below the claimed minimum.	burn_buyback	—
6050	BuybackUsdcNotSpent	The claimed USDC must be spent before burning the bought tokens.	burn_buyback	—
6051	BuybackInvalidTokenMint	The buyback mint or its extensions are unsupported.	configure_buyback , claim_buyback , burn_buyback	—
6052	BuybackInsufficientFees	The fee treasury cannot cover this buyback claim.	claim_buyback	—
7000	StateTransitionNotAllowed	The escrow state does not allow this action.	mark_delivered , dispute , accept , release_after_window , resolve , refund_on_timeout , refund_after_resolution_deadline	INVALID_TRANSITION
7001	ActorNotAllowed	This wallet cannot perform this escrow action.	dispute , accept (not session key or admin); resolve (not the arbiter)	FORBIDDEN
7002	DeadlineNotReached	The escrow deadline has not been reached.	release_after_window , refund_on_timeout , refund_after_resolution_deadline	DEADLINE_NOT_REACHED
7003	DeadlinePassed	The escrow deadline has passed.	pay_escrow , mark_delivered , dispute , resolve	DEADLINE_PASSED
7004	ArbiterRequired	This escrow has no named arbiter.	dispute , resolve	INVALID_TRANSITION

Code	Name	Message	Raised by	API code
7005	QuoteMismatch	The escrow quote does not match the signed payment.	pay_escrow	INVALID_INPUT
7006	TokenAccountMismatch	The escrow token account or its balance is invalid.	accept , release_after_window , resolve , refund_on_timeout , refund_after_resolution_deadline	INVALID_INPUT
7007	MerchantCannotArbitrate	The merchant cannot arbitrate its own escrow.	pay_escrow	INVALID_INPUT

C.2 API error codes

The API answers every failure with `{ error: { code, message, retryable, details? }, meta }`. `code` is one of the 26 values of `ErrorCode` in the shared contract (see §11.1). `retryable` is true for HTTP 429, for 5xx responses other than `CONFIGURATION_REQUIRED`, and for an expired preparation (`details.prepareAgain`, to be prepared again under a new key); a program refusal is never retryable, because the same bytes against the same state fail the same way. When a program error causes the failure, the API reads the `Error Code` log line of the simulation, maps the name with the table in C.1, puts the original name in `details.programCode`, and maps any name without an entry to `INVALID_TRANSITION`. During preparation the refusal is returned inside the preview as `simulation.allowed: false` with that error; at submission it is an HTTP 409, whatever status the same code carries when the API itself raises it.

Code	HTTP	What it means to a client
INVALID_INPUT	400, 413, 415	The request, or the transaction it describes, is malformed: invalid JSON or fields, a body over 64 KiB, a non-JSON content type, or a program refusal of a proof, quote, amount or account. Fix the input
UNAUTHORIZED	401	No valid session, or the sign-in challenge is missing, expired, reused or wrongly signed. Sign in again
FORBIDDEN	403	The wallet lacks the identity this resource or action needs, the CSRF token is missing, or the program refused the signer or an account's owner
NOT_FOUND	404	The resource is not indexed; from the program, the bond has no pending withdrawal to finalize
CONFLICT	409	The request conflicts with current state: an expired preview, a changed policy revision, an intent or quote already consumed, a withdrawal already pending. Re-read state and prepare again
RATE_LIMITED	429	The per-client request limit was reached. Retry after the <code>Retry-After</code> delay
INTERNAL_ERROR	500, 503	Transport, database or RPC failure, or the worker has not yet committed the latest chain state. Retry; for a submission, retry the identical signed transaction and submission key
CONFIGURATION_REQUIRED	409, 500, 503	A required setting is missing: the protocol is not initialized or indexed, the RPC belongs to another cluster, escrow configuration is missing, a passkey policy has no registered credential, or the program found a configuration invalid. An operator must act; retrying does not help
SIMULATION_ONLY	409	The prepared transaction comes from fixture mode and can be previewed but never signed or submitted

Code	HTTP	What it means to a client
VAULT_FROZEN	409	The vault is frozen, or a frozen vault or a running unfreeze tail still locks the operator's collateral
WINDOW_BUDGET_EXCEEDED	409	The payment would exceed the vault's or the merchant's window budget. Wait for the window to roll over or lower the amount
PER_TRANSACTION_CAP_EXCEEDED	409	The amount exceeds the merchant's per-transaction cap, or a voucher's delta exceeds the line limit
QUOTE_REQUIRED	409	The merchant rule requires a merchant-signed quote
APPROVAL_REQUIRED	409	The amount is above the co-sign threshold. Obtain an approval in a separate transaction first
APPROVAL_EXPIRED	409	The approval's expiry has passed. Request a new one
APPROVAL_USED	409	The approval was already consumed
VOUCHER_EXPIRED	409	The voucher or its line had expired at the time the program judged it
VOUCHER_REPLAY	409	The voucher does not advance the line's nonce and cumulative floors, or a filed claim's. Use the newest voucher
INSUFFICIENT_FUNDS	409	The vault, the bond, the float allowance or the freeze reserve cannot fund the action
DEADLINE_NOT_REACHED	409	A waiting period is still running: a decrease, key or guardian notice, a claim window, an escrow deadline, an approval's validity or a checkpoint's protection
DEADLINE_PASSED	409	The window for the action has closed. An expired preparation or blockhash carries <code>details.prepareAgain: true</code> : prepare the action again
INVALID_TRANSACTION	409	The action does not fit the current state: an active default, an escrow in another state, a missing arbiter, a guardian change outside the rotation instructions, or an unmapped program refusal
COVERAGE_TOO_LOW	409	The action would leave the bond below the operator's minimum coverage
WITHDRAWAL_COOLING_DOWN	409	The bond withdrawal's unlock time has not arrived
IDEMPOTENCY_CONFLICT	409	The idempotency key was already used for a different request or a different signed transaction
INDEXER_BEHIND	—	Part of the shared code set, beside the optional <code>meta.indexerLagSlots</code> field. When the worker has not yet committed the chain state a request needs, the live API answers with a retryable 503 <code>INTERNAL_ERROR</code>

Appendix D. Glossary

Terms are alphabetical. Names are as in the program. Values marked *configuration* come from the devnet parameter file; all other durations are program constants.

\$BONDED — The protocol's token: 1,000,000,000 fixed supply, issued in one public launch on equal terms. Every fee the protocol keeps is for buying it on the market and burning it (\$10.7). It is never collateral and never decides a claim.

Admin key — The operator's key, `Operator.admin`. It creates vaults and lines, sets policy and merchant rules, moves collateral, rotates keys and unfreezes.

Agent — The AI system that spends from a vault; on-chain, its session key.

Approval — An `Approval` at `[b"approval", vault, intent_hash]` recording a co-signer's consent to one payment intent, merchant, amount and expiry under the current policy revision. The payment that consumes it closes it, as can anyone after its expiry or a policy change.

Arbiter — The key a quote names to resolve a disputed escrow, by release or refund, before `resolve_until`. Zero means none; it cannot be the quote's merchant.

Base units — The smallest USDC amount: 1 USDC = 1,000,000 base units. Every program amount is a `u64` of base units.

Bond — The operator's collateral, `Bond` at `[b"bond", operator]`, funded by `post_bond`. It backs the operator's lines and pays the claims of a default event.

Buyback — The only way out of the fee treasury: one transaction from the configured crank that claims the fees (`claim_buyback`), swaps them for \$BONDED through Jupiter, and burns everything the swap returned (`burn_buyback`).

Bonded-tab — The x402 scheme for tabs: a 402 response offers the next voucher's fields, and the client retries with the signed voucher.

bps — Basis points: 10,000 bps = 100%.

Claim — A `Claim` at `[b"claim", default_event, merchant]` holding a merchant's latest voucher in a default and its eligible amount. It closes when paid.

Claim window — The 259,200 s (72 h) after a default event opens (`CLAIM_WINDOW`). Claims are filed strictly before `window_end`; payouts run at or after it.

Co-signer — The key in `policy.cosigner`, Ed25519 or a `PasskeyCredential`, that approves program payments above `cosign_threshold`. Zero means none.

Coverage ratio — $[(\text{bond balance} - \text{pending withdrawal}) \times 10,000 \div \text{reserved_lines}]$, in bps. It is public.

Cumulative amount — The total a voucher states is owed on its line since it opened. It only increases.

Deadline — A quote's `deliver_by`. Delivery may be marked at or before it; after it, an unmarked escrow can be refunded by anyone.

Default event — A `DefaultEvent` at `[b"default", bond, sequence]`, opened by `settle_tab` when the vault cannot pay a voucher's delta or a frozen vault has passed its freeze grace.

Dispute window — The 172,800 s (48 h) after `mark_delivered` . The payer may dispute strictly before `dispute_until` if an arbiter is named; at or after it, anyone may release the funds.

Escrow — An `Escrow` at `[b"escrow", vault, quote_hash]` whose token account holds a quoted job's principal and fee until release or refund.

Escrow fee — `escrow_fee_bps` of the principal, reserved on top at funding, paid to the fee treasury on release and returned on refund. Configuration: 25 bps.

Fee treasury — The protocol's USDC account at `[b"fees"]` , receiving tab and escrow fees; its only outflow is the crank's buyback, which swaps the USDC for \$BONDED and burns it.

Float — The SPL delegate allowance on a vault's token account that the session key spends with a top-level `TransferChecked` .

Float cap — `policy.float_cap` , the largest allowance a refill approves; at most the window budget.

Freeze — The `freeze` instruction, by a guardian or the admin. It revokes the float, stops vault payments and refuses new exposure until the admin unfreezes.

Freeze checkpoint — A `FreezeCheckpoint` at `[b"freeze", vault, generation]` recording one freeze's fixed start time, and its end once unfrozen; vouchers are judged as of that start.

Freeze grace — `freeze_grace_seconds` : the frozen time within one freeze episode after which `settle_tab` against a frozen vault opens a default. At least 345,600 s (96 h); configuration: 345,600 s.

Guardian — A key, `Operator.guardian` or `policy.guardian` , that can only freeze. Replacing it takes 72 hours.

Keeper — Anyone submitting permissionless instructions, such as `refill_float` , `execute_default_payout` and `refund_on_timeout` .

Limit — A line's `limit` : the most a voucher may exceed the settled floor, and the amount reserved against the bond. Decreases take 72 hours.

Line — A `CreditLine` at `[b"line", bond, merchant]` letting one merchant serve one vault's agent on a tab until `expires_at` .

Merchant — A seller of services to agents, identified by its key.

Merchant rule — A `MerchantRule` at `[b"rule", vault, merchant]` allowlisting a merchant for program payments and escrow, with a per-transaction cap, window budget and quote requirement.

Minimum coverage — The bond's `minimum_coverage_bps` , at least 3,300, set by `create_operator` . Opening, increasing or renewing a line, and withdrawing collateral, are refused if coverage would fall below it.

Nonce — A voucher's sequence number; it must exceed the line's `maximum_nonce` .

Operator — The company running an agent; its `Operator` at `[b"operator", admin]` carries public counters.

Payment intent — The 121-byte description of one program payment. Its SHA-256, the intent hash, seeds its approval and receipt.

PDA — Program-derived address: derived from seeds and the program ID, with no private key; only the program signs for it.

Policy — A vault's `PolicyArgs` : session key, float cap, window length and budget, co-signer, co-sign threshold, guardian. Each `set_policy` increments `policy_revision` .

Pro-rata payout — When eligible claims exceed the pool, each receives $[\text{pool} \times \text{eligible} \div \sum \text{eligible}]$; the remainder stays in the bond.

Program payment — A `pay` transfer from a vault to an allowlisted merchant, checked by the merchant rule and window budgets.

Protocol authority — The program's upgrade authority, which sets fees within fixed maxima and configures the buyback's mint, crank and minimum.

Quote — A 169-byte merchant-signed message naming vault, merchant, amount, specification hash, deadline, arbiter, expiry, nonce and lane.

Recourse lock — While any of an operator's vaults is frozen, and until freeze grace plus 72 h after an unfreeze, the operator's bond withdrawals, line renewals and applying matured limit decreases are refused.

Refill — `refill_float` : books consumed float, then approves $\min(\text{float cap}, \text{window budget} - \text{window spent})$.

Reserved lines — `reserved_lines` , the sum of the limits a bond backs.

Session key — The agent's key: float delegate, signer of program payments and escrow funding, and a new line's first voucher key.

Settle — `settle_tab` : the merchant submits its latest voucher and the program pays its delta from the vault, less the tab fee, or opens a default when the vault is short.

Settled floor — $\max(\text{settled_cumulative}, \text{retired_cumulative})$; only a voucher's excess over it is owed.

Tab — The lane where a merchant serves on credit against a line and settles in arrears.

Tab fee — A line's `tab_fee_bps` , fixed at opening, deducted from each settlement. Configuration: 10 bps, waived.

Usage digest — A voucher's last 32 bytes, committing to the metered usage; the merchant middleware hashes the request digest, charge and units into it. The program does not interpret it.

Vault — A policy vault, `Vault` at `[b"vault", operator, agent_id]` , holding one agent's USDC under its policy.

Voucher — The 96-byte Ed25519-signed message: domain tag, line, cumulative amount, nonce, expiry, usage digest.

Voucher attestation — A `VoucherAttestation` recorded during a line-key rotation notice, capping the outgoing key's payable nonce and cumulative amount.

Voucher key — A line's `voucher_session_key` ; `rotate_line_key` replaces it after 72 hours' notice.

Window — A vault's spending period of `window_seconds` , advancing in whole periods from `window_started_at` .

Window budget — The most booked per window: `policy.window_budget` for float and program payments; a merchant rule's `window_budget` for that merchant.

Withdrawal cooldown — `unlock_at` = $\max(\text{request time}, \text{longest line expiry}) + 259,200 \text{ s} + \text{withdrawal_margin_seconds}$ (configuration: 86,400 s).

x402 — The HTTP 402 payment protocol; the float pays its exact scheme.

Appendix E. References

Numbers in brackets in the text refer to these sources.

- [6] Fireblocks, glossary: x402. <https://www.fireblocks.com/glossary/x402>
- [7] x402 specification, the `exact` scheme on SVM. https://github.com/coinbase/x402/blob/main/specs/schemes/exact/scheme_exact_svm.md
- [8] Crypto Briefing, 15 September 2026: Solana's share of x402 volume. <https://cryptobriefing.com/solana-x402-market-share-dominance/>
- [9] CoinAlertNews, 14 January 2026 (Artemis data): daily x402 volume and transactions. <https://coinalertnews.com/news/2026/01/14/solana-overtakes-base-x402-transactions>
- [10] Grafa, 17 February 2026 (Dune data): x402 facilitator shares on Solana. <https://grafa.com/en/news/crypto/solana-49-percent-x402-micropayments>
- [13] Cloudflare, x402 and the deferred payment scheme; Cloudflare x402 documentation. <https://blog.cloudflare.com/x402> · <https://developers.cloudflare.com/agents/tools/payments/x402/>
- [15] Squads, "Smart Account Program live on mainnet", 14 April 2026. <https://squads.xyz/blog/squads-smart-account-program-live-on-mainnet>
- [19] Cointelegraph, 13 May 2026: a prompt injection against an agent wallet. <https://cointelegraph.com/news/grok-bankr-wallet-free-nft-prompt-injection>
- [20] Cointelegraph, 23 February 2026: an agent's mistaken transfer. <https://cointelegraph.com/news/openai-employee-s-ai-agent-accidentally-sent-442k-to-beggar>
- [21] Cointelegraph: the AIXBT dashboard breach. <https://cointelegraph.com/news/hacker-breaches-ai-crypto-bot-aixbt-steals-55-eth>
- [27] Solana documentation, fees and compute budget. <https://solana.com/docs/core/fees>
- [28] Solana documentation, accounts and rent. <https://solana.com/docs/core/accounts>
- [29] Solana documentation, cross-program invocation. <https://solana.com/docs/core/cpi>
- [30] Solana documentation, programs and precompiles. <https://solana.com/docs/core/programs>
- [32] Solana mainnet RPC `getAccountInfo` on the USDC mint `EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v`, 15 October 2026: owner `TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA`.
- [35] IETF draft-sander-open-tabs-passkey-00. <https://datatracker.ietf.org/doc/draft-sander-open-tabs-passkey/>
- [38] Para, "Solana Passkeys". <https://blog.getpara.com/solana-passkeys/>