

Bonded litepaper

Policy vaults and slashable USDC bonds on Solana that let AI agents buy on credit: the mechanism in brief.

Version 1.1 · 23 October 2026 9 pages, A4

1. Credit for agents

AI agents already pay for what they use. The x402 protocol turns an HTTP 402 Payment Required answer into a payment: the agent signs a USDC transfer for each request, and the merchant serves the request once the payment lands. Every call is prepaid, one transfer per request.

Metered services don't sell that way to anyone else. Inference, data, search, RPC and compute providers bill their customers in arrears: usage accumulates, and an invoice settles it. Billing in arrears needs credit, and credit to an agent has had nothing behind it. An agent has no balance sheet, no legal identity and no collections process. A merchant that serves it on credit is trusting whoever runs it, with no way to collect if that trust is misplaced.

Bonded makes the credit collectable. It is a Solana program and a service around it with two devices:

- A **policy vault** for each agent: a USDC account whose spending rules the program enforces on every payment. The agent's key can spend only inside them.
- An operator's **bond**: the operator's own USDC, locked in the program, which pays a merchant's claim when a vault comes up short.

With those two, a merchant can extend a **credit line** to an agent and know, from the chain, both that the agent can't spend outside its policy and that a bond of a known size stands behind the line.

2. Three lanes

An agent pays in one of three lanes, each fitted to a kind of purchase:

Lane	For	Enforcement	What bounds the loss / pays the merchant
Float	open-web x402 calls to any merchant	a small USDC allowance delegated to the session key, refilled within the vault's window budget	the allowance at a time, and the window budget across refills
Tab	metered services billed in arrears	a credit line opened by the operator; the agent signs a cumulative voucher with each request	the operator's bond, if the vault is short at settlement
Escrow	jobs delivered later	payment bound to a merchant-signed quote, co-signed above a threshold when the vault names a co-signer	automatic refund if no delivery receipt arrives by the deadline

The roles are separate keys, and each can do one thing:

Key	Held by	Can
Admin	the operator	create vaults and set their policy, fund them, post and withdraw the bond, open lines

Key	Held by	Can
Session key	the agent	sign payments and vouchers inside its vault's policy
Co-signer	the operator, or a person	approve payments above the vault's threshold (Ed25519 or a passkey)
Merchant	the merchant	settle vouchers, file claims, mark escrow jobs delivered
Guardian	a monitor the operator chooses	freeze a vault, and nothing else
Keeper	anyone	run what's due: timeouts, default payouts

3. The vault

A vault holds an agent's USDC and its **policy**:

- `session_key` : the only key that may spend from the vault.
- `float_cap` : the allowance the session key may spend directly as an SPL Token delegate.
- `window_seconds` and `window_budget` : the most the session key may spend per window, across float spending, direct payments and escrow.
- `cosigner` and `cosign_threshold` : payments above the threshold need the co-signer's approval as well.
- `guardian` : an optional key that can freeze this vault.

Each merchant the agent pays directly sits on the vault's list with its own **merchant rule**: a per-payment cap, its own window budget, and whether a signed quote is required. The program checks a payment in a fixed order and refuses at the first rule it breaks:

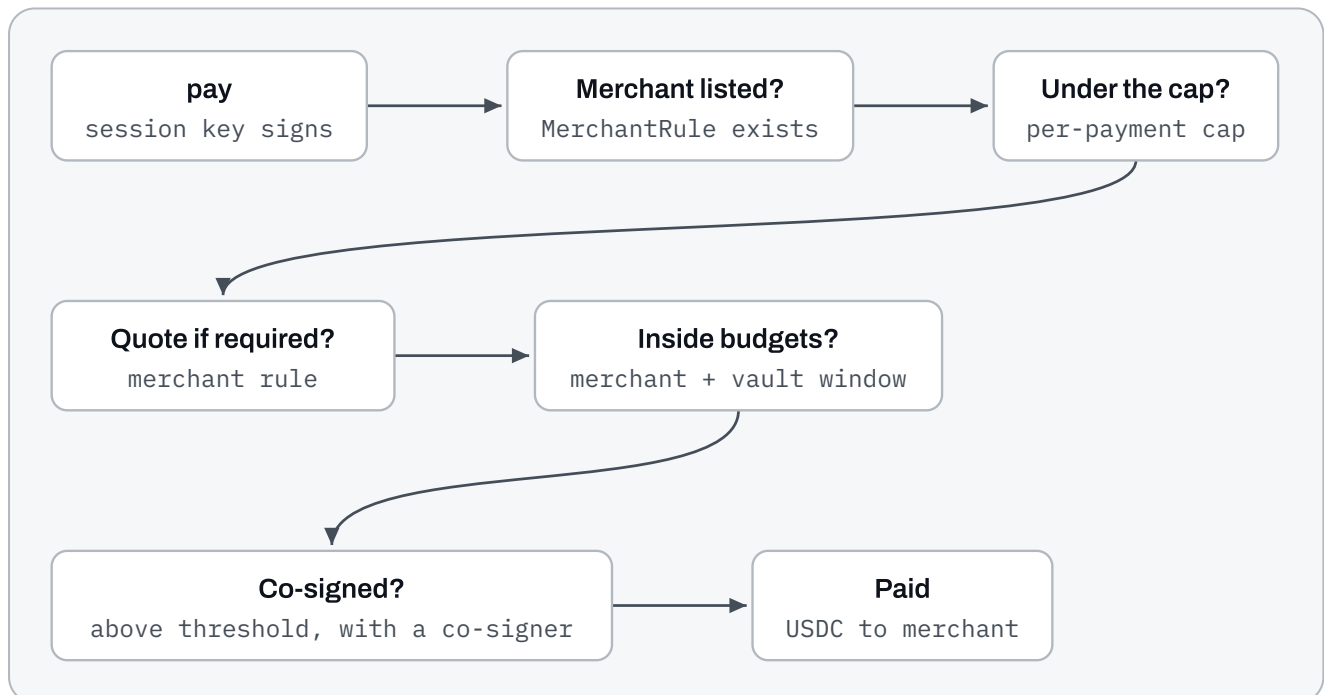


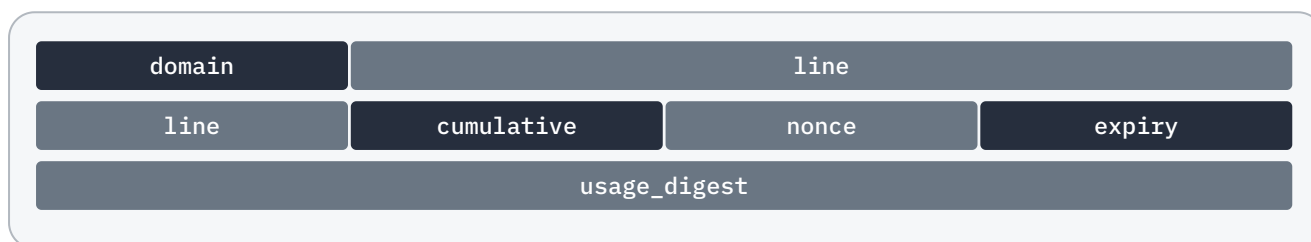
Figure 1. The checks the program runs on a direct payment, in order; each refusal names the rule.

Because every payment goes through these checks on chain, spending outside the policy can't happen, however the agent is instructed. What remains are bounded amounts: an injected instruction on the open web can cost at most the float allowance at a time and the window budget across refills; a merchant on the list can receive at most its cap and budget; a tab can accrue at most its line's limit.

The operator can **freeze** a vault at once, from the admin key or the guardian key. A frozen vault refuses every payment by the session key, and its float allowance is revoked in the same instruction.

4. Tabs and vouchers

A tab is a credit line in use. With each request, the agent signs a **voucher**: 96 bytes that state the cumulative amount owed on the line so far.



96 bytes · one cell per byte

Offset	Bytes	Field	Meaning
0	8	domain	first 8 bytes of sha256("Bonded voucher v2\0" program id genesis hash)
8	32	line	the credit line's address
40	8	cumulative	total vouched on the line, USDC base units
48	8	nonce	strictly increasing per line
56	8	expiry	unix seconds; never past the line's expiry
64	32	usage_digest	digest of the request, its price and units

Figure 2. The voucher the agent signs with each request, 96 bytes, integers little-endian.

The domain tag binds the voucher to one deployment of the program on one network, so a voucher signed for a test network can't be settled anywhere else. The usage digest commits to exactly what was served: the request's URL, method, body and key, the charge and the units.

The amount is cumulative, so **only the latest voucher counts**. It supersedes every one before it, and the merchant holds one voucher per line, not one per request. The merchant checks each voucher as it arrives (signature, counters, line limit, expiry, digest), serves the request, and settles the latest voucher before it expires (30 minutes by default) and before the line does:

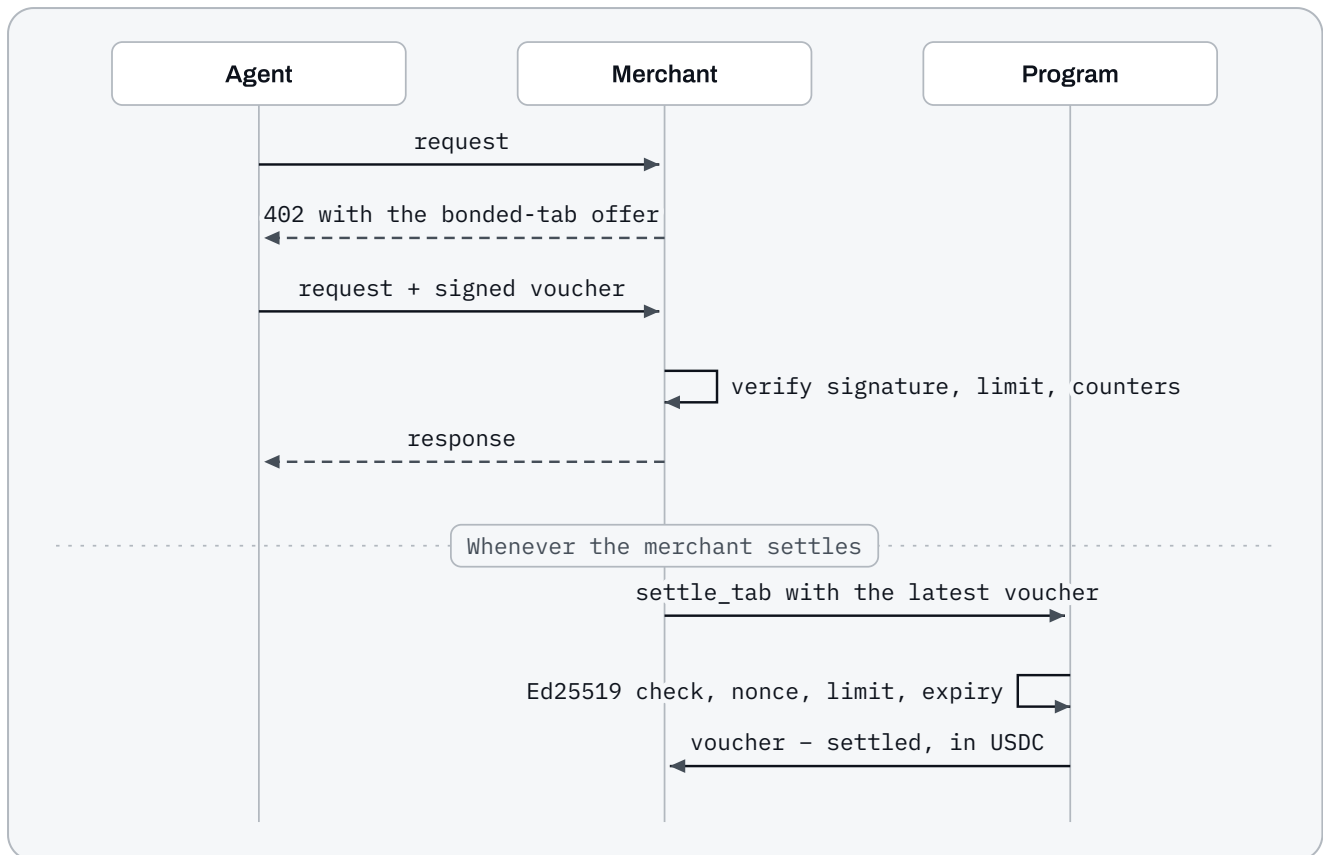


Figure 3. A tab: the agent signs a voucher with each request; the merchant settles the latest one.

`settle_tab` carries the voucher in Solana's Ed25519 signature-verification instruction, and the program reads the verified message from the instructions sysvar. Measured on the built program, a settlement is 894 bytes of the 1,232-byte transaction limit and 61,044 compute units. Its base fee counts two signatures, the merchant's and the voucher's Ed25519 check: 10,000 lamports, however many requests the voucher covers.

5. Bonds, lines and coverage

An operator posts one **bond** and opens lines from it. Each line reserves its limit against the bond. The bond's **coverage ratio** is public:

$$\text{coverage_bps} = \lfloor (\text{bond.balance} - \text{pending withdrawal}) \times 10\,000 \div \sum \text{line.limit} \rfloor$$

<code>bond.balance</code>	the USDC in the bond, base units
<code>pending withdrawal</code>	what the operator has asked to withdraw, if anything
<code>$\sum$ line.limit</code>	the limits of every line the bond backs

An operator chooses its minimum coverage when it creates its operator account: 100 % unless it asks for less, and never below the program's floor of 33 %. At 100 %, the bond must hold at least what its lines could owe. A line that would take coverage below the operator's minimum is refused with `COVERAGE_T00_LOW`. During the

beta a single bond holds at most \$25,000 and all bonds together at most \$250,000.

Two rules keep a bond in place while merchants depend on it:

- **Line decreases wait.** Raising a line's limit applies at once; lowering it takes effect 72 hours after the request, so a merchant already serving the agent has time to settle.
- **Withdrawals cool down.** A bond withdrawal unlocks only after the longest line's expiry, plus the 72-hour claim window, plus a safety margin; what remains must still cover the lines at the operator's minimum. An operator can't leave ahead of a default.

Merchants read the bond, the line and the operator's record (tabs settled, defaults) before serving an agent; all of it is on chain, and the app and API show it.

6. Defaults and claims

When a merchant settles a voucher and the vault holds less than `voucher - settled`, nothing moves from the vault. The program opens a **default event** on the operator's bond and starts a 72-hour **claim window**. Every merchant with a line on that bond may file its latest voucher as a claim within the window. After it, the bond pays each claim:

$$\text{eligible} = \min(\text{voucher.cumulative} - \text{settled}, \text{line.limit})$$

$$\text{payout} = \text{eligible} \text{ if } \sum \text{eligible} \leq \text{bond.balance}$$

$$\text{payout} = \lfloor \text{bond.balance} \times \text{eligible} \div \sum \text{eligible} \rfloor \text{ otherwise}$$

`eligible` what a claim may be paid, capped by its line's limit

`settled` the higher of the line's settled total and the voucher total an earlier paid claim retired

$\sum \text{eligible}$ the sum over every claim filed in the window

When the bond covers every claim, each is paid in full. When it doesn't, each claimant receives the same share of what it's owed, and the rounding remainder stays in the bond. Worked example: one bond of \$900 backs five lines; the vault is short, and all five merchants file.

Merchant	Line limit	Latest voucher	Settled	Eligible	Paid
Inference API	400.00	1,880.00	1,500.00	380.00	300.000000
Data API	300.00	950.00	600.00	300.00	236.842105
RPC provider	300.00	420.00	300.00	120.00	94.736842
GPU rental	250.00	610.00	360.00	250.00	197.368421
Search API	250.00	290.00	200.00	90.00	71.052631
Total				1,140.00	899.999999

Claims total \$1,140 against a \$900 bond, so each merchant receives 78.9 % of its eligible amount; one base unit stays in the bond. The data API's eligible amount stops at its line's limit: \$350 was vouched past the last settlement, and the bond pays up to the \$300 the operator committed.

Claims resolve from signatures and timestamps alone: no vote, no adjudicator. The keeper, or anyone, executes each payout once the window has closed.

7. Escrow

For a job that's delivered later, the merchant signs a **quote**: the price, a hash of the specification, a deadline, an expiry for the quote itself, and optionally an arbiter. The agent's vault funds it with `pay_escrow`; payments above the vault's threshold need the co-signer here too. The price and a 25 bps fee move into the job's escrow account, and the job moves through its states by instruction and deadline:

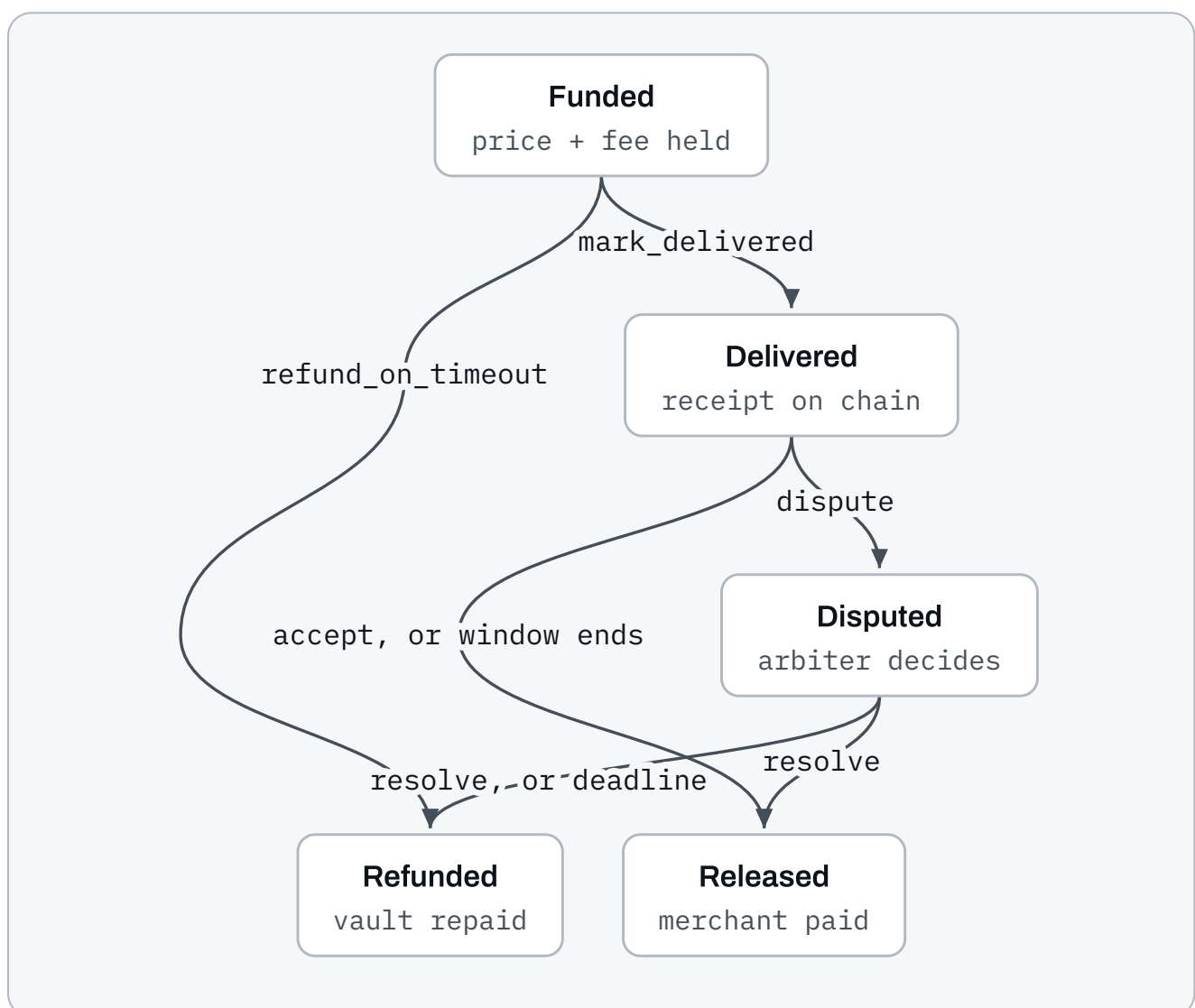


Figure 4. An escrow job's states, and the instruction that moves it from each one.

- **Delivery.** Before the deadline, the merchant marks the job delivered with the hash of the deliverable. If the deadline passes first, anyone can refund the vault.
- **Acceptance.** The agent accepts, or the 48-hour dispute window passes, and the price goes to the merchant, the fee to the protocol.

- **Dispute.** If the quote names an arbiter, the agent can dispute within the window. The arbiter releases or refunds; if it doesn't act before its resolve deadline, the job refunds.

8. Fees and \$BONDED

Fee	Rate	Paid by
Tab settlement	10 bps of the amount settled	the merchant, from the settlement
Escrow	25 bps of the price	the vault, reserved at funding, earned at release

The tab settlement fee is waived during the beta; each line keeps the rate it was opened with. Fees round down to the base unit and go to a program-owned USDC account. Float payments, bonds, lines, claims and payouts carry no Bonded fee; every transaction pays Solana's 5,000 lamports per signature and a small priority fee.

Bonded's token is **\$BONDED**. Its supply is fixed at 1,000,000,000, and all of it went to one public launch on equal terms: no presale, team, investor, airdrop or reserved allocation, and no vesting. Every fee the protocol keeps buys \$BONDED on the market and burns it, so the burn grows with the tabs and escrow jobs that settle through Bonded. The fee account has one way out: a transaction from the protocol's crank that claims the fees (`claim_buyback`), swaps them for \$BONDED through Jupiter, and burns everything the swap returned (`burn_buyback`). The program refuses a claim without that burn after it, or a swap below its quoted minimum. Bonds, settlements and claims stay in USDC: \$BONDED is never collateral and never decides a claim.

9. The system

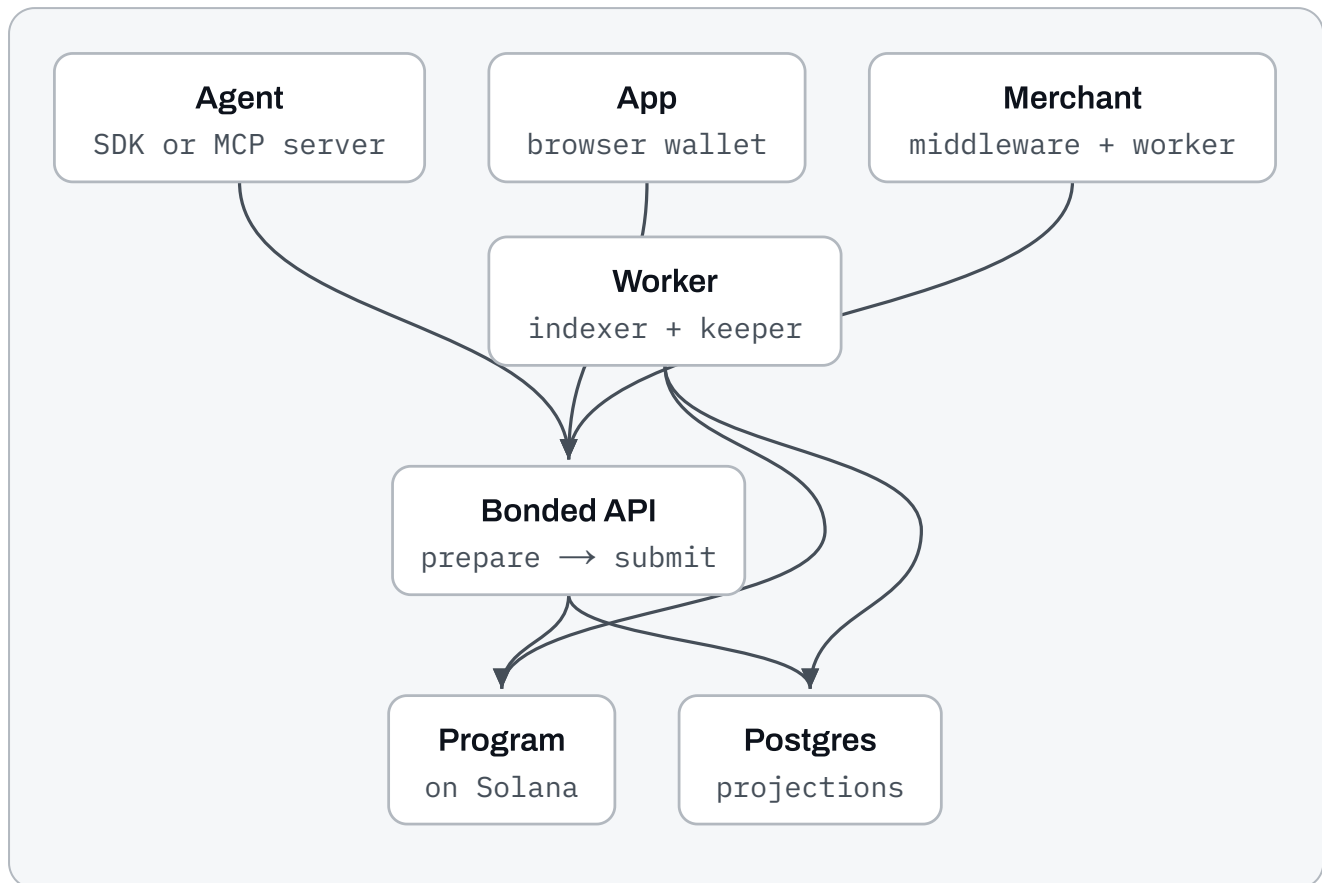


Figure 5. Where each piece runs: three clients, the API between them and the program, the worker beside it.

- **The program** holds every rule: vault policies, lines, bonds, defaults, escrow. It's the only place money moves.
- **The API** prepares each action as a transaction, simulates it, and after the user's wallet signs it, checks the signed transaction against the preparation and sends it. It holds no user key.
- **The worker** indexes the program into Postgres for the app and the API, and runs the keeper: timeouts and payouts that are due.
- **The merchant middleware** sits in front of a merchant's service: it answers with the tab offer, checks vouchers, meters requests, and runs the settlement and claims worker with the merchant's key.
- **The agent SDK and the MCP server** hold the agent's session key. They pin the program and the network independently of the API and refuse to sign for anything else; the MCP server adds per-call and window ceilings the model can't change.
- **The app** is where operators set up vaults, bonds and lines, and merchants see their lines, settlements, claims and escrow jobs, each action signed in a browser wallet.

10. Using it

- Docs: connecting a wallet; running agents as an operator; selling to agents as a merchant; building an agent with the SDK or the MCP server; the API.
- The app: the operator's setup and overview, and the merchant's view.

- The whitepaper: every account, instruction, formula and parameter, the off-chain services, security design and deployment.