



Sway whitepaper

Sealed brand rebates for AI shopping agents: the protocol, the program and the services, end to end.

Edition 1 · October 2026 · about 226 minutes to read

Contents

1. Introduction

- 1.1 Agents as buyers
- 1.2 Paying for conversions
- 1.3 Why both sides stay sealed
- 1.4 Why Solana and Arcium
- 1.5 One pass through Sway
- 1.6 The first deployment
- 1.7 How this paper is organized

2. System overview

- 2.1 Actors
- 2.2 Components
- 2.3 Identifiers and keys
- 2.4 One conversion, end to end
- 2.5 Intent outcomes
- 2.6 Proof tiers
- 2.7 Where data lives

3. Protocol parameters

- 3.1 One parameter set
- 3.2 Units
- 3.3 Parameter table
- 3.4 Constants compiled into the program
- 3.5 Deployment configuration
- 3.6 Policies and per-account parameters

4. Sealed purchase intents

- 4.1 The plaintext intent
- 4.2 Packing into field elements
- 4.3 Key agreement and the Rescue cipher
- 4.4 The nonce and the purchase deadline
- 4.5 Posting the intent
- 4.6 Retry rules
- 4.7 Closing the intent

5. Offers, campaigns and the book

- 5.1 Campaigns
- 5.2 Category shards and the book
- 5.3 The offer plaintext and its packing
- 5.4 Browser encryption
- 5.5 Submitting the offer

5.6 MXE ingest and the rebate cap

5.7 Offer states

6. Matching inside the MXE

6.1 Interface

6.2 Fixed-shape evaluation

6.3 Eligibility

6.4 The score

6.5 Top-three selection

6.6 First-price rebate

6.7 The sealed result: codec v2

6.8 The callback

6.9 Computation lifecycle

6.10 Worked example

7. Vouchers, receipts and verification

7.1 From sealed result to a chosen slot

7.2 The voucher

7.3 Checkout

7.4 The T2 merchant receipt

7.5 Receipt signers and signer epochs

7.6 Ed25519 verification on chain

7.7 record_conversion

7.8 verify_conversion inside the MXE

7.9 The verification callback

7.10 Reservation, cancellation and requeue

7.11 Conversion states

7.12 The production path: certify, then record

8. Escrow, lock, refunds and claims

8.1 The claims vault

8.2 The lock

8.3 Refund notices

8.4 Claim keys

8.5 The claim authorization

8.6 Transaction fees and rent

8.7 Purchase deadline, settlement deadline and grace

8.8 Closing finished accounts

9. Fees and settlement

9.1 What the brand pays

9.2 Rounding in base units

- 9.3 Campaign accounting
- 9.4 Caps in settlement
- 9.5 Worked examples
- 9.6 When each party is paid
- 9.7 The cashback split
- 9.8 SWAY: buyback and burn

10. Cashback fallback and the affiliate oracle

- 10.1 When an intent falls back
- 10.2 The public cashback table
- 10.3 Tracking links
- 10.4 Signed postbacks
- 10.5 Oracle batches and per-event outcomes
- 10.6 T3 escrow on chain
- 10.7 Reversals and refunds
- 10.8 Claim and the 70/10/20 split

11. Merchant bonds and disputes

- 11.1 Registration and receipt signers
- 11.2 Posting the bond and activation
- 11.3 What the merchant status gates
- 11.4 Liability windows
- 11.5 Opening a dispute
- 11.6 The dispute window anchor
- 11.7 Resolution and bond compensation

12. The on-chain program

- 12.1 Compiled constants
- 12.2 Account map
- 12.3 Account layouts
- 12.4 Token accounts and fund movement
- 12.5 Instructions
- 12.6 Signature introspection
- 12.7 How the program pins Config
- 12.8 State machines
- 12.9 Instruction-to-flow mapping
- 12.10 Transaction composition, size and lookup tables

13. Off-chain services

- 13.1 The API and the worker
- 13.2 Request pipeline
- 13.3 Authentication

- 13.4 Idempotency keys
- 13.5 The relay outbox
- 13.6 Durable jobs, locks and reconcilers
- 13.7 The database model
- 13.8 The MCP package

14. Privacy model

- 14.1 What each party learns
- 14.2 The shopper's agent
- 14.3 Losing brands
- 14.4 The winning brand
- 14.5 Sway's API, worker and relayer
- 14.6 The MXE nodes
- 14.7 What is public on chain
- 14.8 Linking intents and claims
- 14.9 Data minimization in the API
- 14.10 The cashback path

15. Security design

- 15.1 Domain separation
- 15.2 Binding to program and accounts
- 15.3 Replay and cross-use prevention
- 15.4 Key custody
- 15.5 Wallet transaction validation
- 15.6 Idempotent operations
- 15.7 Finality
- 15.8 Bounded worker scans
- 15.9 HTTP boundary

16. Infrastructure and deployment

- 16.1 Service layout
- 16.2 From nothing to a running deployment
- 16.3 Keys and where they live
- 16.4 Reading the chain economically
- 16.5 Waiting for the program
- 16.6 Upgrades

17. Appendices

- A. Byte layouts
- B. API reference
- C. Error codes
- D. Glossary

1. Introduction

Sway is an offer exchange for AI shopping agents. An agent posts its user's purchase intent in encrypted form. Brands keep sealed USDC rebate offers in a category book. An Arcium multi-party computation scores every offer against the intent and returns the best three, sealed to a key that only the user's agent holds. When the user buys and the merchant signs a receipt, the winning brand's rebate is escrowed on Solana through the merchant's return window and then claimed to any address the user chooses. This chapter explains why the system exists, summarizes how one intent moves through it, and maps the rest of the paper.

1.1 Agents as buyers

When software buys on someone's behalf, there is no results page for a person to scan and no ad slot for a brand to fill. The attention that advertising used to buy is absent at the moment that matters, so brand money has to attach to the purchase decision itself. The large platforms are already moving it there inside their own systems. Google's Direct Offers lets retailers push discounts to shoppers in AI Mode whom Google judges ready to buy, and OpenAI began testing sponsored units in ChatGPT in February 2026. In both cases the platform sees every query and every bid, and it decides what the shopper is shown.

Sway is built for the agents outside those systems: open-source assistants, wallet agents, browser agents and vertical shopping bots. These agents have no advertising sales team, and an agent that quietly sells its rankings loses the users who chose it for its judgment. Perplexity dropped its ad strategy in February 2026 to protect trust in its answers; an independent agent has even less room to experiment with paid placement. The clean way for such an agent to accept brand money is to pass it to its user as a price reduction and to take a disclosed fee for finding it.

Checkout is moving to agent protocols such as ACP, UCP and AP2, on card rails such as Visa Intelligent Commerce. Cashback, the established form of brand-funded price reduction, was built for browsers: extensions place affiliate cookies, and that model has a record of abuse. In December 2024 Honey was accused of swapping affiliate cookies at checkout, and it lost about 8M Chrome users by the end of 2025. An API checkout carries no cookie and no click for attribution to attach to. Sway replaces the cookie with a voucher bound to one intent and a receipt signed by the merchant.

1.2 Paying for conversions

A brand cannot buy an impression from an agent, but it can make a specific purchase cheaper. Sway therefore sells conversions rather than attention. A brand bids a rebate for a purchase intent, and it pays only when the purchase has happened, the merchant has attested it, and the lock covering the merchant's return window has ended without a refund. Until a conversion is verified, the money stays in the brand's own campaign vault, and unspent funds remain withdrawable (\$5).

Brand-funded, conversion-verified rebates are an established model. Rakuten advertises over 17M members, and Cardlytics runs card-linked offers through banks that reach about 215M shoppers by

its own count. These models depend on browser cookies or bank data. Sway provides a version that an agent calls directly and that settles in USDC, with neither the user nor the brand revealing private inputs to the operator.

The payment rule is first-price: a winning brand pays exactly the rebate it offered, and the user receives all of it. Sway's fees are added on top of the rebate rather than taken out of it. A brand-direct conversion costs the brand the rebate plus 15% of the rebate, of which 10% goes to Sway's treasury and 5% to the integrator, the developer of the agent that found the offer. The treasury's whole share buys \$SWAY, Sway's fixed-supply, fair-launch token, on the market and burns it (\$9.8). For example, a \$120 pair of running shoes with a \$15 rebate costs the brand \$17.25: \$15 to the user, \$1.50 to Sway and \$0.75 to the integrator. §9 specifies the fee basis, the rounding, the transfers and the SWAY buyback and burn.

A second-price rule would pay the user the runner-up's rebate, moving surplus from users to brands, and would reveal the clearing price, a rival's bid, to the winner. Its usual purpose is to make truthful bidding safe when bidders cannot trust the auctioneer. In Sway the bids are evaluated inside a computation that no single party can read or alter, so nobody can peek at bids or insert shill bids, and pay-as-offered keeps the whole rebate with the user.

1.3 Why both sides stay sealed

A purchase intent is a shopping profile. In the first vertical, running and outdoor footwear and gear, it carries a shoe size, a width, a maximum price, the brands the user already owns, and per-brand bonuses such as "a pair from brand X is worth \$10 more to me". Broadcasting that to every brand that might bid would give each of them a targeting record whether or not it wins.

An offer is a pricing strategy. It carries the product price, the rebate and the targeting rule: a size range, the supported widths, the region, and whether the offer is reserved for customers new to the brand. Publishing it to agents means publishing it to competitors and coupon scrapers, and a brand that has to publish its deepest discount has a reason not to offer one.

An operator that matched plaintext intents against plaintext offers would see every user's profile and every brand's discount strategy, the position of a conflicted auctioneer. In Sway, intents are encrypted on the user's device and offers in the brand's browser, both to the public key of an Arcium MXE (multi-party execution environment). The MXE evaluates eligibility and score over secret shares and returns only the top three, sealed to a per-intent key held by the user's agent. Sway's API, its relay, the public chain and the brands handle only ciphertext. §14 states precisely what each party learns.

Multi-party computation takes longer than plaintext matching. Real-time display bidding could not absorb that latency, but an agent assembling a purchase tolerates a wait measured in seconds, and the protocol bounds it: a queued matching computation expires after 180 Solana slots, the agent may retry once, and a second expiry sends the intent to the cashback fallback.

1.4 Why Solana and Arcium

Arcium's network uses Solana for orchestration, payments and computation callbacks, and its Mainnet Alpha launched in February 2026; that dependency fixes the chain. Solana also fits the settlement profile: rebates of a few dollars and integrator fees under a dollar need transaction fees far below a cent, and merchant receipts are Ed25519 signatures that the program checks through the Ed25519 precompile and the instructions sysvar (§7). Settlement uses plain SPL USDC with six decimals.

1.5 One pass through Sway

The following summary follows one brand-direct intent from a standing offer to a claim. Each step is specified in a later chapter.

1. A brand funds a campaign vault and submits an offer encrypted in its browser. The `ingest_offer` computation checks the sealed rebate against the campaign's public cap and writes the rule, re-encrypted to the MXE, into one of the 32 slots of the category book (§5).
2. The shopper's agent encrypts the intent with a fresh x25519 key created for this intent alone, adds a fresh Ed25519 claim key, and sends both public keys and the ciphertext through the Sway API; the relayer posts them with `post_intent` (§4).
3. The `match_intent` computation scores each eligible offer as $\text{brand_bonus} + \text{rebate} - \text{price}$, keeps the top three, and seals them to the intent's x25519 key, with a second copy encrypted to the MXE for verification (§6).
4. The agent decrypts the top three on the device, the user chooses, and the agent buys through the merchant's normal checkout with a voucher naming the intent and the book slot (§7).
5. The merchant signs a 128-byte receipt. `verify_conversion` confirms inside MPC that the slot, price and rebate match the sealed top three; then one transaction records the receipt through `record_conversion`, checks it, and moves the rebate and fees from the campaign vault into escrow (§7).
6. The program moves the rebate and fees into the claims vault until `unlock_at`, the end of the campaign lock. A signed refund notice inside the lock returns them to the brand (§8).
7. After `unlock_at`, the per-intent claim key signs a claim naming any destination. The program pays the rebate there and both fees to the treasury and the integrator in one instruction (§8, §9).

When no brand offer fits, or when matching expires twice, the intent ends in the cashback fallback state (§10). On that path the agent buys through a tracking link whose sub-ID is the random intent identifier, and the user receives 70% of the affiliate commission once Sway's affiliate oracle has been paid; the integrator receives 10% and Sway 20%. The path turns on when the deployment connects an affiliate network. Merchant bonds back disputes in which a merchant issues no receipt, reports the wrong amount, or refunds a conversion that was not returned (§11).

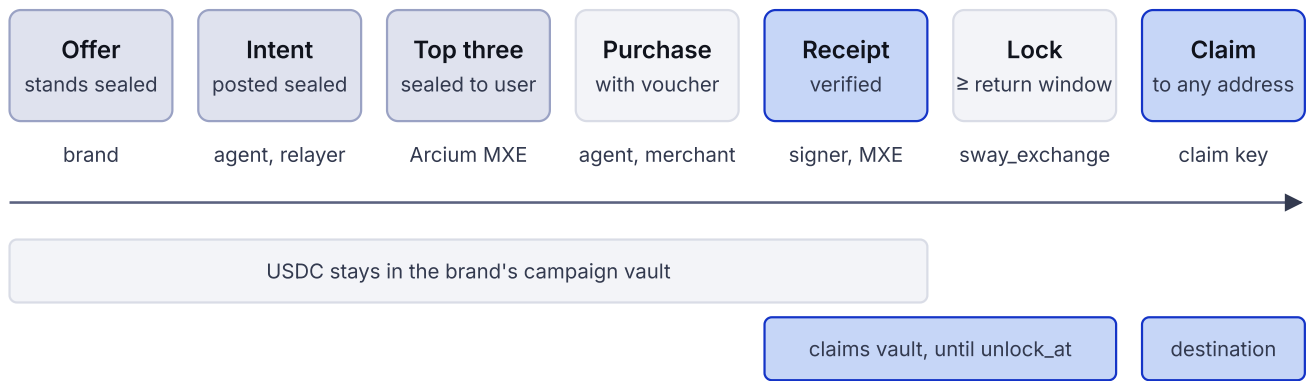


Figure 1.1 — The phases of one brand-direct intent, the party that acts in each, and where the brand's USDC sits. The first three phases handle only ciphertext.

1.6 The first deployment

The v0 parameter set (§3) targets running and outdoor footwear and gear for shoppers in the US, with Shopify direct-to-consumer brands as the first merchants. It caps a rebate at 50 USDC per conversion and a campaign's net funding at 5,000 USDC, accepts T2 merchant receipts as proof of purchase, and enables T3 affiliate commissions for a deployment that connects an affiliate network. The deployer key holds the programs' upgrade authority, and the Config admin is the wallet the deployment names, handed over at bootstrap (§12.7). Every value flow is denominated in USDC: brands fund vaults and bonds in USDC, users and integrators are paid in USDC, and Sway's fees accrue in USDC.

1.7 How this paper is organized

§2 gives the system overview: actors, components, one conversion end to end, the proof tiers and where data lives. §3 lists every protocol parameter with its unit and enforcement point. §4 to §7 follow the matching path: sealed intents, offers and the book, matching inside the MXE, and vouchers, receipts and verification. §8 to §11 cover settlement: escrow and claims, fee arithmetic, the cashback fallback and its oracle, and merchant bonds and disputes. §12 and §13 specify the on-chain program and the off-chain services. §14 and §15 present the privacy model and the security design, §16 the infrastructure and deployment, and §17 the byte layouts, API reference, errors and glossary.

2. System overview

Sway consists of three parts that run in different places. The `sway_exchange` program on Solana holds escrow, books, intents and conversions, and it enforces every cap, deadline and signature. Three Arcis circuits run in an Arcium MXE and are the only place where encrypted intents and offers are evaluated. An off-chain service, the Sway API and its worker, authenticates agents and brands, stores ciphertext, queues computations and relays transactions. Agents reach the service through REST or the `@sway/mcp` tools; brands reach it through a console and an offer SDK in the browser. This chapter introduces the actors and components, follows one conversion end to end, describes the two proof tiers used in v0, and maps where each piece of data lives.

2.1 Actors

Actor	Holds	Does
Shopper	Nothing on chain until a claim, and then only a destination address	States what they want to buy and chooses among the offers the agent shows
Agent	The plaintext intent; a per-intent x25519 secret key and Ed25519 claim secret key; the decrypted top three; checkout evidence	Encrypts intents on the device, posts them with a signed agent key, decrypts results, buys with a voucher, signs claims
Integrator	An authority key, a USDC payout wallet, and the agent keys it registers	Registers agent keys with scopes (<code>intent:write</code> , <code>intent:read</code> , <code>purchase:report</code> , <code>cashback:link</code>); receives the integrator fee on its agents' conversions
Brand (merchant)	A merchant wallet, a receipt-signing key, plaintext offer rules, campaign funds	Registers, posts a bond, funds campaign vaults, submits encrypted offers, signs receipts and refund notices
Relayer	Sway's fee-payer key, also stored as <code>Config.buyback_authority</code>	Signs and submits transactions for intents, receipts, refunds and claims as fee payer; pays SOL fees and account rent; signs the SWAY buyback
Arcium MXE	The cluster's secret key material, held as shares by its Arx nodes	Runs <code>ingest_offer</code> , <code>match_intent</code> and <code>verify_conversion</code> on encrypted inputs and returns their callbacks
Sway API and worker	PostgreSQL state: ciphertext, public keys, hashes, signed messages, mirrors of finalized accounts	Authenticates requests, prepares transactions, queues computations, reconciles finalized chain state
<code>sway_exchange</code> program	<code>Config</code> , books, intents, conversions, vaults and bonds	Enforces caps, deadlines and signatures; moves USDC
Affiliate oracle	The key stored in <code>OracleAuthority.oracle</code> and a USDC token account	Posts paid T3 commissions into escrow and reverses them inside the network's return window
Config admin	The wallet stored in <code>Config.admin</code> , which the deployment names and which differs from the programs' upgrade authority	Activates and suspends merchants, releases bonds and resolves disputes against bonds, signing each transaction on the operator page; rotates configuration

In v0 the brand that bids and the merchant that sells are one registered entity. The brand identifier hashed into an offer is the merchant's identifier, and `ingest_offer` rejects an offer whose sealed brand differs from the merchant that submitted it. The receipt signer is the merchant's attestor for T2 conversions; the affiliate oracle is the attestor for T3 conversions.

2.2 Components

On the shopper's side, @sway/mcp runs locally as a stdio MCP server with five tools: `find_offers`, `get_voucher`, `report_purchase`, `claimable` and `claim`. It packs and encrypts intents with @sway/crypto, keeps per-intent secrets in the operating system's keychain, and signs every HTTP request with the agent key. Hosted agents use the REST API and the TypeScript client directly and encrypt in their own backend. The protocol requires only that the plaintext intent and the per-intent secrets stay with the agent.

On the brand's side, the console and offer SDK encrypt the offer rule in the browser to the MXE's public key. The merchant wallet signs transactions that the API prepares; the API then submits the exact signed bytes. The receipt signer is a separate Ed25519 key that the merchant holds itself, in the brand console's browser profile or on its own server, and that never reaches Sway.

The Sway API is an HTTP service backed by PostgreSQL. Every transport object rejects unexpected fields, so a plaintext attribute sent by mistake is refused rather than stored. Every mutation carries an Idempotency-Key, and a retry with the same key and body returns the original result. The worker runs durable jobs from a leased queue: posting intents, matching, retries, offer ingest, receipt recording and verification, refunds, claims, cashback batches and the SWAY buyback. Its relay outbox commits signed transaction bytes and their signature before broadcasting and follows each signature to finality, so a lost RPC response never produces a second, different transaction. §13 describes these services in detail.

The Arcium MXE runs three circuits written in Arcis. Their source is generated from the same TypeScript eligibility predicates and score expression that the agent and the site use (`scoring.ts`), and the generated source embeds the SHA-256 of that scoring file. `ingest_offer` admits an offer into a book slot, `match_intent` produces the sealed top three, and `verify_conversion` checks a claimed purchase against the sealed result and reveals one bit (§6, §7).

The program is an Anchor program. Its accounts are summarized below and specified in §12.

Account	Holds	Created by
Config	Admin, USDC mint, MXE authority, claims vault, treasury, vault floor, minimum bond, dispute seconds, fee rates, rounding policy, purchase window, settlement grace, SWAY mint, buyback authority	initialize, signed by the program's upgrade authority
Merchant, bond account, ReceiptSignerEpoch	Receipt signer and its validity history, return window, status, pending conversions, liability horizon	register_merchant, rotate_receipt_signer
Campaign and vault	Per-conversion cap, daily cap, lock days, funding, spend and reservation counters	create_campaign
OfferBook	32 slots, each a campaign address, an Enc<Mxe> rule, a nonce, an element count and a version; the active mask; a revision	create_book; slots written by ingest_offer_callback
Intent, IntentBook, InputReplay	Encrypted intent, public keys, deadlines, sealed results; the book captured at posting; the one-use input guard	post_intent
Conversion with OrderGuard or OracleEvent	Terms, fees, proof tier, unlock time, status; the order or network-event replay guard	record_conversion, oracle_post_conversion

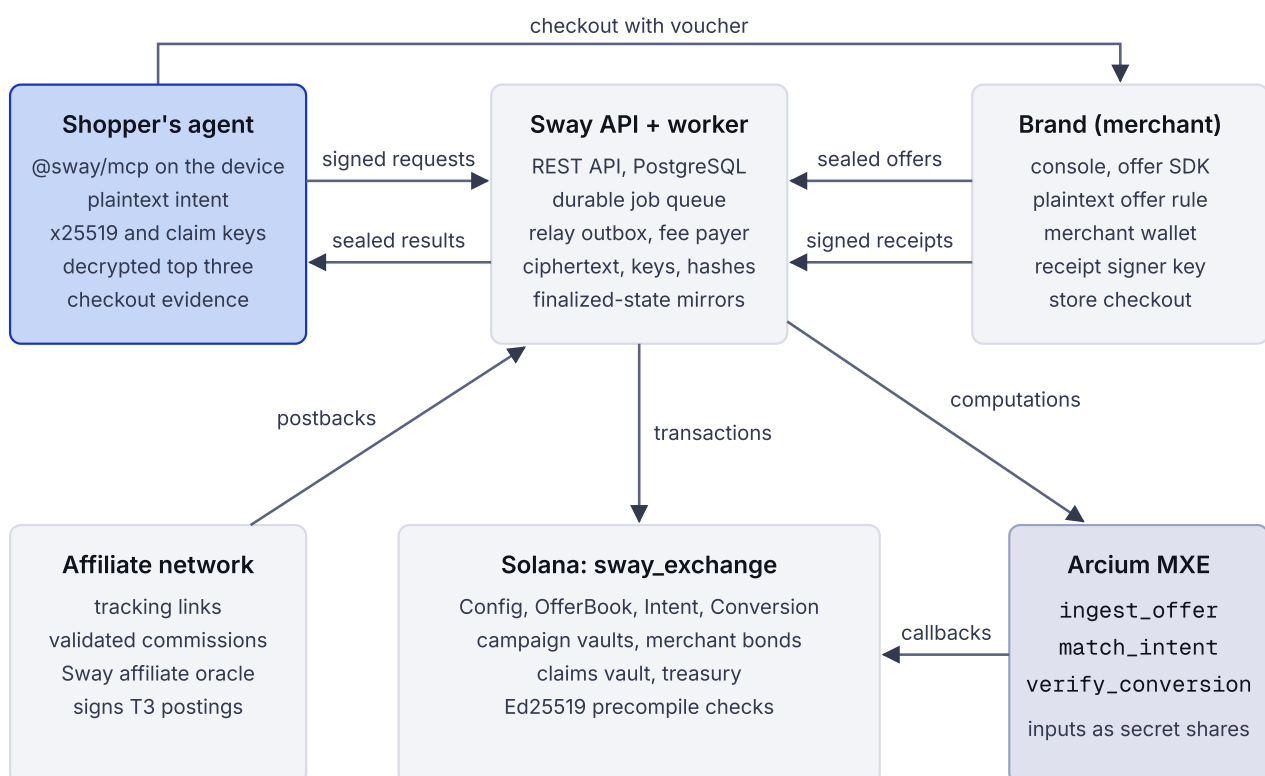


Figure 2.1 — Components and what crosses between them. Only the shopper's agent and the brand hold plaintext; the MXE evaluates inputs as secret shares.

2.3 Identifiers and keys

Each intent has three identifiers. The agent generates a random opaque `intentId` for REST, which also serves as the affiliate sub-ID in the cashback path. The API allocates an internal UUID whose 16 bytes seed the `Intent` account address (seed "intent"), and receipts sign that account address, never a textual identifier. Each intent also has two fresh keypairs created on the device: an x25519 key that both encrypts the intent and receives the result, and an Ed25519 claim key that authorizes the payout. The API refuses a claim key that already belongs to another intent, and the program refuses an intent whose input key differs from its result key (`InputKeyMismatch`). A brand's merchant wallet, receipt signer and campaign are long-lived; an agent's keys are per integrator and revocable.

2.4 One conversion, end to end

Figure 2.2 follows a brand-direct (T2) conversion. The numbers below match the figure.

1. The agent reads `GET /v1/encryption` for the MXE public key, the chain clock, the purchase window and the admitted shards. It packs the intent, generates the per-intent keys, chooses a purchase deadline no later than chain time plus `Config.purchase_window_secs`, writes that deadline into the first eight bytes of the 16-byte nonce, and encrypts. It sends `POST /v1/intents` with `intentId`, `shard`, `encryptedIntent`, `claimPublicKey` and `x25519PublicKeyBase64`, signed under the `SwayAgent` scheme. The API checks that the sender key equals the result key and that the deadline lies inside the window, reserves the (`sender key`, `nonce`) pair against reuse, and stores the ciphertext.
2. The worker captures the shard's book together with its effective active mask, and the relay submits `post_intent`. The program creates the `Intent`, an `IntentBook` holding the 32 campaign addresses and active mask at that moment, and an `InputReplay` account seeded by the x25519 key and nonce. It records `input_commitment`, a SHA-256 digest over the intent address, the hash of the whole book account, the key, nonce, element count, ciphertext, claim key and integrator, and sets `expires_at_slot` to the current slot plus 180.
3. The worker queues `match_intent` with the intent ciphertext, the 32 `Enc<Mxe>` offers of the captured book, and public inputs: the active mask and the campaign, brand, category and intent hashes.
4. The MXE's callback arrives as `match_intent_callback`, signed by the authority recorded in `Config.mxe`. The program accepts it only for the current attempt, before `expires_at_slot`, and with an `input_commitment` equal to the stored one. It stores the result sealed to the agent, the MXE-encrypted copy, and `snapshot_hash`, a SHA-256 digest of the copy.
5. The agent polls `GET /v1/intents/:id` and receives the sealed top three as opaque ciphertext elements. It also reads `GET /v1/intents/:id/book`, the complete public map of the 32 captured slots.
6. On the device, the agent decrypts the result, checks that its intent and category hashes match this intent, resolves each entry's campaign and brand hashes against the full slot map, and presents the offers. The user chooses one.

7. The agent requests a voucher for the chosen slot. The voucher carries the intent account address and the string `sway:{intentId}:{slot}` to attach at checkout.
8. The agent buys through the merchant's normal checkout with the voucher attached. It can file the order digest and an evidence digest with `report_purchase`; the checkout receipt itself stays on the device.
9. The merchant's signer signs the 128-byte receipt: domain `SWAYRCP1`, intent account, slot, order hash, purchase amount, timestamp, signer epoch and program ID. Anyone may relay it to `POST /v1/receipts`.
10. The worker queues `verify_conversion` through the Arcium coordinator with the receipt, the merchant's proposed rebate and the MXE copy of the top three; the claimed slot, purchase amount, rebate and identity hashes are its public inputs.
11. The callback reveals one bit, authenticated by the cluster's signature. A true bit becomes a certificate held by the coordinator. A false bit rejects the proposal, and nothing is written to the financial program.
12. The relayer submits an Ed25519 precompile instruction followed by `record_certified`, which calls `record_conversion` and the verification callback in one transaction (§7.12). The program checks that the captured book assigns the slot to this campaign, that the snapshot hash matches, that the receipt names this intent, slot, program and the signer epoch valid at its timestamp, and that the timestamp falls inside the intent's purchase window. It adds the fees to the rebate, creates the `Conversion` and an `OrderGuard` for the merchant's order hash, moves the total from the campaign vault to the claims vault, sets `unlock_at` to the verification time plus `lock_days × 86400` seconds, and extends the merchant's liability horizon.
13. The funds stay escrowed until `unlock_at`. If the product is returned, a refund notice (domain `SWAYREF1`) signed by the merchant's current receipt signer returns the full debit to the campaign vault.
14. After `unlock_at`, the agent signs a 128-byte claim message (domain `SWAYCLM1`) binding the program, the conversion account, the destination wallet, an expiry and a nonce, and sends it to `POST /v1/claims` with a fresh claim-key authorization.
15. The relayer submits an Ed25519 precompile instruction followed by `claim` and pays the fee, so the claim key and the destination need no SOL.
16. The program pays the rebate to the destination's token account, Sway's fee to the treasury and the integrator's fee to a token account owned by the integrator wallet recorded on the intent, all in one instruction.

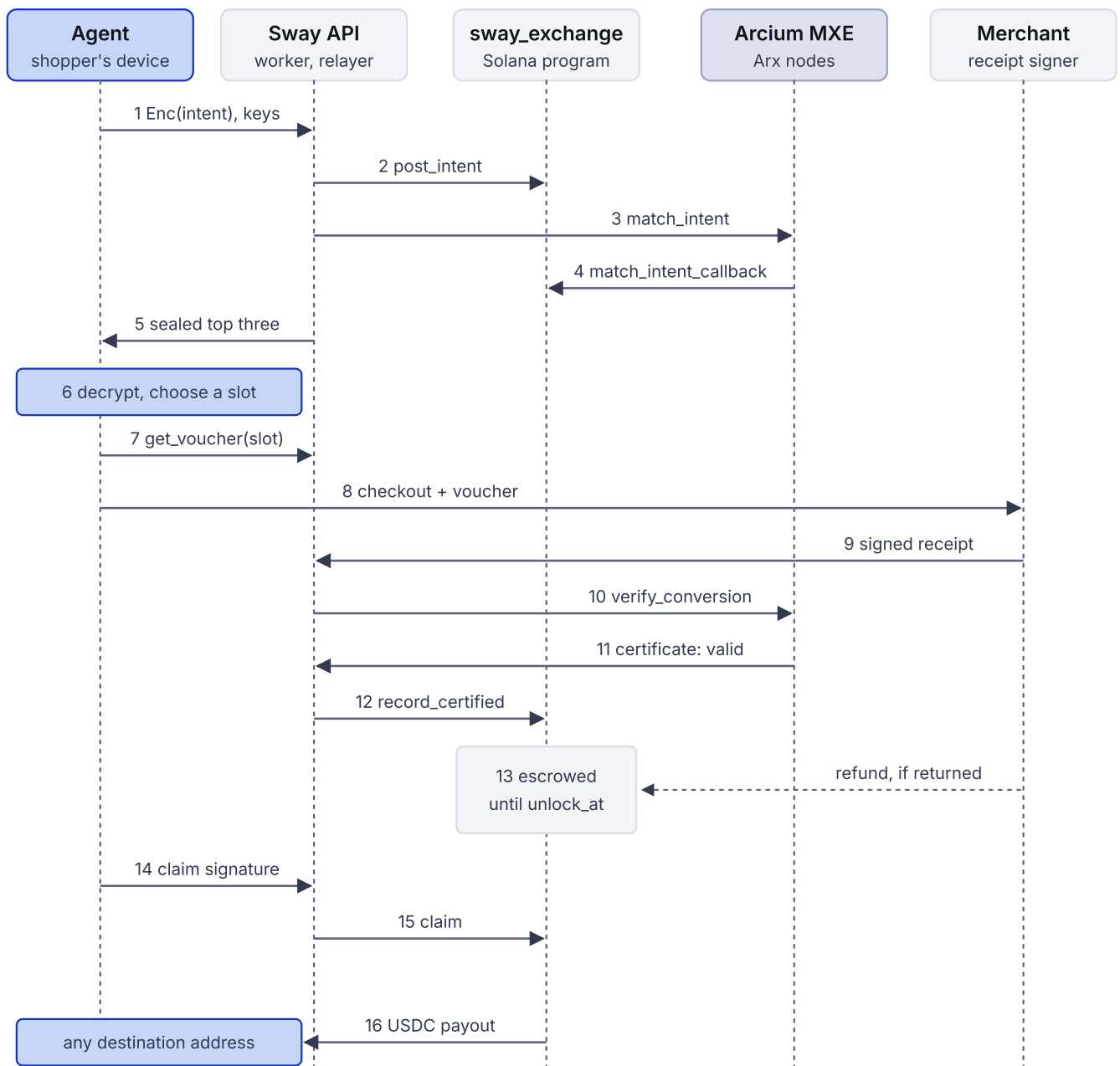


Figure 2.2 — A brand-direct (T2) conversion from intent to claim. Step 16 pays the rebate to the chosen address and, in the same instruction, Sway's fee to the treasury and the integrator's fee to the integrator.

2.5 Intent outcomes

An intent account moves through four on-chain states: pending (0), matched (1), fallback (2) and settled (3). The API reports a richer view of the same lifecycle: pending, matched, expired, cashback_fallback and failed.

If no callback lands before `expires_at_slot`, the computation has expired. The agent may ask for one retry; `retry_intent` moves the intent from attempt 1 to attempt 2 and opens a new 180-slot window. A second expiry sets the on-chain status to fallback, and the API reports `cashback_fallback` with reason `COMPUTATION_EXPIRED_TWICE`. An intent whose sealed result holds no offer also falls back to cashback, and the API names that reason `NO_ELIGIBLE_OFFERS`. The on-chain result is a fixed-size

ciphertext in every case, so the chain does not show whether the top three was empty. An agent that wants only cashback can register its intent with `cashbackOnly: true`; the intent still carries ciphertext and a claim key, but no matching computation is queued (reason `CASHBACK_ONLY`).

Two deadlines bound every intent. The purchase deadline, committed in the nonce, ends the period in which a receipt timestamp or an affiliate purchase timestamp is accepted. The settlement deadline, `purchase_deadline + Config.settlement_grace_secs`, ends the period in which a receipt or a paid commission can still be recorded. After the settlement deadline, and once no conversion is open, `close_intent` returns the rent of the Intent, IntentBook and InputReplay accounts to the original payer.

2.6 Proof tiers

A conversion is a claim that a purchase happened. Sway names the kinds of evidence it accepts as proof tiers; the parameter set defines three and enables two in v0.

Tier	Evidence	Attested by	v0
T1	USDC payment to the merchant's registered account in the same transaction, checked by instruction introspection	Solana itself	No
T2	128-byte receipt signed with Ed25519	The merchant's receipt signer	Yes
T3	Network-validated affiliate commission	The Sway affiliate oracle	Yes

T2, merchant receipts. T2 settles brand-direct offers. The receipt is the merchant's statement that order `order_hash` was paid at amount `amount` at time `ts` against slot `slot` of intent `intent`. The program verifies the signature through the Ed25519 precompile against the signer key that the merchant's `ReceiptSignerEpoch` records for the receipt's epoch, and it rejects a receipt whose timestamp falls outside that epoch's validity interval. The `OrderGuard` account, seeded by merchant and order hash, makes each order usable once. MPC verification then binds the receipt to the sealed top three: the slot must be one of the three entries, and the receipt amount and the committed rebate must equal that entry's price and rebate. Funds come from the campaign vault and unlock after the campaign's `lock_days`. Merchant honesty is backed by the bond (§11).

T3, affiliate commissions. T3 settles the cashback fallback and needs no MPC. The agent requests a tracking link whose sub-ID is its random intent identifier. When the affiliate network validates the purchase and pays the commission, the affiliate oracle signs `oracle_post_conversion`, which transfers the entire received commission from the oracle's token account into the claims vault. The program splits it 20% to Sway and 10% to the integrator, each rounded down, and assigns the remainder to the user; that remainder must not exceed `REBATE_CAP`, 50 USDC. The conversion occupies the reserved slot value 255, its replay guard is an `OracleEvent` account seeded by the SHA-256 identity of the network event, and its lock ends `network_return_days` after posting. Inside

that lock, `oracle_refund_conversion` returns the full commission to the oracle's source account. \$10 covers the oracle and the rate table.

Both tiers end in the same `claim` instruction, authorized by the same per-intent claim key.

2.7 Where data lives

Data	Where it lives	Form
Plaintext intent: size, width, maximum price, brand bonuses, existing brands	Agent device	Plaintext; never transmitted
Per-intent x25519 secret and claim secret	Agent device, operating-system keychain	Secret keys
Encrypted intent	Intent.encrypted on chain; PostgreSQL Intent.ciphertext	Eight 32-byte Rescue elements and a 16-byte nonce, Enc<Shared>
Plaintext offer rule	Brand browser	Plaintext; never transmitted
Submitted offer	OfferSubmission on chain; PostgreSQL Offer.ciphertext	Enc<Shared> under a fresh brand key
Booked offer	OfferBook slot on chain; PostgreSQL revision snapshot	Enc<Mxe>
Public book map: slot, campaign, merchant, active flag	OfferBook, IntentBook on chain; API book endpoint	Public
Shopper's top three	Intent.user_result on chain; PostgreSQL	Sealed to the per-intent x25519 key
MXE copy of the top three	Intent.mxe_result and snapshot_hash on chain	Enc<Mxe> and its SHA-256 digest
Order reference	Merchant systems	Only its SHA-256 order_hash leaves
Signed receipt	Relayed in the transaction; PostgreSQL; Conversion.receipt_hash on chain	128 signed bytes; on chain only their hash
Checkout evidence	Agent device	Digest and optional opaque encrypted reference in PostgreSQL
Conversion terms: price, rebate, fees, unlock time	Conversion on chain; PostgreSQL mirror	Public
USDC	Campaign vault, then claims vault, then destination, treasury and integrator accounts	SPL token balances
MXE secret key	Arx nodes	Secret shares; never assembled in one place

Plaintext intents and offer rules never leave the device or the browser where they were written. The service and the chain store ciphertext and digests, and the terms of a conversion become public

when the conversion moves USDC. §14 examines each party's view in detail.

3. Protocol parameters

Every number that defines the protocol, from the fee rates to the size of a ciphertext element, is declared once and propagated mechanically to the components that enforce it. This chapter describes that propagation, the units in which values are expressed, and then lists every parameter with the place where it is checked. Later chapters refer back to these tables instead of restating values.

3.1 One parameter set

The protocol constants live in a single JSON-safe object, `PROTOCOL_PARAMETERS`, exported by the `@sway/contract` package from `protocol.ts`. The current set is version `0.4.0`. Each component reads its values from that object or from code generated from it:

- **Program.** `scripts/generate-contract.ts` writes `generated_protocol.rs`, which `lib.rs` includes at compile time. The file carries ten constants (§3.4) and a header forbidding manual edits, so the Rust program compiles the same caps, slot counts, expiry and fee rates that the TypeScript object declares.
- **Circuits.** `circuits/generate.ts` lowers the eligibility predicates and the score expression of `scoring.ts` into the Arcis source of `ingest_offer`, `match_intent` and `verify_conversion`, and it emits the wire structures from the field descriptors of `@sway/crypto`. The generated file records the SHA-256 of the scoring source (`f01fed5d...e023c6`). The generator fails on a rule it cannot lower, and its `--check` mode fails when the committed circuit source is stale. The book size, the top-three width and the tie-break are therefore fixed in the circuit by the same definitions that the agent uses to explain results.
- **API.** The request and response schemas take their bounds from the object: book slots `0` to `31`, ciphertext elements of exactly 32 bytes, nonces of exactly 16 bytes, at most eight intent elements, at most `topK × 3` result elements, at most `agentRetries + 1` computation attempts, the fixed cashback shares, and the 128-byte receipt. The service publishes the object at `GET /v1/protocol`, and the contract package ships it as `protocol.json`.
- **Documentation.** The site's facts module derives its percentages, caps and slot counts from the same object, so a published number cannot disagree with the deployed one.

A change to a protocol value is a change to `protocol.ts` followed by regeneration of the program constants and the circuit source.

3.2 Units

Money is expressed in USDC base units. USDC has six decimals, so one USDC is 1,000,000 base units, and 50 USDC is written `50000000`. Amounts cross the API as canonical unsigned decimal strings in the `u64` range, never as floating-point numbers; the amount schema accepts exactly the strings from `0` to `18446744073709551615`. PostgreSQL stores them as `Decimal(20,0)` with unsigned checks, TypeScript handles them as `bigint`, and the program uses `u64` with `u128` intermediates.

Fee rates are basis points out of 10,000. The program computes every fee with one function and rounds down to the base unit:

```
fee(amount, bps)      = floor(amount × bps / 10000)      (u128 intermediate)
T2 total_debit        = rebate + fee(rebate, 1000) + fee(rebate, 500)
T3 user_rebate        = commission - fee(commission, 2000) - fee(commission, 1000)
```

`Config.rounding_policy` must equal 1, floor to the base unit; `initialize` rejects any other value. In the brand-direct case the fees are added on top of the rebate, so rounding never reduces what the user receives. In the cashback case the user receives the remainder, so any fractional base unit left by the two floors goes to the user. For example, a 15 USDC rebate (15000000) produces a Sway fee of 1500000, an integrator fee of 750000 and a total campaign debit of 17250000.

Time uses three units. Computation lifetimes are counted in Solana slots. Deadlines, receipt timestamps and unlock times are signed 64-bit Unix seconds read from the Solana clock. Return windows and locks are whole days, converted at 86,400 seconds per day, and the daily campaign cap resets on the UTC day index `unix_timestamp div 86400`. Scores are signed: the circuit computes `brand_bonus + rebate - price` in 128-bit integers, and the TypeScript reference uses `bigint`.

3.3 Parameter table

Paths are relative to `PROTOCOL_PARAMETERS`. "Program" refers to `sway_exchange (lib.rs)`, "MXE" to the generated Arcis circuits, and "API" to the Sway service and its schemas.

Name in code	Value	Unit	Meaning	Where enforced
version	"0.4.0"	—	Version of the parameter set	API: GET /v1/protocol.js
settlement.currency	"USDC"	—	Asset of every vault, bond, rebate and fee	Program: Confirmed at buy token account constrained
settlement.decimals	6	decimals	Base-unit scale of amounts	Program: in requires min: 6 (WrongMint schema)
settlement.baseUnitsPerUsdc	"1000000"	base units	Base units in one USDC	API: amount program: u64
settlement.tokenProgram	"spl-token"	—	Token program for all transfers	Program: Program: Program: Token> and token CPIs
settlement.token2022	false	flag	Token-2022 extensions are not used	Program: all are SPL Tokens
settlement.token.ticker	"SWAY"	—	Ticker of Sway's token; settlement amounts stay in USDC (settlement.currency)	The mint in Config.sway. as SWAY_MINT
settlement.token.supply	"1000000000"	whole tokens	Total SWAY supply	Fixed at the
settlement.token.fixedSupply	true	flag	No further SWAY is minted; burns only lower the supply	Program: no mints SWAY; buyback_finance: supply to fall
settlement.token.launch	"fair"	—	The whole supply went to the public launch on equal terms	Launch term
settlement.token.feeShare	"buyback-burn"	—	Sway's fee share buys SWAY on the market and burns it	Program: buy the treasury' and buyback. a burn of at least (\$9.8)
auction.paymentRule	"first-price"	—	A winning brand pays the rebate it offered	MXE: verify requires the rebate to equal rebate; program that rebate

Name in code	Value	Unit	Meaning	Where enforced
<code>auction.score</code>	"brand_bonus + rebate - price"	USDC base units, signed	Value of an offer to the shopper	MXE: <code>match_reference_scoring.ts</code>
<code>auction.bookSlots</code>	32	slots	Offers per category book	Program: <code>BookOfferBook.ar</code> [<code>Enc<Mxe, 0-32</code>]; API: <code>slot</code>
<code>auction.topK</code>	3	offers	Offers sealed to the shopper per intent	MXE: <code>three-t</code> API: <code>sealed r</code> 9 elements
<code>auction.maxIntentEncryptedElements</code>	8	32-byte elements	Ciphertext budget of one intent	Program: <code>MAX_INPUT_EI</code> <code>post_intent</code> ; input schem <code>encryptField</code>
<code>auction.offerEncryptedElements</code>	8	32-byte elements	Ciphertext capacity of one book slot	Program: <code>OfferBook.ei</code> <code>slot</code> ; elemen <code>submit_offe</code> ; <code>ingest_offe</code>
<code>auction.encryptedElementBytes</code>	32	bytes	Size of one Rescue ciphertext element	Program: [<code>ut</code> elements; <code>Al</code> schema; <code>Pos</code> checks
<code>auction.nonceBytes</code>	16	bytes	Encryption nonce length	Program: [<code>ut</code> nonce scher
<code>auction.tieBreak</code>	"lower-slot-first"	—	Equal scores rank the lower slot first	MXE: <code>top-thr</code> network; <code>refi</code> <code>matchIntent</code>
<code>brandDirectFees.basis</code>	"rebate"	—	Brand-direct fees are computed on the rebate	Program: <code>rec</code> computes fe
<code>brandDirectFees.chargedOn</code>	"verified-conversion"	—	The vault is debited only after MPC verification	Program: <code>tra</code> <code>verify_conve</code> when valid
<code>brandDirectFees.totalBps</code>	1500	bps	Total brand-direct fee, 15% of the rebate	Sum of the t
<code>brandDirectFees.swayBps</code>	1000	bps	Sway's fee, 10% of the rebate	Program: <code>DIF</code> into <code>Config.s</code> to the treasu

Name in code	Value	Unit	Meaning	Where enforced
brandDirectFees.integratorBps	500	bps	Integrator's fee, 5% of the rebate	Program: DIRECT_INTEGRATOR Config: integratorBps paid to the integrator at
brandDirectFees.userReceivesFullOfferedRebate	true	flag	Fees are added on top of the rebate	Program: clientRebate rebate to the
cashbackSplit.userBps	7000	bps of commission	User's share of a T3 commission, 70%	Program: oracle_post assigns the rate schema
cashbackSplit.integratorBps	1000	bps of commission	Integrator's share, 10%	Program: CASHBACK_INTEGRATOR API: rate schema
cashbackSplit.swayBps	2000	bps of commission	Sway's share, 20%	Program: CASHBACK_SWAY three shares 10,000 (Configurati
caps.perConversionRebateUsdc	"50000000"	base units (50 USDC)	Maximum rebate of one conversion	Program: REBATE create_campaign record_conversion oracle_post MXE: ingest rebate above cap
caps.perCampaignUsdc	"5000000000"	base units (5,000 USDC)	Maximum net funding of a campaign and maximum daily cap	Program: CAMPAIGN fund_campaign create_campaign
locks.unit	"days"	—	Unit of return windows and locks	Program: LOCKS
locks.minimum	"merchant-return-window"	—	A lock is never shorter than the merchant's return window	Program: createLock requires lock return_window (LockTooShort)
locks.configuredPerCampaign	true	flag	Each campaign chooses its own lock	Program: Campaign.lock

Name in code	Value	Unit	Meaning	Where enforced
locks.cashbackRequiresCommissionReceived	true	flag	T3 escrow is funded only by commission already paid	Program: oracle_post, transfers the commission vault; API: or can be batch
computations.expirySlots	180	Solana slots	Lifetime of a queued computation	Program: QU expires_at_ after it fail (s retry_inten (NotExpired)
computations.agentRetries	1	retries	Retries after the first expiry	Program: re moves attempt at m
computations.fallback	"cashback"	—	Outcome after the second expiry	Program: int API: cashbac COMPUTATION.
proofTiers[0]	T1, v0: false	—	On-chain USDC payment checked by instruction introspection	Not enabled
proofTiers[1]	T2, v0: true	—	128-byte merchant receipt signed with Ed25519	Program: rec (proof_tier
proofTiers[2]	T3, v0: true	—	Network-validated commission posted by the affiliate oracle	Program: oracle_post, (proof_tier
receipt.bytes	128	bytes	Length of every signed settlement message: receipt, refund, claim	Program: [u arguments; \ requires a 12 message; Af schema
receipt.signature	"Ed25519"	—	Signature scheme of receipts, refunds and claims	Program: Ed precompile, the instructi
receipt.p256OnChainV0	false	flag	P-256 merchant keys serve as dispute evidence only	API: receipt stored, neve chain
scope.vertical	"running-and-outdoor-footwear-and-gear"	—	Product vertical of v0	API: shard al creation

Name in code	Value	Unit	Meaning	Where enforced
scope.regions	["US"]	—	Supported shopper and offer regions	Crypto: pack packOffer at MXE: region be set
scope.amazonSupported	false	flag	Amazon purchases are outside the v0 scope	Scope defini
requiredConfiguration.*	null (16 keys)	—	Deployment-specific values supplied at launch	\$3.5

3.4 Constants compiled into the program

The generator maps ten values to Rust constants. Their types are the narrowest that hold the value, and the program uses checked arithmetic around them.

Rust constant	Type	Value	Source path	Used by
REBATE_CAP	u64	50000000	caps.perConversionRebateUsdc	create_campaign, record_conversion, oracle_post_conversion
CAMPAIGN_CAP	u64	5000000000	caps.perCampaignUsdc	create_campaign (daily cap), fund_campaign (net funding)
QUEUE_SLOTS	u64	180	computations.expirySlots	post_intent, submit_offer, retry_intent
DIRECT_SWAY_BPS	u16	1000	brandDirectFees.swayBps	initialize copies it into Config.sway_bps
DIRECT_INTEGRATOR_BPS	u16	500	brandDirectFees.integratorBps	initialize copies it into Config.integrator_bps
CASHBACK_USER_BPS	u16	7000	cashbackSplit.userBps	oracle_post_conversion (sum check)
CASHBACK_INTEGRATOR_BPS	u16	1000	cashbackSplit.integratorBps	oracle_post_conversion
CASHBACK_SWAY_BPS	u16	2000	cashbackSplit.swayBps	oracle_post_conversion
BOOK_SLOTS	u8	32	auction.bookSlots	submit_offer, clear_book_slot, record_conversion
MAX_INPUT_ELEMENTS	u8	8	auction.maxIntentEncryptedElements	post_intent, submit_offer, ingest_offer_callback

A few further constants are fixed in code rather than in the parameter object, because they name formats rather than policy. The program's signed messages carry eight-byte domains: SWAYRCP1 for receipts, SWAYREF1 for refund notices and SWAYCLM1 for claims. T3 conversions occupy the reserved slot value 255, outside the book's range. The encoding package fixes MAX_BRAND_PREFERENCES = 10 brand entries per intent, encodes sizes in half steps ($\text{size} \times 2$ in 16 bits), and tags each wire structure with an eight-byte ASCII marker in the low 64 bits of its metadata field: SWAYINT1 for intents, SWAYOFF1 for offers and SWAYTOP2 for the top-three result. §4 and §17 give the byte layouts.

3.5 Deployment configuration

Sixteen keys in `requiredConfiguration` are deliberately `null`: they are deployment-specific, and `null` never stands for a default. The service refuses to run in live mode while a required policy is blank, and the program's `initialize` rejects a zero vault floor, minimum bond, dispute period, purchase window or settlement grace, and any rounding policy other than 1 (`ConfigurationRequired`). Thirteen of them land in the program or the service:

Key	Lands in	Effect
activeSlotVaultFloorUsdc	Config.vault_floor; service VAULT_FLOOR_USDC	A book slot is passed to matching as active only if its campaign vault holds at least this amount and its merchant is active
agentKeyRateLimit	API AGENT_RATE_LIMIT_REQUESTS, AGENT_RATE_LIMIT_WINDOW_SECONDS	Size and window of the per-agent-key request bucket
minimumMerchantBondUsdc	Config.minimum_bond	activate_merchant requires a bond balance at least this large (Underfunded)
disputeWindowSeconds	Config.dispute_seconds; API DISPUTE_WINDOW_SECONDS	Extends Merchant.liability_until past each unlock and bounds when disputes can be opened
relayerFeePolicy	API RELAYER_FEE_POLICY	How relayed SOL fees and computation costs are covered
feeRoundingPolicy	Config.rounding_policy; API FEE_ROUNDING_POLICY	Must be 1, floor to the base unit
affiliateNetwork	API AFFILIATE_NETWORK	The network whose commissions feed T3
affiliateCommissionRates	Public rate table, GET /v1/cashback/rates	Commission rate per merchant that tracking links are created against
usdcMint	Config.mint; build-time PINNED_USDC_MINT	initialize rejects any other mint
treasuryAddress	Config.treasury; service TREASURY_ADDRESS	The USDC token account at ["treasury", config] that receives Sway's fee at claim; its USDC leaves only through buyback_begin
affiliateOracleKeys	OracleAuthority.oracle; API ORACLE_PUBLIC_KEYS	The key allowed to post and reverse T3 conversions
programId	declare_id!; service SWAY_PROGRAM_ID	Bound into every receipt, refund notice and claim message
arciumCluster	Config.mxe	The MXE authority whose callbacks the program accepts

The SWAY buyback (§9.8) adds five settings of its own, outside requiredConfiguration:

Setting	Lands in	Effect
SWAY_MINT	Config.sway_mint, through deploy:init or update_config; API and worker	The plain SPL mint the buyback buys and burns. The worker runs the buyback only while it equals Config.sway_mint
BUYBACK_AUTHORITY	Config.buyback_authority, written by deploy:init as the relayer's public key	The only key buyback_begin and buyback_finish accept; the worker signs with RELAYER_KEYPAIR
BUYBACK_MIN_USDC	worker, default 10000000 (10 USDC)	Treasury balance below which no buyback runs
BUYBACK_INTERVAL_SECONDS	worker, default 3600	Shortest time between two buybacks
BUYBACK_SLIPPAGE_BPS	worker, default 100 (1%)	Slippage of the Jupiter quote, which sets min_burn

3.6 Policies and per-account parameters

Some values are neither protocol constants nor launch configuration. They are set per deployment in Config, or per merchant, campaign or oracle, and the program bounds each of them.

Parameter	Set by	Bound	Meaning
Config.purchase_window_secs	initialize	> 0	Longest allowed distance between posting an intent and its purchase deadline
Config.settlement_grace_secs	initialize	> 0	Time after the purchase deadline during which receipts and paid commissions can still be recorded
Merchant.return_window_days	register_merchant	u16	The merchant's return window, copied into each campaign
Campaign.lock_days	create_campaign	$\geq$ return_window_days	Lock applied to each verified conversion of the campaign
Campaign.per_conversion_cap	create_campaign	$0 < \text{cap} \leq \text{REBATE_CAP}$	Public cap; ingest_offer rejects a sealed rebate above it
Campaign.daily_cap	create_campaign	$0 < \text{cap} \leq \text{CAMPAIGN_CAP}$	Maximum total debit, rebate plus fees, per UTC day
OracleAuthority.network_return_days	initialize_oracle, rotate_oracle	u16	Lock applied to each T3 conversion

The purchase window and settlement grace interact with the intent nonce. The agent writes the purchase deadline into the first eight bytes of the intent's nonce, so the deadline is fixed by the ciphertext itself, and `post_intent` accepts it only when $\text{now} < \text{deadline} \leq \text{now} + \text{purchase_window_secs}$. The settlement deadline follows as $\text{deadline} + \text{settlement_grace_secs}$ and is stored on the intent, so a later change to `Config` never moves the deadlines of intents already posted. §4 describes the nonce format and §8 the effect of these deadlines on escrow.

4. Sealed purchase intents

An intent is the shopper's request as the agent submits it: one public category shard and a short vector of encrypted values that describe what the shopper wants to buy, the most it will pay, and how much it prefers particular brands. The agent builds the intent on the shopper's device, packs it into at most eight field elements, encrypts it to Arcium's MXE (the multi-party execution environment that runs Sway's circuits) with a key that exists for this intent only, and hands the ciphertext to the Sway relay, which records it on chain with `post_intent`. No step on this path carries the plaintext. This chapter follows an intent from its plaintext fields to its on-chain account and ends with the rules for retrying. Matching is the subject of §6; the voucher and receipt that follow a purchase are described in §7.

4.1 The plaintext intent

The plaintext type is `PlaintextIntent` in `packages/contract/src/scoring.ts`. It exists only on the device. Every REST schema rejects fields it does not declare (`additionalProperties: false`), so plaintext attributes cannot travel beside the ciphertext by accident.

Field	Type	Range and validation
<code>categoryShard</code>	string	Required. The public book the intent is matched against, for example <code>running-footwear</code> . On the wire it matches <code>^[a-z0-9]+(?:[._-][a-z0-9]+)*\$</code> , 1–80 characters.
<code>size</code>	number	Non-negative, in whole or half steps. Packed as <code>size × 2</code> in 16 bits, so <code>size × 2 ≤ 65,535</code> . Each category defines its own sizing convention.
<code>width</code>	narrow, standard or wide	Packed as index 0, 1 or 2.
<code>region</code>	string	Only US in v0 (<code>scope.regions</code>).
<code>maxPriceUsdc</code>	string	Canonical unsigned integer in USDC base units, <code>^(0 [1-9][0-9]*)\$</code> , at most 18,446,744,073,709,551,615 (u64).
<code>brandBonusesUsdc</code>	map of brand id to amount	The value the shopper adds to an offer from that brand. An absent brand has a bonus of zero.
<code>existingBrandIds</code>	string[]	Brands the shopper already buys from. Brands not listed are new to the shopper.

All money values are integers in USDC base units; one USDC is 1,000,000 base units (`settlement.decimals = 6`). Half sizes are the only fractional quantity, and the codec turns them into integers before encryption.

A bonus of 12000000 for a brand means an offer from that brand is worth 12 USDC more to this shopper than the same offer from an unlisted brand; §6.4 shows where it enters the score. The

existing-brand list serves the new-customer rule: a brand may restrict an offer to new customers, and the intent tells the computation, privately, whether this shopper is one.

4.2 Packing into field elements

Arcium encrypts vectors of elements of the field of order $p = 2^{255} - 19$, each serialized as 32 bytes. The codec in `packages/crypto/src/codec.ts` turns a `PlaintextIntent` into exactly eight such elements: the protocol's `maxIntentEncryptedElements` and the program's `MAX_INPUT_ELEMENTS`.

4.2.1 Identifier hashes

Strings never enter a circuit. Brand identifiers, the category shard, campaign identifiers and the intent's account address are reduced to 64-bit hashes by `hashIdentifier(value, domain)`:

```
id_hash = u64_le( SHA-256( "sway/" || domain || "/v1" || 0x00 || value )[0..
```

The domain is brand, category, campaign or intent, so one string hashes differently in each role. The value must be 1–256 characters long. The hash value zero is reserved because zero marks an empty position in every wire format; an identifier that hashes to zero is refused. For example, the category hash of `running-footwear` is `3046417618813319877` (`0x2a470caf06dbfac5`). It is unrelated to the 32-byte shard seed that addresses the book account on chain, which is the plain SHA-256 of the shard string (§5.2.1).

4.2.2 INTENT_FIELDS and the eight elements

`INTENT_FIELDS` declares 23 unsigned integers:

Logical field	Width	Count
metadata	128 bits	1
category	64 bits	1
maxPrice	64 bits	1
brandIds[0..9]	64 bits	10
bonuses[0..9]	64 bits	10

Together they hold 1,536 bits. Arcium's packer fills elements up to 214 bits each (`ARCIS_PACKING_SIZE`), below the field size, so every packed value is a valid cipher plaintext. Sway fixes the layout with `groupedPacker`: the first group is the header (metadata at bit 0, category at bit 128), every later group is three consecutive 64-bit words at bits 0, 64 and 128, and each group must pack into exactly one element. The result is one header element and seven triples, shown in Figure

4.1; e1 starts with `maxPrice`, e4 holds the last two brand hashes and the first bonus, and e7 the last three bonuses.

The circuit declares the same shape, `IntentCipher { header: Pack<HeaderWire>, data: [Pack<TripleWire>; 7] }`, and `decode_intent` reads the 21 words back in order: word 0 is `maxPrice`, words 1–10 are the brand hashes, words 11–20 the bonuses. Because the grouping is fixed, the encrypted layout never depends on how many brands the shopper listed, and the circuit's argument shape is constant.

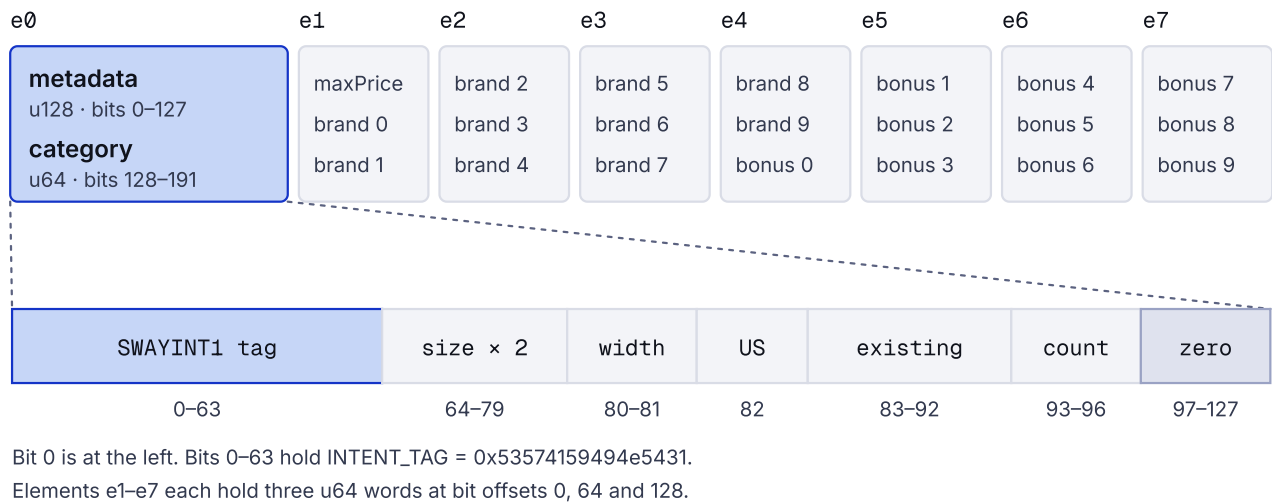


Figure 4.1 — The intent wire format. A header element carries the 128-bit metadata word and the category hash; seven triple elements carry the maximum price, ten brand hashes and ten bonuses. The lower strip expands the metadata word.

4.2.3 The metadata word

The 128-bit metadata field carries everything that is not a 64-bit quantity:

Bits	Content	Encoding
0–63	INTENT_TAG	0x53574159494e5431, the ASCII bytes SWAYINT1 read most significant first
64–79	size	size × 2, unsigned 16 bits
80–81	width	0 narrow, 1 standard, 2 wide
82	region	1 for US, the only accepted value in v0
83–92	existing-brand bits	bit 83 + i is set when preference slot i holds a brand the shopper already buys from
93–96	brand count	number of used preference slots, 0–10
97–127	reserved	zero

```
metadata = INTENT_TAG | (size·2) << 64 | width << 80 | 1 << 82 | existi
```

The tag is a type check that survives encryption. A ciphertext decrypted under the wrong key or with the wrong nonce yields field elements whose low 64 bits are not SWAYINT1, and every decoder rejects it.

4.2.4 Brand preference slots

An intent has ten brand preference slots (`MAX_BRAND_PREFERENCES = 10`). `packIntent` fills them deterministically:

1. Collect every brand with a non-zero bonus, add every brand in `existingBrandIds`, remove duplicates and sort the identifiers as strings. A brand with a zero bonus that the shopper does not already buy from is left out, because absence already means a zero bonus and a new customer.
2. Refuse more than ten brands.
3. Hash each identifier in the brand domain, and refuse the intent if two hashes collide.
4. For slot `i`, write the hash to `brandIds[i]`, the bonus (or zero) to `bonuses[i]`, and set metadata bit `83 + i` if the brand is in `existingBrandIds`.
5. Leave every unused slot zero with its existing bit clear, and write the number of used slots to bits 93–96.

The identifiers must be the ones offers are encrypted with, which are the merchants' canonical identifiers. The agent reads them, slot by slot, from the book map described in §4.5.5.

For example, the contract's sample intent (fictional brands) has size 9.5, width standard, a bonus of 12 USDC for `sample-kestrel-loop` and 6 USDC for `sample-bracken-pace`, and lists `sample-fell-and-fern` as an existing brand. Sorting gives the slot order `bracken-pace`, `fell-and-fern`, `kestrel-loop`, with bonuses `6000000`, `0` and `12000000` and the existing bit on slot 1:

```
size × 2 = 19   width = 1   region = 1   existing = 0b00000000010   count = 3
metadata = 0x6015001353574159494e5431
```

4.2.5 Canonical decoding

The same rules hold in reverse wherever an intent is decoded. `unpackIntent` in TypeScript and `valid_intent` in the circuit both require the tag in bits 0–63; zero in bits 97–127; the region bit set; a width index below 3; a count of at most 10; a non-zero category; non-zero, pairwise distinct brand hashes in the used slots; and a zero brand, bonus and existing bit in every unused slot. The TypeScript decoder also re-packs what it decoded and requires the result to equal the input elements bit for bit (`canonicalUnpack`), so exactly one element vector represents each intent. In the

circuit, an intent that fails `valid_intent` makes every offer ineligible, and the computation returns an empty top three.

4.3 Key agreement and the Rescue cipher

4.3.1 Keys on the device

For each intent the agent creates two keypairs: an x25519 keypair whose secret is 32 bytes from the platform's cryptographic random source (`createEncryptionKeypair`), and an Ed25519 claim keypair. The x25519 key encrypts the intent and later decrypts its result. The claim key authorizes the payout (§8). Neither serves a second intent: `encryptIntent` takes the per-intent secret as an explicit argument, the API refuses a claim key already bound to another intent (`REPLAY_DETECTED`), and the agent keeps both secrets in its device key store. `GET /v1/encryption` publishes the MXE's x25519 public key, the encoding name, the element budget and the chain's purchase window.

4.3.2 From shared secret to cipher key

The agent computes the x25519 shared secret $Z = x25519(s, pk_{mxe})$ from its per-intent secret s and the MXE public key. The MXE computes the same Z from its own secret and the agent's public key. Arcium's `RescueCipher` derives the cipher key with the Rescue-Prime hash, following the one-step key derivation of NIST SP 800-56C (Option 1) with a counter of 1 and the output length as fixed information:

```
key = RescuePrimeHash( [ 1, Z, 5 ] )
```

Here Z is the 32-byte secret read as a little-endian integer, and 5 is the cipher's block size m .

4.3.3 Counter mode over the field

The Rescue cipher runs in counter mode over $p = 2^{255} - 19$, five elements per block. The 16-byte nonce is read as a little-endian integer n , and counter block i is the vector $[n, i, 0, 0, 0]$. The keyed Rescue permutation of that block yields five keystream elements, and each plaintext element is added to its keystream element modulo p :

```
c[5i + j] = ( m[5i + j] + Rescue_key( [n, i, 0, 0, 0] )[j] ) mod p
```

An eight-element intent uses two counter blocks. Each ciphertext element is serialized as 32 little-endian bytes. The cipher rejects any plaintext element that is not below p ; the 214-bit packing guarantees that none is.

4.3.4 The `rescue-32-v1` envelope

`encryptFields` wraps the ciphertext in the envelope that every API object carries: `elements` (1–8 ciphertext elements of exactly 32 bytes), `nonceBase64` (16 bytes), `senderPublicKeyBase64` (the sender's 32-byte x25519 key) and the literal encoding: `"rescue-32-v1"`; §17 A.7 lists it field by field. Every base64 string must re-encode to itself, so each value has one textual form. The sender key must be a canonical x25519 coordinate: read as a little-endian integer it is non-zero and below $2^{255} - 19$. The SDK (`assertCanonicalX25519PublicKey`), the API and the program (`canonical_x25519`) apply the same test. Without it, two byte strings could name one public key, and the key bytes could not serve as a replay identifier (§4.5.3).

4.3.5 One key for the input and the result

The intent's sender key and its result's recipient are the same key. The API rejects a request whose `x25519PublicKeyBase64` differs from the envelope's sender, the database requires the two stored key columns to be equal, and `post_intent` takes both keys and fails with `InputKeyMismatch` unless `input_x25519 == user_x25519`. Inside the computation, `match_intent` seals its result with `input.owner.from_arcis(...)`, which is the owner of the encrypted input itself, not a separately supplied recipient.

This binding decides who can read a result. A party that copies an intent's ciphertext and reposts it under its own sender key makes the MXE derive a different shared secret: the copy decrypts to unrelated field elements, fails `valid_intent`, and its result is an empty top three sealed to the copier. Reposting under the original sender key is impossible, because the (sender, nonce) pair is single-use. The agent decrypts the genuine result with `decryptSealedTopThree`, which recomputes `Z` from the same per-intent secret. The result travels under its own nonce, chosen by the MXE, so the input's keystream is never reused (§6.7).

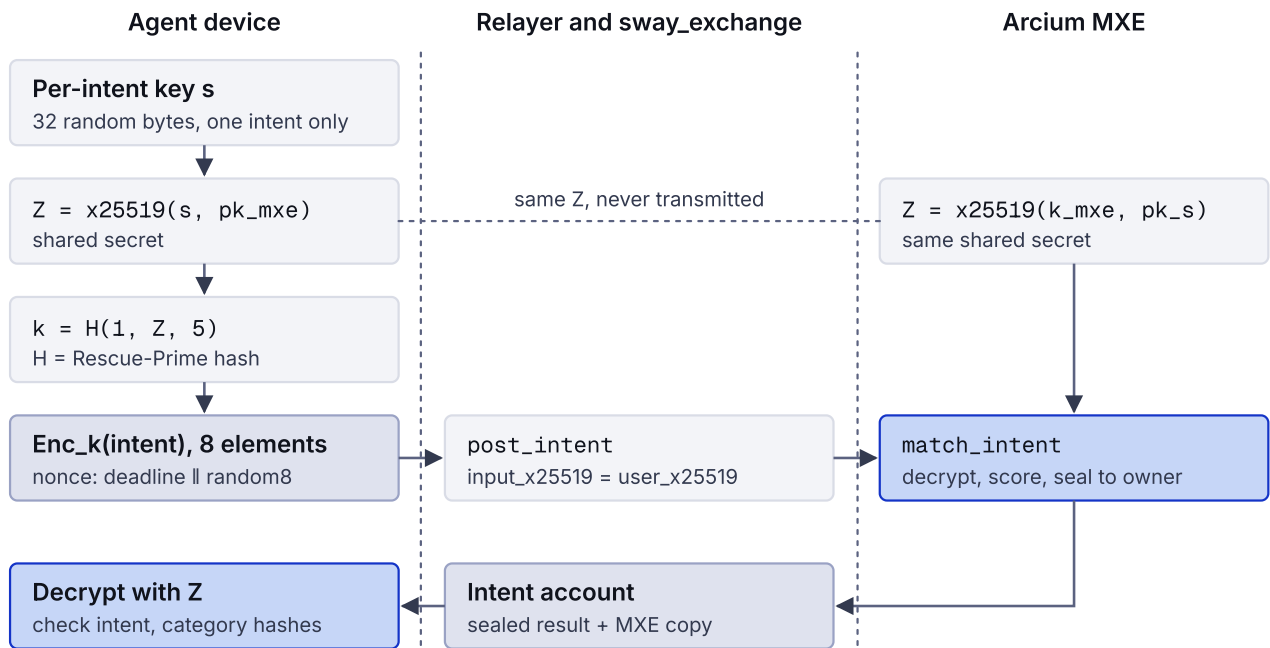


Figure 4.2 — One x25519 key per intent. The agent and the MXE derive the same shared secret Z from their own secrets and each other's public keys; the agent's public key pk_s travels in the envelope as the sender key. The intent is encrypted under the key derived from Z , and the result is sealed to the owner of that input and decrypted with the same Z .

4.4 The nonce and the purchase deadline

An intent's nonce is not random throughout. GET /v1/encryption advertises its format as expiry-u64le-random8:

Bytes	Content
0–7	purchase deadline in Unix seconds, unsigned 64-bit little-endian
8–15	eight random bytes

`encryptIntent(intent, mxePublicKey, perIntentSecretKey, purchaseDeadlineUnixSeconds)` draws 16 random bytes and overwrites the first eight with the deadline. The deadline must be positive and at most $2^{63} - 1$, so that it converts to the program's signed i64 time. The agent reads `chainUnixTimestamp` and `purchaseWindowSeconds` from GET /v1/encryption and chooses a deadline no later than their sum; it may choose an earlier one. The Sway MCP tools use the full window.

Placing the deadline in the nonce binds it to the ciphertext. The nonce is the first word of every counter block, so changing any deadline byte changes every keystream element. A relay that edits the nonce to extend an intent's life makes the MXE decrypt different plaintext, which fails the tag and canonical checks. The program, for its part, reads the deadline without decrypting anything.

`post_intent` reads `deadline = u64_le(nonce[0..8])`, converts it to i64, and requires

```
now < deadline ≤ now + Config.purchase_window_secs
```

failing with `PurchaseWindowClosed` otherwise. The API applies the same test against the finalized chain clock before it accepts a request. The program then fixes two times on the intent:

```
purchase_deadline    = deadline
settlement_deadline = deadline + Config.settlement_grace_secs
```

`purchase_window_secs` and `settlement_grace_secs` are u32 values set when the program is initialized, and both must be positive. The purchase deadline bounds the purchase itself: a receipt timestamp must fall in `[created_at, purchase_deadline)` (§7), and the match callback and `retry_intent` fail once it has passed. The settlement deadline bounds how late a receipt or a paid commission may still be recorded (§8, §10). Neither is related to the 180-slot computation expiry of §6.9, which counts slots and applies to each matching attempt separately.

Offer nonces and result nonces keep the opaque format: 16 random bytes with no structure.

4.5 Posting the intent

4.5.1 The REST request

The agent sends `POST /v1/intents` as a signed agent request with the `intent:write` scope and an `Idempotency-Key` header:

Field	Content
<code>intentId</code>	Random opaque identifier chosen by the agent, 1–96 characters of <code>[A-Za-z0-9_-]</code> . It doubles as the affiliate sub-ID (§10).
<code>shard</code>	The category shard
<code>encryptedIntent</code>	The <code>rescue-32-v1</code> envelope
<code>claimPublicKey</code>	The per-intent Ed25519 claim public key, base58
<code>x25519PublicKeyBase64</code>	The result key; must equal the envelope's sender key
<code>cashbackOnly</code>	Optional. Registers the intent for the cashback fallback without matching (§10).

The API checks, in order: the agent signature and scope; that cashback-only registration is configured if it is requested; that the envelope is canonical (sender key, 16-byte nonce, 1–8 elements of 32 bytes, encoding name); that the sender key equals `x25519PublicKeyBase64`; that the claim key decodes; that the nonce's deadline lies inside the purchase window measured on the chain clock;

and that the shard is admitted. It then creates the intent in one database transaction together with the replay reservation of §4.6.1, queues the job that posts it on chain, and returns the intent in state pending.

4.5.2 What the relayer and API store

The service stores ciphertext, public keys and chain metadata. No column holds any intent attribute in plaintext.

Stored value	Purpose
externalId, internal UUID id	The REST identifier, and the 16 bytes that seed the Intent PDA
agentKeyId, integratorId	The posting agent key, and the integrator whose payout wallet receives the integrator fee
bookId	The shard
claimPublicKey	Unique across all intents
clientX25519PublicKey, userX25519PublicKey	Sender key and result key, required equal
ciphertext, ciphertextNonce, ciphertextElementCount	The exact envelope bytes
purchaseDeadlineAt, settlementDeadlineAt	Mirrors of the chain deadlines
matchBookSnapshot, matchBookRevision	The frozen book capture used for matching (§4.5.5)
sealedTopThree, mxeTopThree, their nonces, mxeSnapshotHash	The two result copies once the callback lands (§6.8)
status, fallbackReason, matchAttempt, expiresAtSlot	Lifecycle mirrors

A separate IntentEncryptionGuard row holds the sender key, the nonce, a SHA-256 digest of the envelope and the identifier of the intent it is bound to.

4.5.3 The post_intent instruction

A worker posts the intent through the relayer, which signs as fee payer and as the intent's agent. retry_intent and close_intent later require the same agent signature.

```
post_intent(id: [u8;16], shard: [u8;32], encrypted: [[u8;32];8], nonce: [u8;16],
           input_count: u8, input_x25519: [u8;32], user_x25519: [u8;32],
           claim_key: Pubkey, integrator: Pubkey)
```

The program checks:

1. $0 < \text{input_count} \leq \text{MAX_INPUT_ELEMENTS}$ (8), and neither `claim_key` nor `integrator` is the default key (`InvalidInput`).
2. `input_x25519 == user_x25519`, and the key is canonical (`InputKeyMismatch`).
3. The nonce's deadline lies in the purchase window (`PurchaseWindowClosed`).
4. The supplied book's stored shard equals `shard` (`InvalidInput`).

It creates three accounts:

Account	Seeds	Size	Content
Intent	["intent", id]	1,309 bytes	The intent, below
IntentBook	["intent_book", intent]	3,672 bytes	The book's 32 campaign keys and active mask at posting time, with each slot's two offer ciphertext elements, nonce and element count
InputReplay	["input", input_x25519, nonce]	9 bytes	Makes the (sender, nonce) pair single-use

`InputReplay` is created with `init`, so a second intent with the same sender and nonce cannot be posted while the first one's guard exists. The fields `post_intent` writes, with their byte offsets in the `Intent` account:

Offset	Field	Value at posting
9	id	16-byte internal UUID
25	shard	SHA-256 of the shard string
57	agent	The signing agent (the relayer)
89	claim_key	Per-intent Ed25519 key
121	integrator	Integrator payout wallet
153, 185	input_x25519, user_x25519	The equal x25519 keys
217	encrypted	8 × 32 bytes; unused trailing elements are zero
473	nonce	Deadline and random bytes
489, 490, 491	input_count, attempt, status	element count, 1, 0 (pending)
492	expires_at_slot	current slot + 180
500	created_at	chain time
1116, 1148	payer, book	Rent payer, book account
1180, 1188	book_revision, book_active_mask	Book state at posting
1192	input_commitment	§4.5.4
1293, 1301	purchase_deadline, settlement_deadline	§4.4

4.5.4 The input commitment

post_intent binds the intent to everything the computation will use:

```
book_hash          = SHA-256( OfferBook account data )
input_commitment = SHA-256( "sway/intent-input/v1" || 0x00 || intent_address
                           || input_x25519 || nonce || input_count
                           || encrypted[0 .. input_count] || claim_key ||
```

The book's 10,120 bytes are hashed with Solana's SHA-256 syscall and only the digest enters the commitment, which keeps the instruction within the ordinary compute budget. The match callback must present this exact value or fail with SnapshotMismatch (§6.8), and the worker recomputes it from its own frozen copy of the book and the stored envelope before it starts a computation. Matching therefore runs on exactly the ciphertext, book bytes, claim key and integrator that the program recorded.

4.5.5 The frozen book

At posting, the worker captures the book under a per-shard lock: the raw account data, each slot's campaign and brand, and each slot's effective active flag from the public test of §5.2.5. The capture is stored as `matchBookSnapshot` with its revision and stays fixed for the intent's lifetime, so a later change to a vault balance, the vault floor or a merchant's status cannot change what a retry evaluates. `record_conversion` checks the purchased slot against the campaign keys and mask frozen in `IntentBook` (§7).

The owning agent reads the public part of the capture with `GET /v1/intents/:id/book`: for each of the 32 slots, the campaign identifier, brand identifier and active flag, or null for an empty slot. It resolves the hashes in its decrypted result against this map (§6.7.3). The map names no winner and contains no price, rebate or rule.

4.6 Retry rules

Two different things can be retried: the delivery of a request, and the matching computation.

4.6.1 Retrying the request

- Repeating `POST /v1/intents` with the same `Idempotency-Key` and body returns the original result.
- A request whose `intentId` already exists returns that intent only if every committed term is identical: the cashback-only mode, the agent key, the claim key, the shard, the sender key, the nonce and every ciphertext byte. Any difference fails with `IDEMPOTENCY_CONFLICT` (409).
- A claim key already bound to another intent fails with `REPLAY_DETECTED` (409).
- `IntentEncryptionGuard` owns each (sender key, nonce) pair. The reservation runs in the transaction that creates the intent. An exact retry finds its own reservation; the same pair with another intent or a different envelope digest fails with `REPLAY_DETECTED`. The guard has no foreign key, so deleting an intent row does not release the pair.
- On chain, the `InputReplay` account rejects a second `post_intent` with the same pair.
- A changed request is a new intent with a new sender key and nonce. The agent stores the keys and the exact request under the intent identifier before the first post and resends that request when it resumes; the MCP tools refuse to reuse an intent identifier for different preferences and never overwrite an existing record's keys.
- An intent that never reached the chain and whose deadline has passed is marked `failed`; the agent starts a new intent.

4.6.2 Retrying the computation

A matching attempt lives for 180 slots. If attempt 1 expires without a callback, the API reports the intent as `expired`, and the agent may call `POST /v1/intents/:id/retry` once. The retry reuses the posted ciphertext and the frozen book; nothing is re-encrypted.

Intent state when the retry is requested	Result
pending	FORBIDDEN (409): the computation must expire before its one retry
expired	The intent returns to pending; the worker calls <code>retry_intent</code> (attempt 2, a fresh 180-slot window) and queues matching again
matched or <code>cashback_fallback</code>	The current state is returned unchanged

If attempt 2 also expires, the intent ends in the cashback fallback state (`cashback_fallback`) with reason `COMPUTATION_EXPIRED_TWICE`. The program enforces the count: `retry_intent` moves attempt 1 to attempt 2, and a second call sets the terminal fallback state. §6.9 gives the full lifecycle.

4.7 Closing the intent

After `settlement_deadline`, ONCE `open_conversions` is ZERO, `close_intent` closes the Intent, `IntentBook` and `InputReplay` accounts and returns their rent to the recorded payer. The nonce of a closed intent carries a deadline in the past, so `post_intent` rejects the same (sender, nonce) pair after its guard account is gone, and the database guard keeps the pair reserved permanently.

5. Offers, campaigns and the book

A brand takes part in Sway through campaigns and offers. A campaign is a USDC vault with spending limits and a lock period. An offer is a sealed rule, made of eligibility conditions, a product price and a rebate, that occupies one of the 32 slots of a category book and is evaluated against every intent posted to that category. This chapter describes the campaign account and its money rules, the book and the way its slots are allocated and activated, the offer's plaintext and wire format, and the path an offer takes from the brand's browser through `submit_offer` and the MXE's `ingest_offer` computation into the book. What happens to a campaign's money after a purchase is the subject of §7 to §9.

5.1 Campaigns

A merchant must be registered and active before it can create a campaign. The program requires `merchant.status == 1`, which the administrator sets with `activate_merchant` only once the merchant's bond holds at least `Config.minimum_bond` (§11).

5.1.1 The campaign account

`create_campaign(id, per_conversion_cap, daily_cap, lock_days)` creates a Campaign account of 157 bytes at `["campaign", merchant, id]`, together with its vault.

Field	Type	Meaning
<code>id</code>	<code>[u8; 16]</code>	Campaign UUID
<code>merchant</code>	Pubkey	Owning merchant account
<code>vault</code>	Pubkey	USDC token account at <code>["campaign_vault", campaign]</code> , whose authority is the campaign PDA
<code>per_conversion_cap</code>	<code>u64</code>	Largest rebate a single conversion may carry
<code>daily_cap</code>	<code>u64</code>	Largest total debit (rebate plus fees) per UTC day
<code>funded_net</code>	<code>u64</code>	Deposits minus withdrawals
<code>spent_today,</code> <code>spent_day</code>	<code>u64,</code> <code>i64</code>	Debit counted for day number <code>spent_day</code>
<code>spent_total</code>	<code>u64</code>	Debit that has left the vault for escrow and has not been refunded
<code>lock_days</code>	<code>u16</code>	Escrow lock after verification
<code>return_window_days</code>	<code>u16</code>	The merchant's return window, copied at creation
<code>pending_total</code>	<code>u64</code>	Debit reserved by recorded conversions that await verification

5.1.2 Caps

```
0 < per_conversion_cap ≤ REBATE_CAP = 50,000,000 base units (50 USDC)
0 < daily_cap ≤ CAMPAIGN_CAP = 5,000,000,000 base units (5,000 USDC)
funded_net ≤ CAMPAIGN_CAP
```

A violation fails with `CapExceeded`. `REBATE_CAP` and `CAMPAIGN_CAP` are generated into the program from `caps.perConversionRebateUsdc` and `caps.perCampaignUsdc` in the protocol parameters, so the program, the API and the SDKs read one source. The API checks the same bounds before it prepares a transaction (`CAP_EXCEEDED`, 422), and it also requires an active merchant whose bond meets the configured minimum (`FORBIDDEN`, 403).

The per-conversion cap limits the rebate; the daily cap and the funding cap limit total debit. With the brand-direct fees of 1,000 and 500 basis points (§9), the largest debit a single conversion can produce is

```
50,000,000 + floor(50,000,000 · 1000 / 10000) + floor(50,000,000 · 500
```

5.1.3 Lock days and the return window

The merchant declares its return window in days when it registers (`register_merchant(receipt_signer, return_window_days)`). A campaign's lock must cover that window: `create_campaign` fails with `LockTooShort` unless `lock_days ≥ merchant.return_window_days`, and the campaign stores the window it was checked against. The lock starts when a conversion is verified, `unlock_at = verified_at + lock_days × 86,400`, and a refund is possible only before `unlock_at` (§8). No campaign can release a rebate while the purchased goods can still be returned.

5.1.4 Funding and withdrawal

`fund_campaign(amount)` moves USDC from the merchant's token account into the vault and raises `funded_net`. It requires `amount > 0` and `funded_net + amount ≤ CAMPAIGN_CAP`. Every token account in the program is constrained to the configured mint, which initialize pins to plain SPL USDC with six decimals.

`withdraw_campaign(amount)` returns unspent funds to a token account of the merchant's wallet. All three conditions must hold:

```
0 < amount ≤ vault.amount
vault.amount - amount ≥ pending_total
funded_net - amount ≥ spent_total + pending_total
```

The second condition keeps reserved debits in the vault. The third keeps the accounting identity that every debit ever reserved or spent was funded. Money that verification has moved to the claims vault is no longer in the campaign vault, so escrowed rebates cannot be withdrawn by the brand.

5.1.5 Daily cap and pending liability

A conversion touches its campaign twice. When a receipt is recorded, `reserve` adds the conversion's total debit to `pending_total` after checking

```

pending_total + total ≤ vault.amount
spent_total + pending_total + total ≤ funded_net
spent_today' + pending_total + total ≤ daily_cap

```

where `spent_today'` is `spent_today` if `spent_day` equals the current day number (`unix_time div 86,400`) and zero otherwise. When the verification callback arrives, the debit leaves `pending_total`. If the conversion is valid, the program checks the vault balance, the funding identity and the daily cap again, transfers the debit from the vault to the claims vault, and adds it to `spent_today` and `spent_total`. A refund moves the debit back and subtracts it from `spent_total`, and from `spent_today` if the conversion was counted on the current day. §8 describes these steps in full.

The API exposes the reservation as `pendingLiability` and the amount a brand can take out as `withdrawableBalance = vaultBalance - pendingLiability`, floored at zero. Its campaign status is `pending` until the account is observed on chain, `underfunded` while the vault balance is below the vault floor (§5.2.5), `active` otherwise, or `paused` and `closed` when the operator sets them.

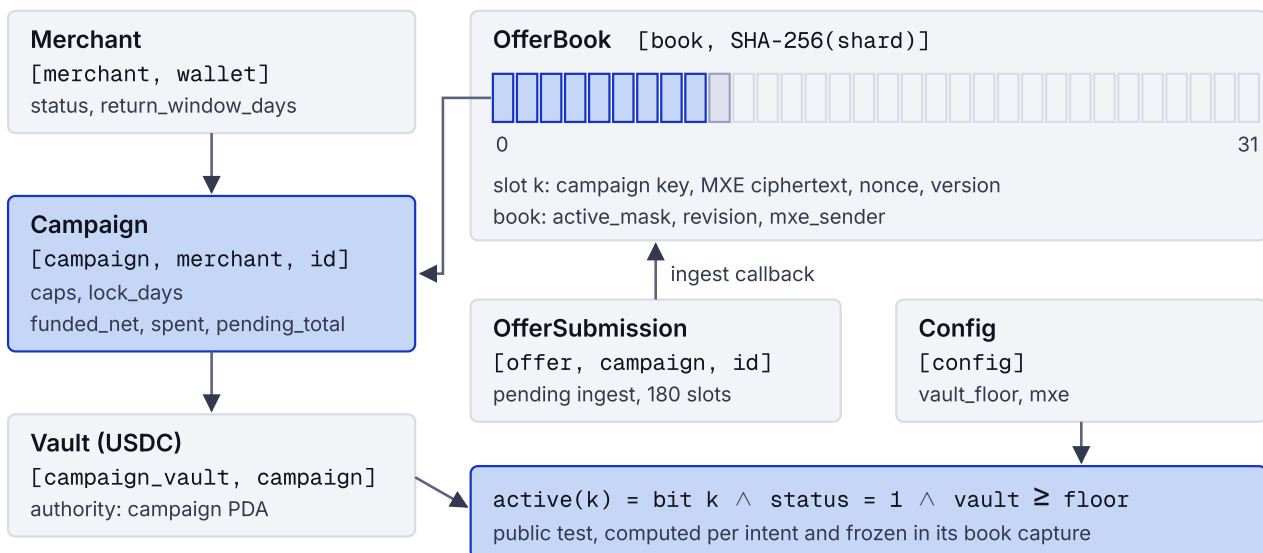


Figure 5.1 — Accounts behind an offer, with seeds in brackets. Each book slot binds one campaign; a campaign belongs to one merchant and owns one vault. A slot counts as active only if its offer was accepted, its merchant is active and its vault holds at least the configured floor. In the strip, slots 0–8 are active, slot 9 holds an accepted offer whose vault is below the floor, and slots 10–31 are empty.

5.2 Category shards and the book

5.2.1 Shards

A shard is a category name such as `running-footwear`. Every intent and every offer names exactly one shard, and matching only ever compares an intent with the offers in its own shard's book. The `v0` scope is one vertical, running and outdoor footwear and gear, sold in the US. A book has 32 slots; a category that needs more offers is split into narrower sub-categories, each with its own book.

The book account lives at `["book", SHA-256(shard)]`. The `create_book(shard)` instruction itself requires only a payer. The API creates books through the relayer, and only for shards on its configured allowlist (`SHARD_ALLOWLIST`), before it accepts the first intent or offer for a shard. `GET /v1/encryption` lists the admitted shards.

5.2.2 The OfferBook account

OfferBook is a zero-copy account of 10,120 bytes:

Offset	Field	Size	Content
0	discriminator	8	
8	bump	1	Followed by 7 bytes of padding
16	shard	32	SHA-256 of the shard string
48	revision	8	Incremented by every accepted ingest and every slot clear
56	campaigns	32 × 32	Campaign account bound to each slot; zero when none
1080	encrypted	32 × 8 × 32	Each slot's MXE ciphertext; an offer uses 2 of its 8 elements
9272	nonces	32 × 16	Each slot's ciphertext nonce
9784	counts	32 × 1	Elements in use per slot
9816	slot_versions	32 × 8	Incremented on every write to the slot
10072	active_mask	4	Bit <code>k</code> set while slot <code>k</code> holds an accepted offer; 4 bytes of padding follow
10080	mxe_sender	32	The MXE x25519 key under which every slot is encrypted
10112	category_hash	8	<code>identifier_hash("category", shard)</code> , the shard's 64-bit category identifier passed to matching

5.2.3 Slot allocation

The API assigns a slot when a brand submits an offer. It holds a per-shard lock and a database advisory lock while it does so, so two submissions never race for one index:

1. Every slot whose bit is set in the book's on-chain `active_mask` is occupied.
2. Every stored offer of this shard in state `ingest_pending` or `accepted` keeps its slot occupied, unless the slot has become reusable: an accepted offer whose slot is no longer in the mask; a submission the chain recorded as rejected, or whose `expires_at_slot` has passed; or a prepared transaction that never landed and whose last valid block height has passed. A reusable record is marked `rejected`, which releases its slot.
3. The lowest unoccupied index from 0 to 31 is chosen.
4. If every slot is occupied, the request fails with `CAP_EXCEEDED` (409): the book has no available slot.

A partial unique index in PostgreSQL owns `(bookId, slotIndex)` only while an offer is `ingest_pending` or `accepted`, so rejected history keeps its metadata without holding capacity.

5.2.4 Slot binding on chain

The program enforces slot ownership on its own terms:

- `submit_offer` accepts a slot that is either clear in `active_mask` or already bound to the same campaign, and fails with `CampaignMismatch` otherwise. A campaign can overwrite its own slot, never another campaign's.
- The submission records the slot's `slot_versions` entry. The ingest callback fails with `StaleCallback` if the slot has been written or cleared since, so an older computation cannot overwrite a newer offer.
- The callback's MXE key must equal the book's `mxe_sender`, unless the book is empty. Every ciphertext in a book is therefore encrypted under one MXE key, which `match_intent` needs because it receives all 32 slots as `Enc<Mxe>` values.
- `clear_book_slot(slot)`, signed by the administrator, empties a slot and increments its version and the book revision.

5.2.5 The public active-slot mask and the vault floor

The book's `active_mask` records which slots hold accepted offers. Matching applies a stricter test to each slot `k`:

```
active(k) = bit k of book.active_mask
           ^ merchant(campaign_k).status = 1
           ^ vault(campaign_k).amount ≥ Config.vault_floor
```

Every term is public chain state, so anyone can recompute the mask an intent was matched with. The worker evaluates it when it captures the book for an intent (§4.5.5) and passes the result to the computation as `active_mask` (§6.1). An inactive offer stays in its slot; it cannot enter a top three until its vault is refilled or its merchant is reinstated, and the next intent posted after that sees it again.

`Config.vault_floor` is a positive u64 set by `initialize` and changed with `update_config`. Its value is a deployment parameter, listed as `activeSlotVaultFloorUsdc` among the protocol's required configuration. The floor keeps offers that their vault cannot pay out of every top three, so a shopper is shown only offers that a campaign can honour. The mask is passed in the clear because `match_intent` evaluates all 32 slots every time, since Arcis loops have fixed bounds. The mask decides which of those evaluations can count, and it shows nothing that vault balances and merchant accounts do not already show.

5.3 The offer plaintext and its packing

5.3.1 Fields

`PlaintextOffer` in `scoring.ts`:

Field	Type	Validation	Encrypted?
<code>slot</code>	integer	0–31; the API assigns the real slot	No, public on chain
<code>campaignId</code>	string	Required	No, public on chain
<code>brandId</code>	string	Required; the merchant's canonical identifier	Yes, as a brand hash
<code>categoryShard</code>	string	Required	Yes, as a category hash
<code>sizeRange</code>	<code>{min, max}</code>	Whole or half steps, $\min \leq \max$, each $\times 2 \leq 65,535$	Yes
<code>widths</code>	<code>Width[]</code>	Non-empty subset of narrow, standard, wide	Yes, as a 3-bit mask
<code>regions</code>	<code>string[]</code>	Non-empty; only US in v0	Yes, as one bit
<code>newToBrandOnly</code>	boolean		Yes
<code>priceUsdc</code>	u64 string	Canonical base units	Yes
<code>rebateUsdc</code>	u64 string	Canonical base units	Yes
<code>active</code>	boolean		No, the public mask

`packOffer` validates the offer with the same `evaluateOffer` function the matcher uses, run against a synthetic intent, so a malformed rule fails on the brand's device before anything is encrypted. The price is the price of the product the offer covers; the shopper's agent buys it at that price, and the merchant's receipt later states the amount paid (§7).

5.3.2 OFFER_FIELDS

```
OFFER_FIELDS = [ metadata u128, category u64, brand u64, price u64, rebate u64 ]

e0 = metadata (bits 0-127)   | category (bits 128-191)
e1 = brand    (bits 0-63)    | price    (bits 64-127)    | rebate (bits 128-191)
```

An offer is two elements, matching the circuit's OfferCipher { header: Pack<HeaderWire>, data: Pack<TripleWire> }.

5.3.3 Offer metadata

Bits	Content	Encoding
0-63	OFFER_TAG	0x535741594f464631, ASCII SWAYOFF1
64-79	minimum size	min × 2
80-95	maximum size	max × 2
96-98	width mask	bit 0 narrow, bit 1 standard, bit 2 wide
99	region	1 for US
100	new-to-brand only	1 when the offer excludes existing customers
101-127	reserved	zero

For example, an offer for sizes 6-13 in standard and wide widths, open to every customer, has the metadata word 0xe001a000c535741594f464631: minimum 12, maximum 26, width mask 0b110, region bit set. unpackOffer and the circuit's valid_offer accept a word only with the tag, zero reserved bits, a non-zero width mask, the region bit, $\min \leq \max$, and non-zero brand and category hashes.

5.4 Browser encryption

The brand console encrypts the rule in the browser with encryptOffer(offer, mxePublicKey). An offer, unlike an intent, is encrypted under a fresh ephemeral x25519 keypair whose secret is discarded at once. The brand never needs to decrypt anything under that key, because the MXE re-encrypts the offer to itself at ingest (§5.6). The nonce is 16 random bytes, and the envelope is the same rescue-32-v1 structure as an intent's, with two elements. The brandId must be the merchant's canonical identifier as returned by GET /v1/merchant; any other value fails the brand check at ingest. The plaintext rule stays in the browser, and the API stores only the envelope.

5.5 Submitting the offer

POST /v1/offers takes {campaignId, shard, encryptedRule} with a merchant wallet session and an Idempotency-Key. The API checks that the campaign belongs to the merchant, that the merchant is active on chain (FORBIDDEN otherwise), that the envelope is canonical and that the shard is admitted. It then allocates a slot (§5.2.3), prepares an unsigned submit_offer transaction for the merchant's wallet, records the offer as ingest_pending, and returns {offer, transaction}. The wallet signs the exact prepared bytes and the API broadcasts them; the API never holds the merchant's key.

```
submit_offer(id: [u8;16], slot: u8, sender: [u8;32], encrypted: [[u8;32];8],
             nonce: [u8;16], input_count: u8)
```

The program requires $\text{slot} < \text{BOOK_SLOTS} (32)$, $0 < \text{input_count} \leq 8$, a canonical sender key, an active merchant, and the slot rule of §5.2.4. It creates two accounts:

Account	Seeds	Size	Content
OfferSubmission	["offer", campaign, id]	444 bytes	Campaign, book, payer, slot, sender key, ciphertext, nonce, element count, status 0, expires_at_slot = slot + 180, the slot's current version
InputReplay	["input", sender, nonce]	9 bytes	Makes the (sender, nonce) pair single-use; intents and offers share this namespace

The submission is the queued ingest. It has the same 180-slot lifetime (QUEUE_SLOTS) as a matching attempt.

5.6 MXE ingest and the rebate cap

Once the submission is finalized, a worker runs the ingest_offer computation:

```
ingest_offer(rule: Enc<Shared, OfferCipher>, cap: u64,
             expected_brand: u64, expected_category: u64)
-> (Enc<Mxe, OfferCipher>, bool)
```

The public arguments come from chain and service state, never from the brand: cap is the campaign's per_conversion_cap, expected_brand is hashIdentifier(merchantId, "brand"), and expected_category is hashIdentifier(shard, "category"). Inside the MXE the circuit decrypts the rule and computes

```
valid = valid_offer(rule) ∧ rule.rebate ≤ cap ∧ rule.brand = expected.
```

It returns the rule re-encrypted under the MXE's own key and reveals only `valid`. The program and the API know the cap, the MXE knows the rebate, and only the comparison leaves the computation, so the rebate is checked without being shown to anyone. The brand and category tests stop a brand from planting an offer under another brand's identity or another shard's category. At matching time the brand hash inside the offer must also equal the slot's public brand hash (§6.3.1). The program checks the cap once more when a conversion is recorded, $\text{rebate} \leq \text{campaign.per_conversion_cap}$ and $\text{rebate} \leq \text{REBATE_CAP}$ (§7), so the cap holds both at admission and at settlement.

The callback is signed by the MXE authority recorded in `Config.mxe`:

```
ingest_offer_callback(valid: bool, encrypted: [[u8;32];8], nonce: [u8;16],
                     element_count: u8, sender: [u8;32])
```

1. The submission must still be pending, with `Clock.slot < expires_at_slot` (`StaleCallback`).
2. If `valid` is false, the submission becomes rejected (status 2) and nothing else changes.
3. Otherwise the merchant must still be active, $0 < \text{element_count} \leq 8$, the sender key canonical and equal to the book's `mxe_sender` unless the book is empty, the slot version unchanged, and the slot clear or bound to this campaign.
4. The program writes the campaign key, the MXE ciphertext, its nonce and its element count into the slot, sets bit `k` of `active_mask`, increments the slot version and the book revision, records `mxe_sender`, and marks the submission accepted (status 1).

Re-encryption makes the book independent of the brands' keys. Every slot holds an `Enc<Mxe>` value that only the MXE can use, and the ephemeral key the brand encrypted with plays no further part.

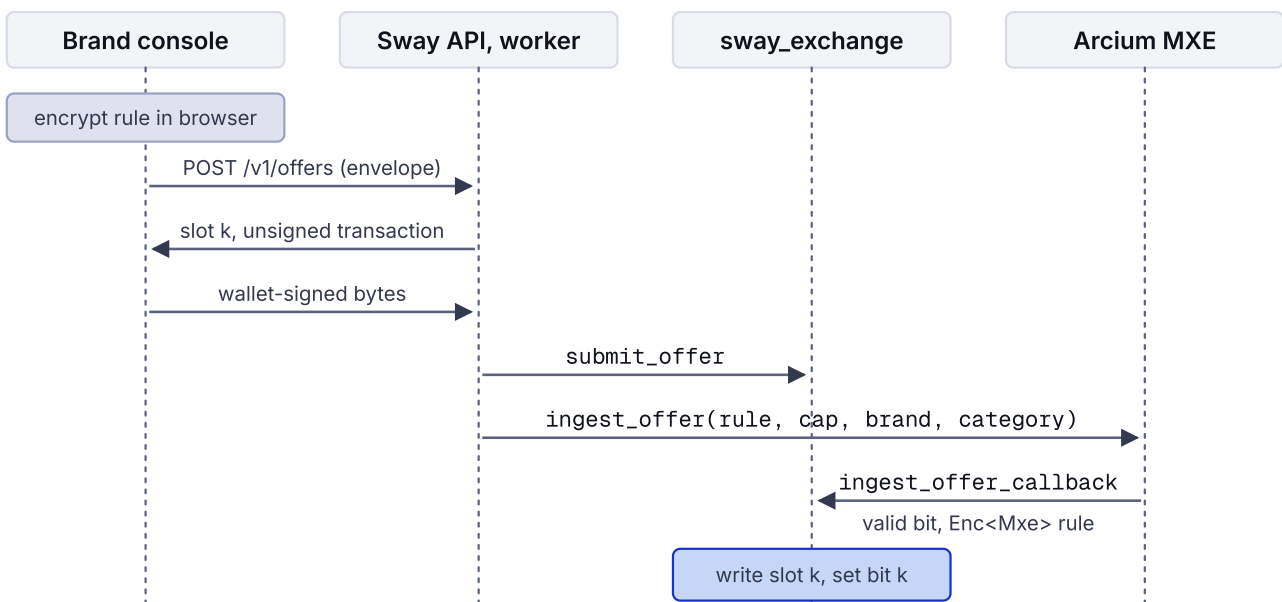


Figure 5.2 — An offer's path into the book. The API allocates the slot and prepares the transaction; the merchant's wallet signs it; the MXE checks the rebate against the campaign's public cap and the brand and category against public context, and the callback writes the MXE-encrypted rule into the slot.

5.7 Offer states

API status	OfferSubmission.status	Meaning
ingest_pending	0 (pending)	Slot reserved; waiting for the wallet signature, finality or the ingest callback
accepted	1	The MXE ciphertext is in the slot, and the slot's bit is set in the book's mask
rejected	2, or 0 past expires_at_slot	Not admitted; the slot is released

A rejected offer carries one of two codes, `REBATE_OVER_CAP` or `COMPUTATION_FAILED`. An ingest whose 180 slots pass without a callback is rejected with `COMPUTATION_FAILED`. An accepted offer stays in its slot until its own campaign overwrites it or the administrator clears the slot; after a clear, the API releases the record (marking it rejected) the next time it allocates a slot in that book. Acceptance alone does not make an offer eligible for an intent: at each posting, its slot must also pass the active-slot test of §5.2.5. `GET /v1/offers` and `GET /v1/offers/:id` return each offer's envelope, slot, book and state to the owning merchant, never a decoded rule.

6. Matching inside the MXE

Matching compares one sealed intent with the 32 sealed offers in its shard's book and returns the shopper's three best offers, sealed so that only the shopper's per-intent key can read them. It runs as the Arcis instruction `match_intent` inside Arcium's MXE. The computation is queued through Solana, executed by the Arcium cluster on secret-shared data, and its output is verified before the callback reaches the Sway program. The intent, the offers and the result are never visible in the clear to a node, the relayer, the Sway operator or any brand. This chapter specifies the computation's interface, the eligibility predicates, the score, the top-three selection, the sealed result, the callback and the lifecycle of an attempt, and closes with a worked example.

6.1 Interface

```
match_intent(  
  input:          Enc<Shared, IntentCipher>,          // 8 elements, the agent's per  
  book:           [Enc<Mxe, OfferCipher>; 32],        // 2 elements per slot  
  active_mask:    u32,  
  campaign_hashes: [u64; 32],  
  brand_hashes:   [u64; 32],  
  category_hash:  u64,  
  intent_hash:    u64,  
) -> (EncData<ResultCipher>, Enc<Mxe, ResultCipher>)
```

The two encrypted inputs are the posted intent (§4) and the book's 32 MXE-encrypted offers (§5). The public inputs bind the computation to chain state:

Public input	Value	Role
active_mask	The effective mask of §5.2.5, from the intent's frozen book capture	Decides which slots may count
campaign_hashes[k]	hashIdentifier(campaign id, "campaign") for the campaign bound to slot k; 0 for an empty slot	Must be non-zero; copied into the result
brand_hashes[k]	hashIdentifier(merchant id, "brand") for that campaign's merchant	Must equal the brand hash sealed inside the offer
category_hash	hashIdentifier(shard, "category")	Must equal the category sealed inside the intent
intent_hash	hashIdentifier(intent account address, "intent")	Must be non-zero; copied into the result

Every public input comes from the service's stored record of the intent, its frozen book capture (§4.5.5) and its on-chain address, never from the agent or a brand. The encrypted book is the

capture's MXE ciphertexts, and the encrypted intent is the posted envelope whose commitment the program recorded (§4.5.4).

Both outputs hold the same result. The first is sealed to the owner of the encrypted input and returned as `EncData`, a ciphertext and nonce without per-value public keys, which keeps the callback small. The second is encrypted to the MXE itself and kept on chain for the verification computation of §7.

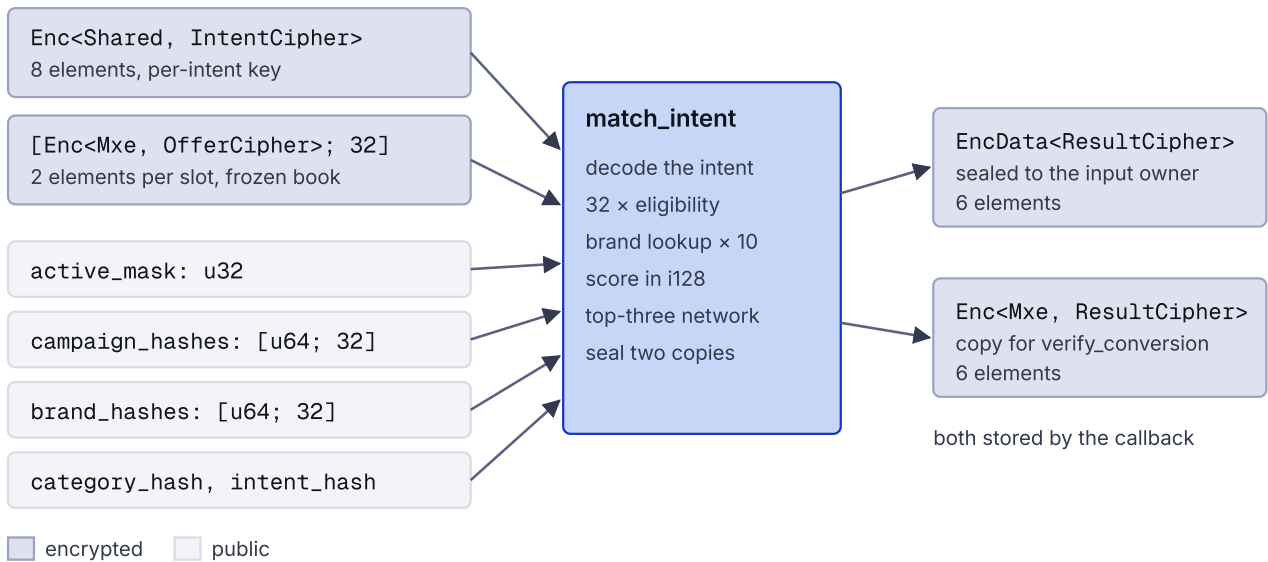


Figure 6.1 — The interface of `match_intent`. Two encrypted inputs and five public inputs enter; one result leaves in two encryptions, one for the shopper's per-intent key and one for the MXE.

6.2 Fixed-shape evaluation

Arcis programs have fixed loop bounds, and both branches of a condition on secret data are executed. `match_intent` is written for that model. It visits all 32 slots, compares each offer's brand with all 10 preference slots of the intent, and passes each candidate through all three ranks, whatever the book and the intent contain. Empty and inactive slots are evaluated like the others and fail the public checks. The work is the same for every intent, and the sequence of operations depends on no secret value.

The circuit source is generated, not written by hand. `circuits/generate.ts` lowers the eligibility predicates and the score expression from `packages/contract/src/scoring.ts` into Arcis, emits the wire structs from the codec's field descriptors, and embeds the SHA-256 of the scoring source in the generated file (`f01fed5dc33b272b7fb1c1d84287d6f1fa54b5cb9152114a8a1bde25a9e023c6` for the current source). A rule the generator cannot lower fails generation, and its check mode reports a circuit that is out of date with the scorer. The plaintext reference `matchIntent`, the landing page's demonstration and the circuit therefore implement one rule.

6.3 Eligibility

6.3.1 Well-formedness and binding gates

Before the rules, the circuit applies gates that have no counterpart in the plaintext reference, because they concern the wire format and the binding to public state:

```
intent_valid = valid_intent(intent) ^ intent.category = category_hash ^ intent_has  
gate(k)      = intent_valid ^ valid_offer(offer_k)  
               ^ offer_k.brand = brand_hashes[k] ^ campaign_hashes[k] ≠ 0
```

`valid_intent` and `valid_offer` are the canonical-form checks of §4.2.5 and §5.3.3. The brand gate ties each sealed offer to the brand that publicly owns its slot's campaign: the ciphertext in a slot cannot be exchanged for another brand's offer without that slot failing for every intent. An intent that fails `intent_valid` fails every slot, and its result is empty.

6.3.2 The seven rules

An offer that passes the gates must then satisfy seven rules. The table names each with the reason code that `evaluateOffer` reports in the plaintext reference:

Reason code	Plaintext rule (<code>scoring.ts</code>)	Circuit test
inactive-slot	<code>offer.active</code>	bit <code>k</code> of <code>active_mask</code>
category-mismatch	<code>offer.categoryShard === intent.categoryShard</code>	<code>offer.category == intent.category (hashes)</code>
size-outside-range	<code>min ≤ size ≤ max</code>	<code>min_half ≤ size_half ≤ max_half</code>
width-not-supported	<code>offer.widths.includes(intent.width)</code>	<code>width_mask & (1 << width) ≠ 0</code>
region-not-supported	<code>offer.regions.includes(intent.region)</code>	<code>offer region bit ^ intent region bit</code>
existing-brand-customer	<code>not (newToBrandOnly and the brand is in existingBrandIds)</code>	<code>¬(new_to_brand_only ^ existing)</code>
price-over-maximum	<code>price ≤ maxPrice</code>	<code>offer.price ≤ intent.maxPrice</code>

Both bounds are inclusive. An offer priced exactly at the shopper's maximum is eligible, and so is a size at either end of a range. Sizes are compared as integers in half steps: size 9.5 is 19, and a range of 6–13 is 12–26.

6.3.3 Brand lookup

An offer's bonus and the shopper's existing-customer flag come from a fixed scan of the intent's preference slots:

```
bonus = 0; existing = false
for p in 0..10:
    if p < count and intent.brandIds[p] = offer.brand:
        bonus      = intent.bonuses[p]
        existing = bit (83 + p) of intent.metadata
```

Brand hashes in an intent are distinct (§4.2.5), so at most one preference slot matches. A brand the intent does not list contributes a bonus of zero and counts as new to the shopper, as in the plaintext rule.

6.4 The score

```
score = brand_bonus + rebate - price
```

This is the protocol parameter `auction.score`, implemented by `scoreValues`. All three terms are u64 base units, and the circuit computes the score in i128, so every value is exact and negative values are ordinary. The score is the value of the offer to the shopper. The rebate lowers the price the shopper effectively pays, and the bonus is the shopper's own adjustment for brands it prefers. An offer at 140 USDC with a 22 USDC rebate from a brand the shopper values at 12 USDC scores $12 + 22 - 140 = -106$ USDC. It ranks behind an offer at 118 USDC with a 16 USDC rebate and no bonus, which scores -102 USDC.

The price enters the score in full, so a brand competes on its price and its rebate together. The bonus can reorder offers without any brand learning that it exists.

6.5 Top-three selection

The circuit keeps three ranks. Each holds a score, a slot and the offer's price, rebate, campaign hash and brand hash. Slots are visited from 0 to 31, and each candidate is pushed through the ranks:

```

for rank in 0..3:
  better = candidate.valid
    ^ ( ¬occupied[rank]
      ∨ candidate.score > score[rank]
      ∨ (candidate.score = score[rank] ∧ candidate.slot < slot[rank]))
  if better:
    swap(candidate, entry[rank])    // the displaced entry continues to the ne

```

A displaced entry moves down one rank, and whatever leaves rank 2 is discarded. The network yields the same result as sorting the eligible offers by score descending, then by slot ascending, and keeping the first three: the protocol's `topK = 3` with `tieBreak = "lower-slot-first"`. Because slots arrive in ascending order, a later slot with an equal score never displaces an earlier one. The tie-break is deterministic and uses only the public slot number. When the loop ends, the number of occupied ranks, 0 to 3, is written into the result's metadata.

6.6 First-price rebate

The payment rule is *first-price*. A brand whose offer is bought pays exactly the rebate it offered, the value carried in the sealed result, plus the fees of \$9. No amount is derived from a competing offer. The rebate goes to the shopper, so a second-price rule would pay shoppers the runner-up's lower rebate and move surplus from shoppers to brands, and it would reveal a rival's bid to the winner through the clearing price. Inside MPC no party can see or shade another brand's bid, so paying as offered needs no clearing price and keeps the whole offered rebate for the shopper.

Settlement enforces the same value. `verify_conversion` (§7) compares the purchased slot, the purchase amount, the rebate, the campaign hash, the brand hash, the category hash and the intent hash with the MXE copy of this result, and reveals only whether all of them match.

6.7 The sealed result: codec v2

6.7.1 Layout

`RESULT_FIELDS` declares the result, and the circuit's `ResultCipher` has the same shape: one header element, four triples and one binding element, six elements in all.

Element	Contents	Bit offsets
e0	metadata (u128), slots[0..2] (u8 each)	0, 128, 136, 144
e1	prices[0..2]	0, 64, 128
e2	rebates[0..2]	0, 64, 128
e3	campaigns[0..2], campaign hashes	0, 64, 128
e4	brands[0..2], brand hashes	0, 64, 128
e5	category (u64), intent (u64)	0, 64

The metadata word is `RESULT_TAG | count << 64`, with `RESULT_TAG = 0x53574159544f5032` (ASCII `SWAYTOP2`, the final digit being the codec version) and `count` the number of occupied ranks. Ranks are in score order. An unused rank holds slot 255 and zero in every other field.

6.7.2 Bindings

Codec v2 carries four bindings besides the slots, prices and rebates. The campaign and brand hash of each rank name the campaign and brand that held the slot when the book was captured. The category hash and the intent hash name the shard and the intent account the result belongs to. A result is meaningful for one intent only: a sealed result attached to another intent's record, or a slot whose campaign changed after capture, fails the agent's checks below and the settlement checks of §7.

6.7.3 Decoding on the device

The agent decrypts with `decryptSealedTopThree`, using its per-intent secret and the MXE public key, and `unpackTopThree` applies the canonical rules: exact re-packing; the tag; `count ≤ 3`; non-zero category and intent hashes; for each used rank, a slot of at most 31, non-zero campaign and brand hashes and no repeated slot; for each unused rank, slot 255 and all zeros. The agent then checks the bindings against public data:

1. `intentHash` equals `hashIdentifier(intent account address, "intent")`, and `categoryHash` equals `hashIdentifier(shard, "category")`.
2. It reads the frozen book map with `GET /v1/intents/:id/book` and checks that the map belongs to this intent and shard and has 32 entries.
3. For each offer, the map's entry at that slot exists and is active, and its campaign and brand identifiers hash to the sealed `campaignHash` and `brandHash`.
4. It recomputes each score from the decrypted price and rebate and its own bonus for that brand.

The agent resolves the result entirely on the device. The server returns the same 32-entry map for every request about the intent, so reading the result tells it nothing about which slots won. The server learns the chosen slot only when the agent requests a checkout voucher for it (§7).

6.8 The callback

The program receives both outputs through `match_intent_callback`, signed by the MXE authority recorded in `Config.mxe`:

```
match_intent_callback(attempt: u8, input_commitment: [u8;32], result_count: u8,
                      user_result: [[u8;32];9], mxe_result: [[u8;32];9],
                      user_nonce: [u8;16], mxe_nonce: [u8;16], sender: [u8;32])
```

The program checks:

1. `now < purchase_deadline` (`PurchaseWindowClosed`).
2. `status = 0`, `attempt` equals the intent's current attempt, and `Clock.slot < expires_at_slot` (`StaleCallback`). A callback for an expired attempt, or a second callback for the same intent, is rejected.
3. `input_commitment` equals the commitment `post_intent` recorded (`SnapshotMismatch`).
4. $0 < \text{result_count} \leq 9$, the sender key is canonical, and every element after `result_count` is zero in both arrays (`InvalidInput`).

It then stores both copies, their nonces, the MXE sender key and the element count, sets `status = 1` (`matched`), and records a hash of the MXE copy:

```
snapshot_hash = SHA-256( "sway/mxe-snapshot/v1" || 0x00 || result_count ||
```

Every later step that relies on the result recomputes this hash from the stored copy and compares it with the stored value: `record_conversion`, `verify_conversion_callback` and the re-queueing of a verification (§7). Codec v2 uses six of the nine element positions. The account reserves nine, and the API schema allows up to $\text{topK} \times 3 = 9$. Each copy always carries six elements, whatever the number of matches.

Offset in Intent	Field	Size
508	user_result	9 × 32
796	mxe_result	9 × 32
1084	user_result_nonce	16
1100	mxe_result_nonce	16
1224	snapshot_hash	32
1256	result_count	1
1261	mxe_sender	32

GET /v1/intents/:id returns the user copy as sealedTopThree (elements, nonce and encoding) once the intent is matched. The MXE copy is never returned through REST.

Each of Sway's three circuits has one callback, and all three require the same MXE signer:

Circuit	Callback	Effect
ingest_offer	ingest_offer_callback(valid, encrypted, nonce, element_count, sender)	Writes the book slot, or rejects the submission (§5.6)
match_intent	match_intent_callback(...)	Stores the sealed result and its snapshot hash
verify_conversion	verify_conversion_callback(receipt_hash, valid)	Escrows or rejects a recorded conversion (§7)

6.9 Computation lifecycle

An intent runs at most two matching attempts (`computations.agentRetries = 1`), each bounded by `computations.expirySlots = 180`, which the program holds as `QUEUE_SLOTS`.

- Queue.** `post_intent` sets `attempt = 1`, `status = 0` and `expires_at_slot = slot + 180`. The worker records a computation for the attempt, with `queuedAtSlot = expires_at_slot - 180`, and enqueues matching.
- Run.** The worker confirms that the intent is still pending and its slot window open, marks the computation running, and submits `match_intent` with the posted ciphertext, the frozen book and the public inputs.
- Complete.** A callback that passes §6.8 makes the intent matched. If the result holds no offers, the API reports the intent as `cashback_fallback` with reason `NO_ELIGIBLE_OFFERS`, and no voucher is issued for it (§10).
- Expire.** Once the chain slot reaches `expires_at_slot` without a callback, the attempt is over: the program refuses its callback. The worker marks the computation expired, and the API reports the

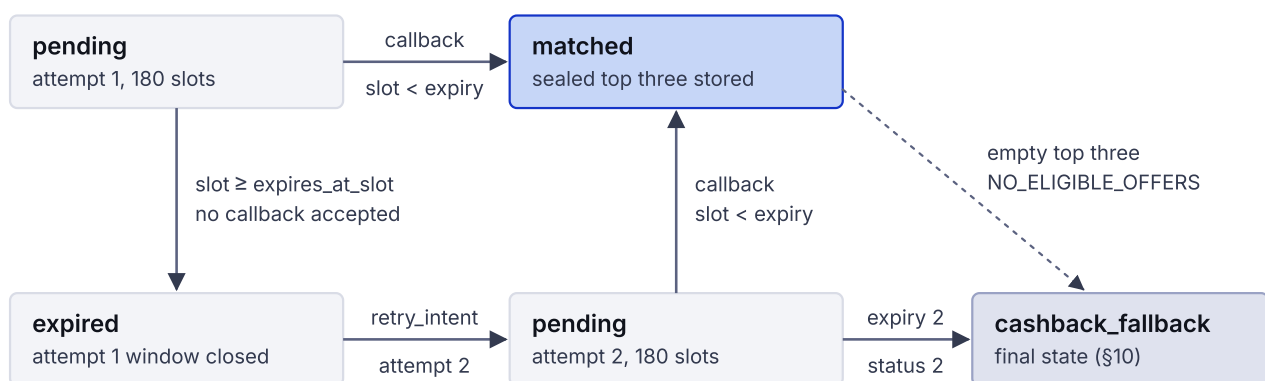
intent as expired.

5. **Retry.** When the agent requests its retry (§4.6.2), the relay calls `retry_intent`. The program requires `now < purchase_deadline`, `status = 0` and `Clock.slot ≥ expires_at_slot` (`NotExpired`); for attempt = 1 it sets attempt = 2 and `expires_at_slot = slot + 180`. Matching runs again on the same inputs, under the same commitment.
6. **Fall back.** If attempt 2 also expires, the worker calls `retry_intent` again. With attempt = 2 the program sets `status = 2`, the cashback fallback, and the API reports `cashback_fallback` with reason `COMPUTATION_EXPIRED_TWICE`. The intent cannot be matched again.

If the computation cannot run on the stored input at all, the intent ends as failed with code `COMPUTATION_FAILED`, marked as not retryable.

API intent status	Intent.status on chain	API computation state
pending	0	queued or running
expired	0, with <code>Clock.slot ≥ expires_at_slot</code>	expired
matched	1	complete
cashback_fallback	1 with an empty result, or 2	complete or expired
failed	0	failed

The worker's job key includes the attempt (`match:{intent}:{attempt}`), and the relay deduplicates the callback transaction per intent and attempt. Restarted or concurrent workers therefore submit at most one result per attempt, and the program's attempt and status checks reject any other.



The match callback and `retry_intent` also require `now < purchase_deadline`. A second expiry sets status 2.

Figure 6.2 — Lifecycle of the matching computation for one intent. Each attempt is open for 180 slots. The agent may request one retry after the first expiry; the second expiry ends in the cashback fallback. A matched intent whose top three is empty is also reported as a cashback fallback.

6.10 Worked example

The contract package ships a sample book and intent for the landing page and the documentation (SAMPLE_BOOK, SAMPLE_INTENT). Their brands, products and amounts are fictional. They serve here because together they exercise every rule.

Intent. Shard running-footwear, size 9.5, width standard, region US, maximum price 160 USDC (160000000), a bonus of 12 USDC for Kestrel Loop and 6 USDC for Bracken Pace, and an existing customer of Fell & Fern.

Book. Ten offers in slots 0–9; slots 10–31 are empty. Unless noted, each offer covers sizes 6–13 in standard and wide widths, in the US, open to every customer.

Slot	Brand	Price	Rebate	Differs from the default	Outcome	Score
0	Kestrel Loop	140	22		eligible, bonus 12	-106
1	Cairn Relay	125	14		eligible	-111
2	Bracken Pace	155	30		eligible, bonus 6	-119
3	Fell & Fern	130	24	new-to-brand only	existing-brand-customer	
4	Alderstride	118	16		eligible	-102
5	Ridgeline Loom	165	25		price-over-maximum	
6	Juniper Tread	130	20	narrow only	width-not-supported	
7	Northspur	145	25	sizes 4–8.5	size-outside-range	
8	Flint Current	125	14		eligible	-111
9	Cloudswitch	125	30	inactive slot	inactive-slot	

Amounts are in USDC; the circuit works in base units, so -102 USDC is -102000000. Size 9.5 enters as 19, Northspur's range as 8–17.

Public inputs. active_mask = 0x000001FF (bits 0–8; slot 9 is not active). campaign_hashes and brand_hashes are non-zero for slots 0–9 and zero for slots 10–31. category_hash = 3046417618813319877, the hash of running-footwear. For example, the brand hashes of the three winners' sample identifiers are 189430025308831370 (sample-alderstride), 3816214166913855063 (sample-kestrel-loop) and 16814432549607573182 (sample-cairn-relay).

Selection. The five ineligible slots never displace anything. The eligible slots pass through the network as follows:

slot in (score)	rank 0	rank 1	rank 2	discarded
slot 0 (-106)	slot 0 · -106	empty	empty	
slot 1 (-111)	slot 0 · -106	slot 1 · -111	empty	
slot 2 (-119)	slot 0 · -106	slot 1 · -111	slot 2 · -119	
slot 4 (-102)	slot 4 · -102	slot 0 · -106	slot 1 · -111	slot 2
slot 8 (-111)	slot 4 · -102	slot 0 · -106	slot 1 · -111	slot 8 (tie)

Figure 6.3 — The ranks after each eligible slot of the worked example, scores in USDC. Highlighted cells changed in that step. Slot 4 enters at rank 0 and pushes slots 0 and 1 down, which pushes slot 2 out. Slot 8 ties slot 1 and loses on the higher slot number.

1. Slot 0 (-106) takes the empty rank 0; slot 1 (-111) takes rank 1; slot 2 (-119) takes rank 2.
2. Slot 4 (-102) beats rank 0. Slot 0 moves to rank 1 and displaces slot 1 to rank 2, and slot 2 leaves the top three.
3. Slot 8 (-111) loses to ranks 0 and 1 and ties slot 1 at rank 2. Its slot number is higher, so it is discarded.

The result before encryption:

```

metadata = RESULT_TAG | 3 << 64          = 0x353574159544f5032
e0       = metadata | 4 << 128 | 0 << 136 | 1 << 144
         = 0x100040000000000000000000353574159544f5032
slots    = [4, 0, 1]
prices   = [118000000, 140000000, 125000000]
rebates  = [16000000, 22000000, 14000000]
campaigns = [h(campaign of slot 4), h(campaign of slot 0), h(campaign of slot 1)]
brands   = [189430025308831370, 3816214166913855063, 16814432549607573182]
category = 3046417618813319877
intent   = hashIdentifier(intent account address, "intent")

```

After decryption and the checks of §6.7.3, the agent shows Alderstride first (118 USDC with a 16 USDC rebate), then Kestrel Loop, which the shopper's 12 USDC preference brings within 4 USDC of first place, then Cairn Relay. Bracken Pace's 30 USDC rebate, the largest among the eligible offers, does not make up for its 155 USDC price even with the 6 USDC bonus. Fell & Fern's offer is excluded because the intent marks the shopper as an existing customer; the brand does not learn that, or that the intent existed. Slots 5, 6, 7 and 9 fail on price, width, size and the active mask.

If the shopper buys the Alderstride product and the receipt is verified (§7), the campaign is debited the rebate plus fees (§9), and the shopper receives the full rebate:

```
rebate          = 16,000,000
Sway fee        = floor(16,000,000 · 1000 / 10000) = 1,600,000
integrator fee  = floor(16,000,000 · 500 / 10000) = 800,000
total debit     = 18,400,000 base units (18.4 USDC)
```

7. Vouchers, receipts and verification

Matching ends with a sealed top three on the intent account (§6). The purchase that follows leaves the chain entirely: it happens in the merchant's own checkout, paid on ordinary payment rails. This chapter describes how that off-chain purchase becomes a conversion on chain. Three artifacts carry it across: the **voucher**, which the agent attaches at checkout; the **T2 merchant receipt**, which the merchant's signer issues after the sale; and the **verification bit**, which the MXE returns after comparing the receipt with the sealed result. The receipt and the verification bit are bound to the accounts they concern, so neither can be replayed against a different intent, slot, merchant, order or program deployment.

7.1 From sealed result to a chosen slot

The agent decrypts the sealed top three with its per-intent x25519 secret (§6.7). Each entry carries a slot, a price, a rebate, and the campaign and brand as 64-bit hashes: `hashIdentifier(value, domain)` is the first eight bytes of SHA-256 over `sway/<domain>/v1\0<value>`, read little-endian. The agent resolves the hashes against the public book map captured with its intent (`GET /v1/intents/:id/book`), which lists campaign id, brand id and active flag for all 32 slots and so reveals nothing about which slots won. It rejects the result if the result's intent hash, category hash, or any entry's campaign or brand hash disagrees with its own intent account, its shard or the captured map. The user's choice is a slot number between 0 and 31.

7.2 The voucher

The agent requests a voucher with `POST /v1/intents/:id/voucher` (operation `getVoucher`). The request is a signed agent request (scheme `SwayAgent`, scope `intent:write`), carries an `Idempotency-Key`, and has a single body field, `{ "slot": n }`. The service issues a voucher only when the intent is matched in its own records and on chain (`Intent.status = 1`) and the chain clock is still earlier than the intent's `purchase_deadline` (§8.7).

Field	Type	Content
<code>intentId</code>	string	The REST identifier the agent chose when posting the intent
<code>intentAccountAddress</code>	base58 public key	The Intent PDA (seeds <code>["intent", id]</code>), the value the receipt signs
<code>slot</code>	integer 0–31	The chosen book slot
<code>attachAtCheckout</code>	string, at most 200 characters	<code>sway:<intentId>:<slot></code> , the tag attached to the order

Beyond the slot, the voucher carries nothing from the sealed result: no price, no rebate, no campaign or brand. It carries no claim key or wallet either. It is not a bearer instrument: holding a voucher confers no claim on funds. Money moves only when the merchant signs a receipt that

passes verification in the MXE, and it moves only to a destination authorized by the intent's claim key (§8.5). The service does not check that the slot belongs to the top three, because it cannot read the sealed result. Membership is checked later, inside the MXE (§7.8).

7.3 Checkout

The agent buys from the chosen merchant through the merchant's ordinary checkout, whether that is a browser session, ACP, UCP or AP2, and attaches the voucher. The merchant signs receipts with its own receipt key, either in the brand console, which keeps the key in the browser profile, or from its own server, which posts each signed receipt to `POST /v1/receipts`. The signer needs two values from the voucher, the intent account address and the slot, and three values from the merchant's own order system: the order reference, the amount charged and the purchase time.

The agent keeps the merchant's checkout receipt (for example an AP2 Checkout Receipt) on the device as dispute evidence. It may report the purchase with `POST /v1/purchases (scope purchase:report)`, which carries only SHA-256 digests (`orderHash`, `evidenceHash`) and an optional opaque evidence handle. A report stays awaiting_attestation until a conversion with the same intent, slot and order hash exists. It never creates escrow; it is the agent's record for a dispute if no receipt arrives (§11).

7.4 The T2 merchant receipt

Sway defines three proof tiers in `PROTOCOL_PARAMETERS.proofTiers`. T1 (USDC payment checked by instruction introspection) is not part of v0. T3 (affiliate commission posted by the oracle) is the cashback path of §10. T2 is the merchant receipt: a fixed 128-byte message signed with Ed25519 (`receipt: { bytes: 128, signature: "Ed25519", p256OnChainV0: false }`).

The receipt is binary and fixed-length so that the bytes the merchant signs are exactly the bytes the program parses. There is no JSON serialization between signing and verification. All integers are little-endian; public keys are raw 32-byte values (base58 only in the API).

offset	size	field	encoding
0	8	domain	ASCII "SWAYRCP1"
8	32	intent	Intent account address
40	1	slot	u8, 0..31
41	32	order_hash	SHA-256 of the merchant's order reference
73	8	amount	u64 LE, purchase price in USDC base units
81	8	timestamp	i64 LE, purchase time in Unix seconds
89	4	signer_epoch	u32 LE, receipt signer epoch
93	32	program	sway_exchange program ID
125	3	reserved	zero

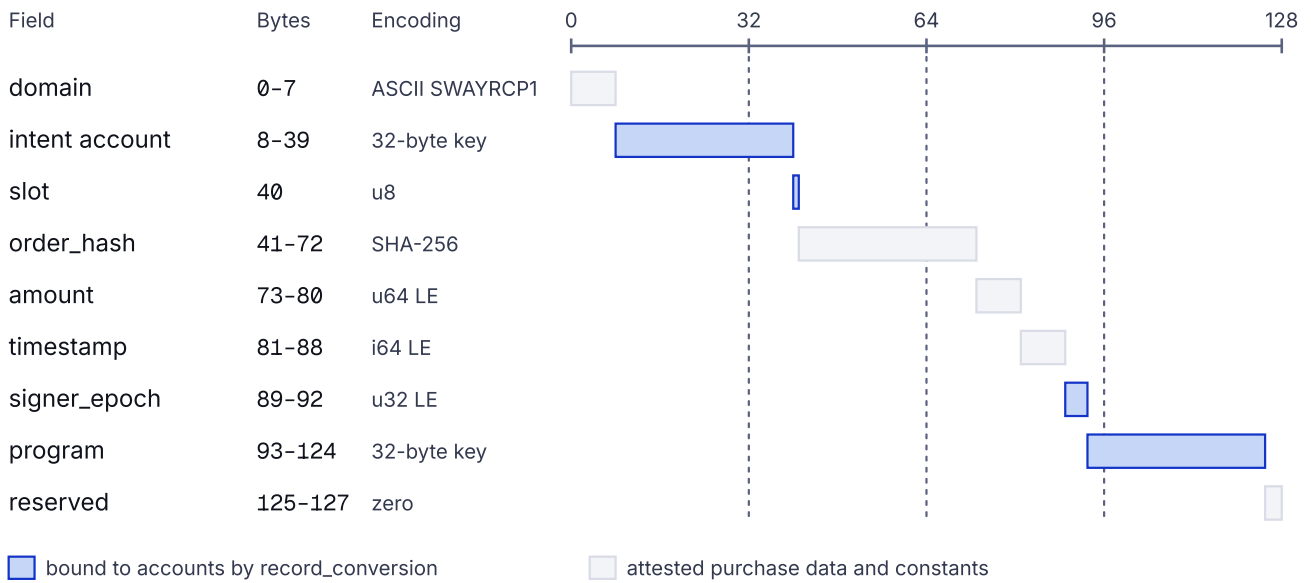


Figure 7.1 — The SWAYRCP1 receipt drawn to scale on a 128-byte ruler. The accented fields select accounts: the intent, the slot (and through the captured book, the campaign), the signer epoch account and the program.

Each field has one job:

- **Domain.** Receipts, refund notices (SWAYREF1) and claim authorizations (SWAYCLM1) are all 128 bytes and all pass through the same Ed25519 check (§7.6). The first eight bytes differ, so a signature over one kind of message is never valid as another.
- **Intent and slot.** Together they name the purchase's position in one intent's book. The receipt does not name the campaign. The program derives it from the intent's captured book map, which binds the slot to a campaign account at the moment the intent was posted. A later change to the live book cannot redirect a receipt.
- **Order hash.** The merchant hashes its own order reference with SHA-256; the plaintext reference never leaves the merchant. The hash seeds an OrderGuard account (seeds ["order", merchant, order_hash]), so one order can back at most one conversion per merchant.
- **Amount.** The purchase price, not the rebate, in USDC base units (decimals: 6). The MXE compares it with the sealed price of the chosen offer (§7.8). The rebate the brand owes is set by the sealed offer; the receipt does not carry it.
- **Timestamp.** The purchase time. It must fall inside the intent's purchase window and inside the validity window of the signer epoch named in the next field.
- **Signer epoch.** Selects which ReceiptSignerEpoch account the program loads; its four bytes are used directly as a PDA seed (§7.5).
- **Program.** Binds the receipt to one sway_exchange deployment, so a receipt signed for one cluster or deployment is invalid on another.
- **Reserved.** Must be zero. Both decoders (decodeReceiptMessage in @sway/contract/settlement and Receipt::decode in lib.rs) reject any other value, which keeps the format versionable

through its domain string.

`encodeReceiptMessage` (TypeScript) and `receipt_message` (Rust) produce identical bytes. The merchant's signer submits the receipt with POST `/v1/receipts` (operation `recordConversion`): { `merchantId`, `signerPublicKey`, `payloadBase64`, `signatureBase64` }, where the payload is exactly 128 bytes and the signature exactly 64 bytes of raw Ed25519, both in canonical base64. The request may also carry `proposedRebateUsdc`, the rebate the merchant's own offer carries, in base units. It is a proposal: the service refuses one above the campaign's per-conversion cap or `REBATE_CAP`, and the MXE certifies it against the sealed rebate before any funds move (§7.8, §7.12). The operation's auth role, `receipt`, means cryptographic attestation rather than a bearer secret: anyone may relay a valid receipt, because the signature is the authority. The response is a conversion in state `verification_pending`.

7.5 Receipt signers and signer epochs

A merchant has two kinds of keys. The merchant **wallet** owns the Merchant account, its bond and its campaigns, and signs every transaction that moves the merchant's money. The **receipt signer** only attests purchases. It never signs a transaction and controls no funds, so it can live in the brand console's browser profile or on the store's own server while the wallet stays cold. A merchant may also register a 65-byte uncompressed P-256 key; in v0 it serves only as dispute evidence (`p256OnChainV0: false`).

Signer history is kept on chain as one account per epoch:

ReceiptSignerEpoch field	Type	Meaning
<code>merchant</code>	Pubkey	Owning Merchant account
<code>signer</code>	Pubkey	Ed25519 receipt key for this epoch
<code>epoch</code>	u32	Index, starting at 0
<code>valid_from</code>	i64	Unix time the key became active
<code>valid_until</code>	i64	Unix time it was retired, <code>i64::MAX</code> while current

The account is a 93-byte PDA with seeds [`"signer_epoch"`, `merchant`, `epoch` as u32 LE]. `register_merchant` creates epoch 0 (§11.1). `rotate_receipt_signer`, signed by the merchant wallet, closes the current epoch at the rotation time, writes the new key to `Merchant.receipt_signer`, increments `Merchant.signer_epoch`, and opens the next epoch at the same instant. A receipt is checked against the epoch it names:

$$\text{signer_epoch.valid_from} \leq \text{receipt.timestamp} \leq \text{signer_epoch.valid_until}$$

This gives rotation exact semantics. A receipt for a purchase made while the old key was current remains recordable after the rotation, and a retired key cannot attest a purchase timestamped after its retirement. The epoch accounts change only once, when `valid_until` is set, so the outcome does not depend on when the receipt is submitted. Once a conversion exists, its verification does not consult the signer again: a later rotation cannot cancel a conversion already recorded. Refund notices follow a different rule and must come from the current signer (§8.3).

7.6 Ed25519 verification on chain

A Solana program does not run Ed25519 itself. The transaction carries an instruction for the native Ed25519 program, which the runtime verifies before any program executes; if that signature is invalid, the whole transaction fails. The `sway_exchange` program then reads the instructions `sysvar` to confirm that the precompile verified exactly the key and message it expects. The function `verify_ed25519(sysvar, signer, message)` performs these checks:

1. The current instruction index is greater than 0, and the instruction at `current - 1` is loaded. The precompile must immediately precede the Sway instruction.
2. That instruction's program ID is the Ed25519 program, it lists no accounts, its data is at least 16 bytes long, it declares exactly one signature (`data[0] = 1`), and its padding byte is zero.
3. In the 16-byte offsets header, the signature, public-key and message instruction indexes are all `0xFFFF`, the message size is 128, every offset is at least 16, and the 64-byte signature fits inside the data. Appendix A.5 (§17) gives the header byte by byte.
4. The 32 bytes at the public-key offset equal the expected signer, and the 128 bytes at the message offset equal the expected message.

An instruction index of `0xFFFF` tells the precompile that the bytes live inside its own instruction data. Requiring it for the signature, the key and the message forbids cross-instruction references, so a crafted transaction cannot have the precompile verify bytes stored in one instruction while the program inspects different bytes in another. Requiring every offset to be at least 16 keeps the referenced bytes out of the header itself.

The same function guards all three signed messages, with a different expected signer for each: the receipt (`ReceiptSignerEpoch.signer`), the refund notice (`Merchant.receipt_signer`) and the claim authorization (`Intent.claim_key`). The service builds the precompile with `Ed25519Program.createInstructionWithPublicKey`, which lays out the 16-byte header, the key at offset 16, the signature at 48 and the message at 112: 240 bytes of instruction data in total.

7.7 record_conversion

```
record_conversion(slot: u8, receipt: [u8; 128], rebate: u64, snapshot: [u8; 32])
```

Account	Constraint
payer	Signer, pays rent for the new accounts
mxe	Signer, must equal Config.mxe
intent	["intent", id], mutable
intent_book	["intent_book", intent], the captured book map
merchant	["merchant", wallet], mutable
signer_epoch	["signer_epoch", merchant, receipt[89..93]], has_one = merchant
campaign	["campaign", merchant, id], has_one = merchant, has_one = vault
vault	Campaign vault token account, authority = campaign
conversion	Created: ["conversion", intent, [slot]], 296 bytes
order_guard	Created: ["order", merchant, receipt[41..73]], 41 bytes
instructions	The instructions sysvar

The rebate argument is the first-price rebate of the chosen offer, proposed at record time and fixed in the conversion. snapshot names the sealed MXE copy of the top three that verification will read. The program checks, in order:

1. slot < 32 and the intent is matched (status = 1).
2. The chain clock is at or before the intent's settlement_deadline.
3. The captured book belongs to this intent, has the slot's bit set in its active_mask, and maps the slot to this campaign account.
4. snapshot equals Intent.snapshot_hash, and both equal a fresh recomputation from the stored MXE copy: snapshot_hash = SHA-256("sway/mxe-snapshot/v1\0" || count || mxe_nonce || mxe_result[0..count]).
5. The receipt decodes: domain SWAYRCP1, reserved bytes zero.
6. The receipt's intent is this intent, its slot is slot, its program is this program, and its epoch is the loaded epoch account's epoch.
7. verify_ed25519 passes for the epoch's signer over the 128 receipt bytes.
8. created_at ≤ timestamp < purchase_deadline, timestamp ≤ now, and the timestamp lies in the epoch's validity window.
9. 0 < rebate ≤ campaign.per_conversion_cap and rebate ≤ REBATE_CAP (50,000,000 base units).

It then computes the fees (\$9.2), reserves the full debit against the campaign (\$7.10), and writes the conversion: intent, campaign, integrator, receipt_hash = SHA-256(receipt), order_hash, purchase_amount, slot, proof_tier = 2, status = 0, signer epoch, rebate and both fees, the snapshot

hash, the payer and a copy of the intent's settlement deadline. `Intent.open_conversions` and `Merchant.pending_count` each increase by one.

Two account creations do replay work without any extra code. The conversion PDA is derived from the intent and slot, so each slot of an intent can hold at most one conversion. The order guard is derived from the merchant and order hash, so a second receipt for the same order fails on account creation, even if it names a different intent.

The service runs the same checks against finalized chain state before relaying, so an invalid receipt is refused before it costs rent, and it treats an unrecorded receipt that arrives after the settlement deadline as a terminal failure. An accepted receipt becomes a `settlement_conversion` job. On a deployed cluster that job first obtains the MXE's certificate and then submits [Ed25519 precompile, record_certified] with the relayer as fee payer; the coordinator's financial authority signs the two financial instructions by CPI (§7.12, §13).

7.8 verify_conversion inside the MXE

```
verify_conversion(snapshot: Enc<Mxe, ResultCipher>, claimed_slot: u8,  
                  purchase_amount: u64, rebate_amount: u64,  
                  campaign_hash: u64, brand_hash: u64,  
                  category_hash: u64, intent_hash: u64) -> bool
```

The circuit decodes the MXE copy of the result into `ResultWire`: a 128-bit metadata word, `slots[3]`, `prices[3]`, `rebates[3]`, `campaigns[3]`, `brands[3]`, `category` and `intent`. The public inputs come from the conversion and its accounts: `claimed_slot` is the receipt's slot, `purchase_amount` is the receipt's amount, `rebate_amount` is the recorded rebate, `campaign_hash` hashes the campaign identifier, `brand_hash` hashes the merchant identifier, `category_hash` hashes the intent's shard and `intent_hash` hashes the intent account address.

```
count      = metadata >> 64  
header_valid = (metadata mod 2^64 = RESULT_TAG) ∧ count ≤ 3  
              ∧ category = category_hash ∧ intent = intent_hash  
              ∧ category_hash ≠ 0 ∧ intent_hash ≠ 0  
valid      = OR over  $r \in \{0, 1, 2\}$  of  
              header_valid ∧  $r < \text{count}$  ∧  $\text{slots}[r] = \text{claimed\_slot}$   
              ∧  $\text{prices}[r] = \text{purchase\_amount}$  ∧  $\text{rebates}[r] = \text{rebate\_amc}$   
              ∧  $\text{campaigns}[r] = \text{campaign\_hash}$  ∧  $\text{brands}[r] = \text{brand\_hash}$ 
```

`RESULT_TAG` is `0x53574159544F5032`, the ASCII string `SWAYTOP2` read as a big-endian integer. Together the conditions express four facts:

- **Membership in the top three.** The claimed slot must appear at a rank below the number of matched entries. A receipt for a slot that did not make this intent's top three fails, as does a receipt for an empty rank.
- **Amount equals sealed price.** The receipt amount must equal, to the base unit, the price the offer carried when it was scored. A receipt for a different product or a different price fails.
- **Rebate equals sealed rebate.** The proposed rebate must be the offer's own rebate. The brand pays exactly what it offered (`paymentRule: "first-price"`), and no other party can raise it.
- **Same campaign, brand, category and intent.** The result cannot be reused across intents or shards, and the slot cannot be reinterpreted as another campaign's.

Arcis circuits have a fixed shape, so all three ranks are evaluated on every call. Only `valid.reveal()` leaves the MXE: one bit. The rank, the other two entries, and every price and rebate stay sealed. The program learns only that the terms it already recorded agree with the sealed result.

7.9 The verification callback

The MXE authority returns the bit with `verify_conversion_callback(receipt_hash, valid)`. The program requires the conversion to be `verification_pending`, the `receipt_hash` argument to equal the conversion's (a callback for some other receipt is stale), the intent to be matched, and the snapshot hash to still match the intent's MXE copy. In both outcomes the reservation is released: `campaign.pending_total` falls by the conversion's total and `merchant.pending_count` by one.

- **Invalid.** The conversion becomes rejected (status 4) and `Intent.open_conversions` falls by one. No funds move.
- **Valid.** The program checks once more that the vault holds the total, that $\text{spent_total} + \text{total} \leq \text{funded_net}$, and that the day's spend stays within `daily_cap`. It then transfers rebate plus fees from the campaign vault to the claims vault, signed by the campaign PDA, updates the campaign's spend counters, sets the conversion to `escrowed` (status 1), stamps `created_at` and `unlock_at = now + lock_days × 86400`, and extends `merchant.liability_until` to at least `unlock_at + dispute_seconds`. §8 covers the escrow and §11 the liability window.

The callback checks no deadline. A conversion that was recorded inside the settlement window can complete verification after the window closes, so a slow computation cannot strand a liability the brand has already reserved.

7.10 Reservation, cancellation and requeue

Recording reserves the conversion's full debit, `rebate + sway_fee + integrator_fee`, in `campaign.pending_total`. The reservation succeeds only if the vault can cover every reservation outstanding, the campaign's net funding covers spend plus reservations, and the day's spend plus reservations stays under the daily cap (the arithmetic is in §9.3). Between record and callback the reserved amount stays in the campaign vault but cannot be withdrawn: `withdraw_campaign` requires `vault - amount ≥ pending_total`.

Two instructions manage a conversion whose verification did not complete. Both are signed by either the admin or the MXE authority.

- `cancel_pending_conversion` moves a `verification_pending` conversion to `cancelled` (status 5) and releases its reservation and counters.
- `requeue_conversion` moves a `rejected` or `cancelled` conversion back to `verification_pending`, provided the chain clock is at or before the settlement deadline, the intent is still matched, and the snapshot hash is unchanged. It reserves the same total again.

Requeue reuses the conversion's immutable terms: receipt hash, rebate and snapshot. A fresh `record_conversion` for the same intent and slot would fail anyway, because the conversion and order-guard accounts already exist. Requeue lets verification run again after a failed computation without a new receipt.

7.11 Conversion states

Code	On-chain state	Merchant API (ConversionSchema)	Agent view (ClaimableSchema)	Entered by	Left by
0	<code>verification_pending</code>	<code>verification_pending</code>	<code>pending</code>	<code>record_conversion</code> , <code>requeue_conversion</code>	<code>callback</code> , <code>cancel</code>
1	<code>escrowed</code>	<code>locked / claimable</code>	<code>locked / claimable</code> <code>/ claiming</code>	<code>valid callback</code>	<code>refund</code> , <code>claim</code>
2	<code>refunded</code>	<code>refunded</code>	<code>refunded</code>	<code>refund_conversion</code>	<code>close</code>
3	<code>claimed</code>	<code>claimed</code>	<code>claimed</code>	<code>claim</code>	<code>close</code>
4	<code>rejected</code>	<code>failed</code>	<code>failed</code>	<code>invalid callback</code>	<code>requeue</code> , <code>close</code>
5	<code>cancelled</code>	<code>failed</code>	<code>failed</code>	<code>cancel_pending_conversion</code>	<code>requeue</code> , <code>close</code>

`escrowed` is projected as `locked` while the chain clock is before `unlock_at` and `claimable` from then on; the agent sees `claiming` while a claim authorization is in flight. The API also has a `disputed` state, which belongs to the dispute flow of §11 and has no on-chain code.

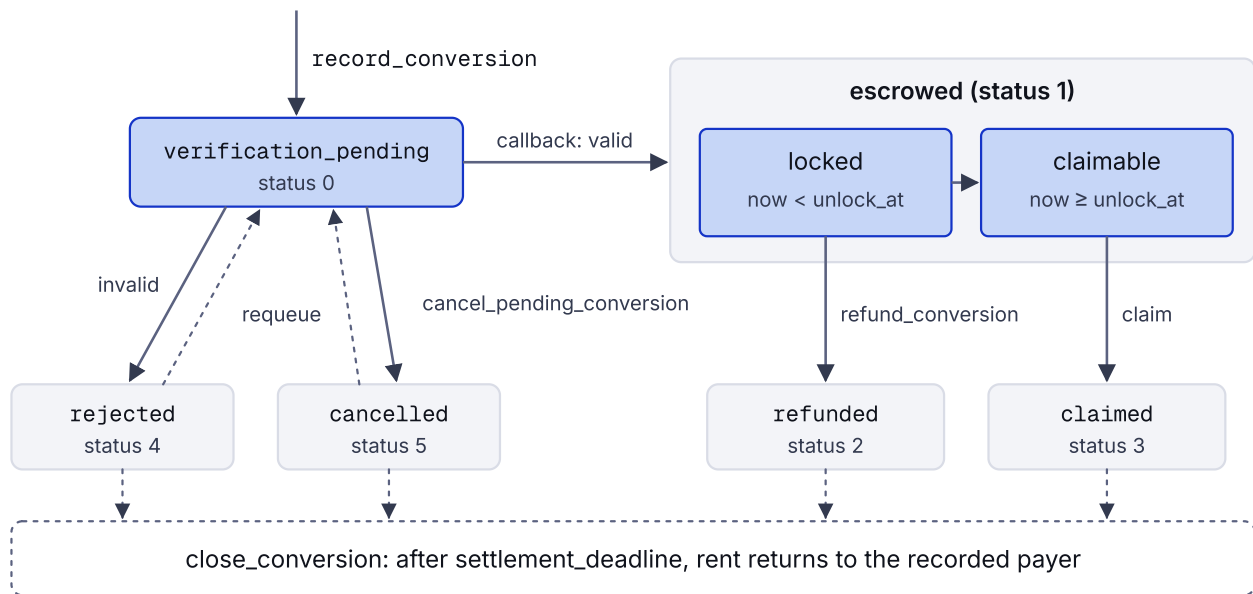


Figure 7.2 — Conversion states. Dashed arrows back to `verification_pending` are `requeue_conversion`; dashed arrows to the bottom bar are closes. Escrowed conversions have no close path: they end only by refund or claim.

Every transition that changes `open_conversions` keeps the intent's count exact: `record` and `requeue` add one; an `invalid` callback, a `cancel`, a `refund` and a `claim` each subtract one. An intent can close only when the count is zero (§8.8).

7.12 The production path: certify, then record

On a deployed cluster the financial program never takes the MXE's bit from the service. `Config.mxe` is the `["financial_authority"]` PDA of the Arcium coordinator `sway_arcium`, and that PDA signs every financial callback by CPI. The coordinator reorders settlement so that a receipt is verified before anything is written to the financial program.

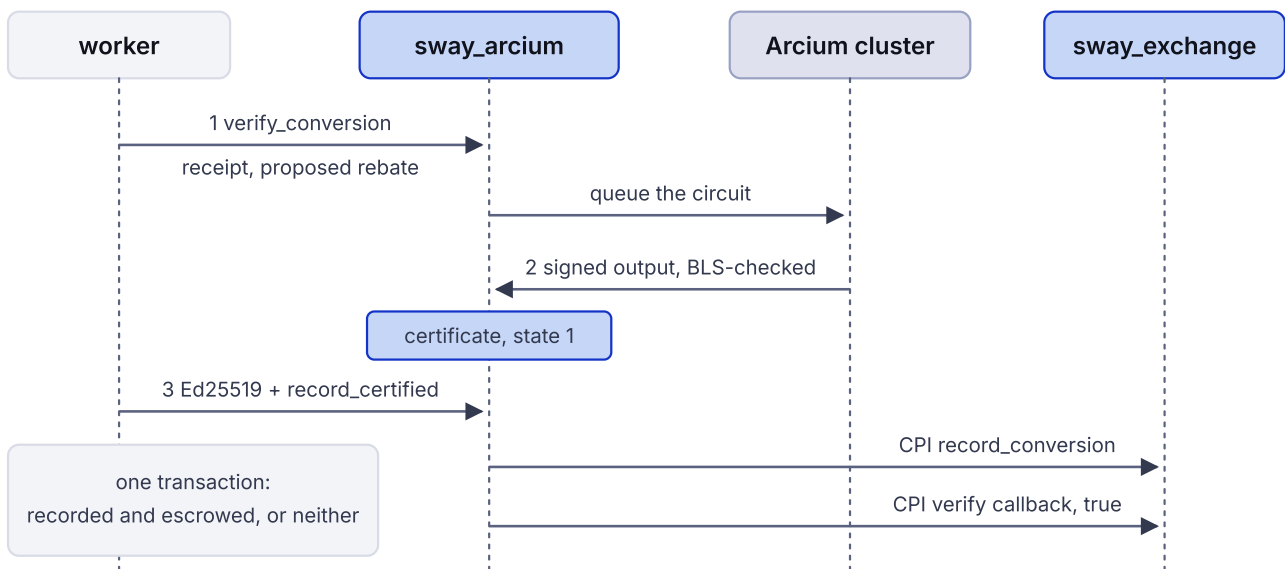


Figure 7.3 — The production order. The circuit runs on the proposed terms first; only a certificate can create a conversion, and the conversion is recorded and escrowed in the same transaction.

- 1. Queue.** The worker calls `sway_arcium.verify_conversion(computation_offset, receipt, proposed_rebate, cu_price_micro, callback_cu_limit)`. It creates a 533-byte `ComputationRequest` at `["request", offset_u64_1e]`, which commits the computation account, the target intent, the Config, the book, the campaign and the merchant, the snapshot hash captured at matching, the 128 receipt bytes and their hash, and the proposed rebate. It then queues the `verify_conversion` circuit of §7.8 on the configured cluster, which compares the native intent snapshot with the receipt's slot and price and with the proposed rebate, campaign, brand, category and intent.
- 2. Certificate.** The coordinator accepts the cluster's callback only after the SDK's BLS verifier has authenticated the signed output. A true result turns the request into a certificate (state 1). A false result is a rejected proposal. Neither outcome writes to the financial program, so a wrong proposed rebate creates no `Conversion` and no `OrderGuard` and consumes neither the receipt nor the slot; the merchant can relay the same receipt with the rebate its offer carries.
- 3. Record.** The worker submits `[Ed25519 precompile over the receipt, sway_arcium.record_certified]`. `record_certified` requires a certificate whose intent, Config, campaign, merchant and snapshot hash equal the accounts passed and the intent's current snapshot, then makes two CPIs signed by the financial authority: `record_conversion(slot, receipt, proposed_rebate, snapshot)`, which runs every check of §7.7 and the reservation of §7.10, and `verify_conversion_callback(receipt_hash, true)`, which escrows as in §7.9. The request moves to recorded (state 3). A repeated `record_certified` for a recorded request returns without effect.

On this path a conversion is `verification_pending` only inside one transaction: outside it, the conversion either does not exist or is escrowed. If the vault, the funding or the daily cap no longer admit the debit when the record lands, the whole transaction fails and nothing moves; the certificate stays usable, so the record can be submitted again once the brand funds the campaign, until

financial admission ends at the settlement deadline. A computation that reaches its 180-slot time-to-live, or whose output fails authentication, gets a fresh attempt with a new deterministic offset inside the settlement grace. The guards of §15.3, one conversion per intent slot and one per merchant order, still allow at most one money movement for a receipt.

8. Escrow, lock, refunds and claims

A verified conversion is a debt the brand has already paid but nobody has yet received. Between verification and payout the program holds that money in one pooled account, keeps it reversible for at least the merchant's return window, and then releases it to whatever address the holder of the intent's claim key names. This chapter describes the accounts that hold the money, the clock that governs it, the two signed messages that can move it (the merchant's refund notice and the user's claim authorization), the deadlines that bound the whole lifecycle, and how finished accounts are closed so their rent returns to whoever paid it.

8.1 The claims vault

Three kinds of USDC token account take part in settlement. All are plain SPL Token accounts (tokenProgram: "spl-token", token2022: false) for the USDC mint pinned at build time (PINNED_USDC_MINT, checked in initialize together with decimals = 6).

Account	PDA seeds	Token authority	Holds
Campaign vault	["campaign_vault", campaign]	Campaign PDA	A brand's unspent budget and its reservations
Claims vault	["claims_vault", config]	Config PDA	Every escrowed conversion, T2 and T3
Treasury	["treasury", config]	Config PDA	Sway's fees after claims

No private key controls the claims vault. Its authority is the Config PDA (seeds ["config"]), so only sway_exchange instructions can move its funds, and only by signing with those seeds. The vault is pooled: it holds the escrow of every open conversion at once. What each conversion is owed is stored on the conversion itself in three integer fields, rebate, sway_fee and integrator_fee, whose sum is Conversion::total().

Money enters the claims vault in exactly two places. verify_conversion_callback transfers total() from the campaign vault when a T2 conversion verifies (§7.9). oracle_post_conversion transfers the full commission from the affiliate oracle's source account when a T3 conversion is posted (§10). Money leaves in exactly three places, and each one moves a conversion out of escrowed:

Instruction	Outflow	Destination	New state
claim	rebate, sway_fee, integrator_fee	destination, treasury, integrator account	claimed
refund_conversion	total()	the conversion's own campaign vault	refunded
oracle_refund_conversion	total() (equal to the posted commission)	the oracle's source account	refunded

Each outflow requires `status = 1` and sets a different terminal status, so the escrow of a conversion leaves the vault at most once, and the vault's balance always covers the sum of `total()` over all escrowed conversions.

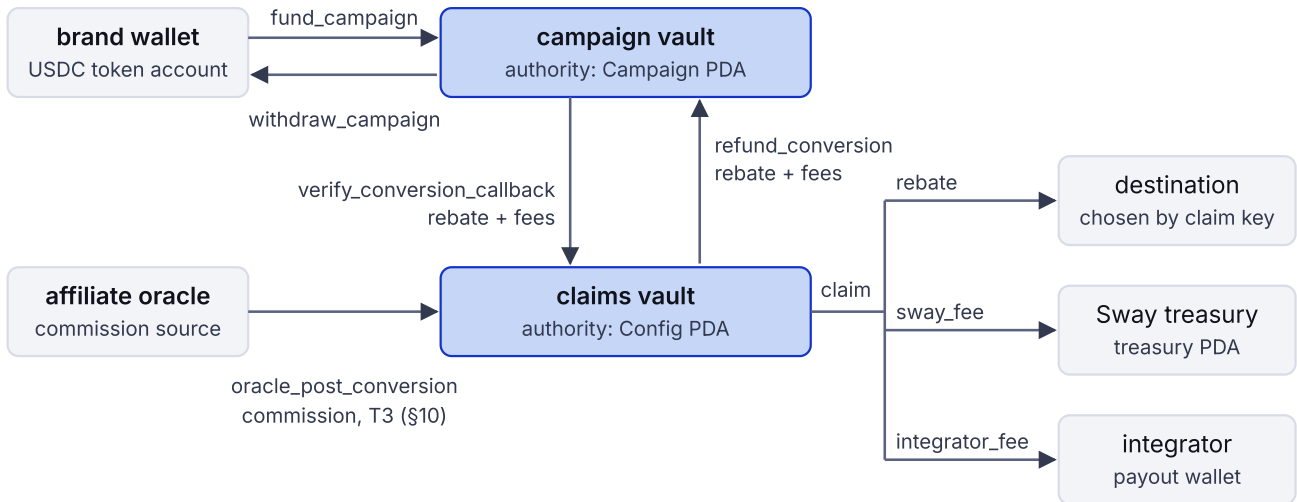


Figure 8.1 — Every USDC movement in brand-direct settlement. The campaign vault is debited once, at verification. The claims vault pays out once per conversion, either back to the vault (refund) or forward to three recipients in a single claim.

8.2 The lock

Each campaign fixes a lock length in whole days, `lock_days` (u16), when it is created. `create_campaign` refuses a lock shorter than the merchant's registered return window (`lock_days ≥ merchant.return_window_days`, error `LockTooShort`) and stores a copy of that return window on the campaign. The lock starts at verification, not at purchase:

$$\text{unlock_at} = \text{verified_at} + \text{lock_days} \times 86400$$

where `verified_at` is the chain's Unix time when `verify_conversion_callback` runs. Verification always follows the purchase, so the escrow stays reversible for at least the merchant's return window counted from the sale. For example, a campaign with `lock_days = 30` verified at time `T` unlocks at `T + 2,592,000`.

The two exits from escrow split time at one instant with no overlap and no gap:

<code>refund_conversion</code>	requires	<code>now < unlock_at</code>
<code>claim</code>	requires	<code>now ≥ unlock_at</code>

At the second `unlock_at`, a refund fails with `RefundWindowClosed` and a claim succeeds. The API projects the same boundary: an escrowed conversion is locked with `refundAllowed: true` and `refundableUntil = unlockAt` before it, and `claimable` from it on. The worker does not poll idle escrows during the lock; a locked conversion generates background work only when someone submits a refund notice or a claim.

Verification also extends the merchant's liability window to `max(liability_until, unlock_at + dispute_seconds)`, which keeps the merchant's bond in place until disputes about the conversion can no longer be opened (§11). T3 conversions use the affiliate network's return period instead of a campaign lock (§10).

8.3 Refund notices

A returned order reverses its conversion through a refund notice signed by the merchant's **current** receipt signer. The notice is another fixed 128-byte message:

offset	size	field	encoding
0	8	domain	ASCII "SWAYREF1"
8	32	conversion	Conversion account address
40	32	receipt_hash	SHA-256 of the original 128-byte receipt
72	32	program	sway_exchange program ID
104	4	signer_epoch	u32 LE, the merchant's current epoch
108	20	reserved	zero

The notice has no free fields. Given a conversion, the program can reconstruct the only valid notice, `refund_message(conversion, conversion.receipt_hash, merchant.signer_epoch)`, and requires the submitted bytes to equal it exactly. `refund_conversion(notice)` takes the configuration, intent, merchant, campaign, conversion, campaign vault, claims vault and instructions sysvar accounts; the conversion must name the intent and campaign (`has_one`), and the campaign must name the merchant and vault. No account signs except the fee payer. Authority comes from the Ed25519 precompile (§7.6), whose expected key is `Merchant.receipt_signer`. The program checks, in order:

1. The conversion is escrowed and `now < unlock_at`.
2. `verify_ed25519` passes for the merchant's current receipt signer over the notice.
3. The notice equals the reconstructed SWAYREF1 message.

On success it transfers the conversion's full `total()` from the claims vault back to the campaign's own vault, signed by the Config PDA, and reverses the campaign accounting: `spent_total` falls by the total, and so does `spent_today` if the refund lands on the same UTC day as the verification. The conversion becomes `refunded` and `Intent.open_conversions` falls by one.

These rules have direct consequences. A refund reverses the whole debit, fees included: the brand pays nothing for a returned order, and neither Sway nor the integrator is paid. The returned funds go to the vault the conversion was paid from and become withdrawable or available for new

conversions (§9.3). A notice can succeed only once, because it requires escrowed and leaves refunded. Because the notice embeds the signer epoch, a rotation makes earlier unsubmitted notices invalid; the new signer signs a fresh notice under the new epoch.

The merchant submits a notice with `POST /v1/conversions/:id/refund` (operation `refundConversion`). The body carries `signerPublicKey`, `signatureBase64`, the 128-byte `refundMessageBase64`, and a JSON notice object (`conversionId`, `intentId`, `slot`, `orderHash`, `issuedAt`, `nonce`) that the service matches against the conversion it names. The signature covers the 128-byte message. The service accepts the request only for a T2 conversion, only from the key the chain currently records as the receipt signer, only for a message that reconstructs exactly, and only while the conversion is escrowed and `unlockAt` is in the future. It then queues a `settlement_refund` job, which submits `[Ed25519 precompile, refund_conversion]` with the relayer as fee payer. If a relayed refund fails without settling, a renewed notice replaces it in the same refund record. T3 reversals come from the affiliate oracle, not the merchant (§10).

8.4 Claim keys

A claim key is an Ed25519 keypair created for one intent, on the device that created the intent. The local MCP server (`@sway/mcp`) generates it when it encrypts the intent and stores its secret seed together with the intent's x25519 secret in the operating system's keychain; a hosted agent does the same in its own backend. The public key is posted with the intent (`post_intent` argument `claim_key`, which may not be the all-zero key) and stored as `Intent.claim_key`. The service refuses a second intent with a claim key already in use, so every claim key belongs to exactly one intent.

The claim key signs three kinds of thing, and never a Solana transaction:

Signed object	Purpose
HTTP challenge, purpose <code>claim_read</code>	Reading claimables and claim status (<code>POST /v1/claims/query</code> , <code>POST /v1/claims/:id/query</code>); balances are never public lookups
HTTP challenge, purpose <code>claim</code>	Authorizing the claim request, one use, bound to the request hash and the conversion id
SWAYCLM1 message	The authorization the program verifies on chain (§8.5)

Claim keys also authorize opening a dispute (§11). Because the key signs messages rather than transactions, it holds no SOL and no USDC, and the user never needs a funded wallet to receive a rebate. The payout address is chosen at claim time, not at intent time, so the intent carries no wallet. A fresh key per intent means two claimables share no on-chain identity unless the user sends both to the same destination, in which case the link is the user's own choice.

8.5 The claim authorization

The claim message binds one payout of one conversion to one destination for a bounded time:

offset	size	field	encoding
0	8	domain	ASCII "SWAYCLM1"
8	32	program	sway_exchange program ID
40	32	conversion	Conversion account address
72	32	destination	wallet that owns the receiving token account
104	8	expires_at	i64 LE, Unix seconds on the chain clock
112	16	nonce	16 random bytes

All 128 bytes carry meaning; there are no reserved bytes. `claim(authorization)` takes the configuration (`has_one` claims vault and treasury), intent, conversion (`["conversion", intent, [slot]]`, `has_one = intent`), claims vault, destination token account, treasury, integrator token account (whose owner must equal `Intent.integrator`) and the instructions `sysvar`. The program checks, in order:

1. The conversion is escrowed; otherwise `AlreadyConsumed`.
2. `now ≥ unlock_at`; otherwise `ClaimLocked`.
3. `verify_ed25519` passes for `Intent.claim_key` over the authorization.
4. `expires_at ≥ now`.
5. The authorization equals `claim_message(conversion, destination.owner, expires_at, nonce)`, rebuilt from the accounts actually passed.

It then makes three transfers from the claims vault, all signed by the Config PDA: rebate to the destination, `sway_fee` to the treasury and `integrator_fee` to the integrator's account. The conversion becomes `claimed`, `Intent.open_conversions` falls by one, and when it reaches zero the intent becomes settled.

Each field in the message closes a specific gap:

- **Program and conversion.** An authorization works for one escrow on one deployment. Each conversion of an intent needs its own signature.
- **Destination.** The program compares bytes 72–103 with the owner of the token account it is about to pay. Anyone who copies a signed authorization out of a pending transaction can only resubmit a payment to the same wallet. This is why claims use a signature rather than a revealed secret: front-running a claim gains the front-runner nothing.
- **Expiry.** It bounds how long a signed authorization stays usable, measured on the chain clock (`Clock::unix_timestamp`). The MCP client sets it to the chain time plus `claimTtlSeconds`, 120 seconds by default.
- **Nonce.** The program keeps no nonce registry. A conversion pays once because `claim` moves it from `escrowed` to `claimed`. The nonce makes every authorization a distinct message, so a renewed authorization never collides with an earlier one, and its SHA-256 (`authorizationHash`) gives the service a unique key for idempotency.

The agent submits the claim with POST `/v1/claims` (operation `claim`): `claimPublicKey`, `claimableIds` (exactly one id in `v0`), `destination`, the HTTP challenge proof, `claimMessageBase64` and `claimSignatureBase64`. The service rebuilds the message from the conversion account, `destination`, `expiry`, `nonce` and `program ID` and verifies the signature. It then checks that the conversion is escrowed and unlocked and that the expiry has not passed, and consumes the one-use HTTP proof in the same database transaction that creates the claim record and its `settlement_claim` job. The response is a `ClaimResult` whose status moves through `queued`, `submitted` and `confirmed` (or `failed`), with `totalAmount` equal to the rebate.

The service pays the destination wallet's associated USDC token account. It derives that account with off-curve owners allowed, so the destination may be a program-derived address such as a multisig vault, and creates the account in the same transaction if it does not exist. If an authorization expires before its transaction lands, the claim fails with no payout and the claim key signs a fresh authorization; the MCP client does this automatically. A request with a different destination is refused while an earlier authorization is still valid or its transaction is unresolved, so two destinations are never in flight for one conversion.

8.6 Transaction fees and rent

The relayer is the fee payer for every settlement transaction an agent or a merchant's signer triggers: `record_conversion`, the verification callback, `refund_conversion`, `claim` and the closes of §8.8. Users and receipt signers never need SOL. A relayed claim transaction contains, in order:

1. `createAssociatedTokenAccountIdempotent` for the destination's USDC account, if the service is paying the associated account.
2. The same for the integrator's associated USDC account.
3. The Ed25519 precompile carrying the claim key, the 128-byte authorization and its signature.
4. `claim`, which reads the precompile at the index immediately before its own.
5. `request_nonce` with 16 random bytes, a no-op instruction that makes two otherwise identical transactions distinct.

The relayer pays the transaction fee and the rent for any token account the first two instructions create. The `claim` instruction has no fee argument and no relayer account; it moves exactly the integers recorded on the conversion, so the user receives the full offered rebate. Whether and how a deployment recovers relaying costs is a configuration point (`relayerFeePolicy` in `PROTOCOL_PARAMETERS.requiredConfiguration`), outside the instruction.

`record_conversion` also makes the relayer the rent payer for the two accounts it creates, the conversion (296 bytes) and the order guard (41 bytes), and records it in `Conversion.payer`. That field fixes where the rent returns when the accounts close. The relay outbox commits the signed transaction bytes before broadcasting, rebroadcasts only identical bytes, and treats an operation as done only at finalized commitment (§13 describes the outbox).

8.7 Purchase deadline, settlement deadline and grace

Two deadlines are written into every intent at `post_intent` and never change. The purchase deadline is chosen by the agent and committed inside the encryption nonce: the first eight bytes of the intent's 16-byte nonce are the deadline as a u64 little-endian Unix time, followed by eight random bytes. The program accepts it only within the configured window, and derives the settlement deadline from it:

```
now < purchase_deadline ≤ now + Config.purchase_window_secs
settlement_deadline = purchase_deadline + Config.settlement_grace_secs
```

Both window lengths are positive u32 values set at `initialize`; their values are deployment configuration. Because the deadline is inside the encrypted input's nonce, it is covered by the intent's input commitment and cannot be altered without re-encrypting.

Action	Time rule
<code>getVoucher</code>	<code>chain time < purchase_deadline</code>
Receipt timestamp	<code>created_at ≤ timestamp < purchase_deadline</code> , and <code>timestamp ≤ now</code>
<code>record_conversion</code> , <code>requeue_conversion</code>	<code>now ≤ settlement_deadline</code>
<code>verify_conversion_callback</code>	none: a recorded reservation can always complete
<code>oracle_post_conversion (T3)</code>	<code>purchase time in [created_at, purchase_deadline)</code> , <code>now ≤ settlement_deadline</code>
<code>refund_conversion</code>	<code>now < unlock_at</code>
<code>claim</code>	<code>now ≥ unlock_at</code> and <code>expires_at ≥ now</code>
<code>close_intent</code> , <code>close_conversion</code>	<code>now > settlement_deadline</code>

The purchase deadline bounds when the shopper may buy. The grace period after it gives attestation time to arrive: the merchant's signer issues receipts after the sale, and affiliate networks report commissions later still, so a deployment that enables the cashback path sets the grace to cover the network's payout lag (\$10). After the settlement deadline no new conversion can be recorded for the intent; the service marks an unrecorded receipt for it as a terminal failure (`SETTLEMENT_WINDOW_CLOSED`). The lock and the claim are not bounded by the settlement deadline. Escrow follows `unlock_at`, which may fall before or after it, and an escrowed conversion stays claimable with no expiry.

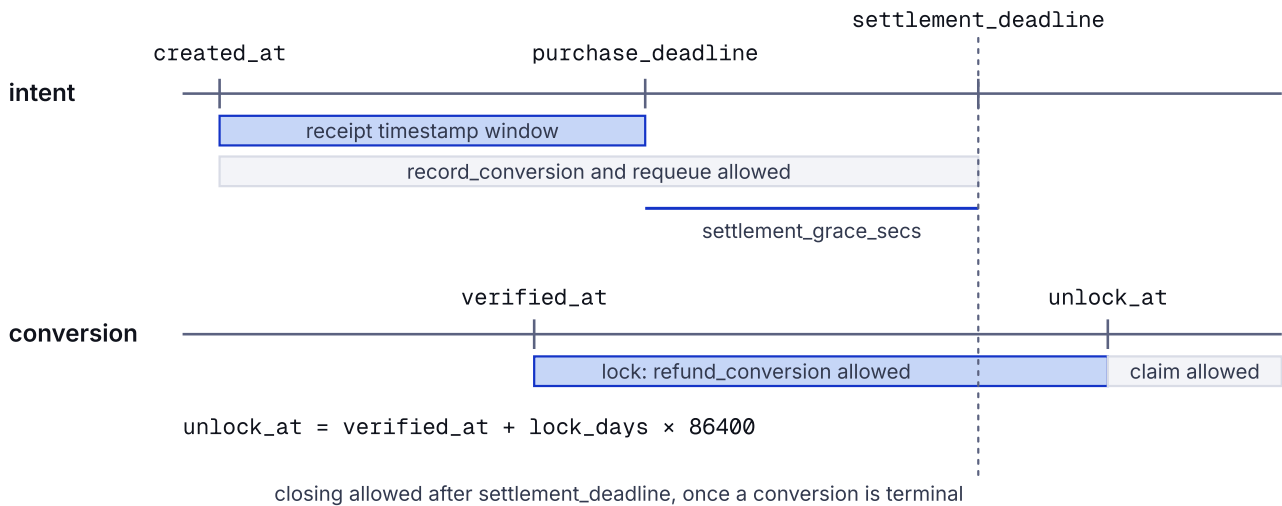


Figure 8.2 — The intent's two deadlines are fixed at `post_intent`; the conversion's lock is fixed at verification. The two clocks are independent: `unlock_at` may fall before or after `settlement_deadline`.

8.8 Closing finished accounts

Every bookkeeping account the purchase-to-claim path creates is paid for with rent, and every one of them is closed once it can no longer change. Closing returns the account's lamports to the payer recorded at creation, never to an address the caller chooses: each close instruction constrains its rent recipient with `address = conversion.payer` or `address = intent.payer`.

Instruction	Closes (bytes)	Conditions	Signer
<code>close_conversion</code>	Conversion (296), OrderGuard (41)	T2; status refunded, claimed, rejected or cancelled; now > <code>settlement_deadline</code>	none
<code>close_oracle_conversion</code>	Conversion (296), OracleEvent (81)	T3; status refunded or claimed; now > <code>settlement_deadline</code>	none
<code>close_intent</code>	Intent (1309), IntentBook (3672), InputReplay (9)	<code>open_conversions = 0</code> ; now > <code>settlement_deadline</code>	the intent's recorded agent

The conversion closes are permissionless: anyone may submit them, and the rent can only go to the recorded payer. An escrowed conversion has no close path. It holds a claim, and it ends only by claim or refund. The deadline used by `close_conversion` is the copy of the intent's settlement deadline stored on the conversion at record time, so a conversion can close even after its intent account is gone.

Closing the order guard does not reopen the order to replay. An old receipt names its intent account and carries a timestamp before that intent's purchase deadline. `record_conversion` accepts it only up to that intent's settlement deadline, which has passed by the time the guard can close. An account later created at the same intent address would carry a `created_at` later than the old receipt's timestamp, and the receipt would fail the timestamp check. The service also keeps

permanent merchant/order and affiliate-event uniqueness in PostgreSQL after the on-chain guards close.

The worker closes accounts on its own. Once an intent's settlement deadline has passed, its reconcile pass submits `close_conversion` for each terminal conversion and `close_intent` for each intent with no open conversions and the relayer as its recorded agent. It waits for finality, confirms that the account is gone, and records the close in its read-only history. Rent that the relayer advanced for an intent's accounts and conversions comes back to it the same way.

9. Fees and settlement

In brand-direct settlement only the brand pays Sway, and only when a purchase it bid for has been verified. The amount is a fixed fraction of the rebate it offered. This chapter gives the fee formula, the integer rounding rules, the accounting that keeps a campaign's vault, reservations and spend consistent, the caps that bound them, and three worked examples using the protocol's real parameters. Amounts are in USDC; on chain and in the API they are unsigned 64-bit integers of base units, with 1 USDC = 1,000,000 base units (`settlement.baseUnitsPerUsdc`).

9.1 What the brand pays

Parameter (<code>PROTOCOL_PARAMETERS.brandDirectFees</code>)	Value
<code>basis</code>	<code>rebate</code>
<code>chargedOn</code>	<code>verified-conversion</code>
<code>totalBps</code>	1500
<code>swayBps</code>	1000
<code>integratorBps</code>	500
<code>userReceivesFullOfferedRebate</code>	<code>true</code>

For a verified T2 conversion with rebate `rebate` (the first-price rebate of the sealed offer, \$7.8):

```
sway_fee      = floor(rebate × 1000 / 10000)
integrator_fee = floor(rebate × 500 / 10000)
brand_debit    = rebate + sway_fee + integrator_fee
user_receives  = rebate
```

The fee is a percentage of the rebate, not of the purchase price. The shopper pays the merchant the purchase price through the merchant's own checkout; that payment never passes through Sway, and Sway takes no share of it. The user receives the whole rebate the brand offered, and both fees are added on top, so the brand pays the rebate plus 15%: 10% to the Sway treasury and 5% to the integrator, the developer of the agent that posted the intent.

The brand pays nothing for having an offer in the book, for being matched, or for appearing in a top three. It also pays nothing for a conversion that is rejected or cancelled, and a refund inside the lock returns the full debit including both fees (§8.3).

The fee rates live in the Config account as `sway_bps` and `integrator_bps`. `initialize` writes them from the generated constants `DIRECT_SWAY_BPS = 1000` and `DIRECT_INTEGRATOR_BPS = 500`.

`update_config` has no fee argument, so changing a rate requires a program upgrade, signed by the program's upgrade authority, the deployer key (§16.6).

The integrator is identified per intent. When the service posts an intent it records the integrator's registered payout wallet as `Intent.integrator`, and at claim time the program requires the integrator's token account to be owned by that key. An agent developer is therefore paid only for intents its own agents posted.

9.2 Rounding in base units

All settlement arithmetic is integer arithmetic on base units. The API carries amounts as decimal strings constrained to the u64 range and never as floating-point dollars. The program computes each fee with

```
fee(amount, bps) = u64( floor( u128(amount) × bps / 10000 ) )
```

The product is widened to 128 bits before division, so no intermediate value can overflow. Every sum and difference that follows uses checked arithmetic and fails with `Overflow` instead of wrapping.

Each fee is rounded down separately. Each floor drops less than one base unit, so the brand's debit falls in a narrow band below the exact 115%:

$$0 \leq 1.15 \times \text{rebate} - \text{brand_debit} < 2 \quad (\text{base units})$$

The user is never short: `user_receives` is the rebate itself, with no rounding. Any fractional base unit stays with the brand. At very small rebates the fees round to zero: `sway_fee` is 0 for rebates under 10 base units and grows by one base unit for every 10, while `integrator_fee` is 0 under 20 base units and grows by one for every 20.

The rounding rule is part of the configuration. `initialize` takes a `rounding_policy` byte and rejects every value except 1, floor to the base unit, with `ConfigurationRequired`; the public parameter list leaves `feeRoundingPolicy` to deployment configuration, and the program admits only this one policy. The service computes the same integers with `BigInt` truncating division (`rebate * 1000n / 10000n`, `rebate * 500n / 10000n`) and refuses to continue a settlement if the conversion recorded on chain holds different values.

9.3 Campaign accounting

A campaign's money sits in its vault, a USDC token account owned by the campaign PDA, and the Campaign account keeps the ledger beside it. §5.1 defines the fields and the conditions each instruction checks. Settlement is where those fields move, and every instruction that touches a campaign moves them in a fixed way. Writing `T` for a conversion's total debit:

Instruction	Vault balance	pending_total	spent_total	spent_today	funded_net
fund_campaign(a)	+a				+a
withdraw_campaign(a)	-a				-a
record_conversion, requeue_conversion		+T			
verification callback, valid	-T	-T	+T	+T	
verification callback, invalid		-T			
cancel_pending_conversion		-T			
refund_conversion	+T		-T	-T if same UTC day	

Read down a column and the ledger's identities follow. The vault moves only with funding, verification and refunds, and `spent_total` moves with exactly the same verification and refund amounts in the opposite direction, so

```
funded_net    = spent_total + vault_balance
vault_balance = pending_total + withdrawableBalance
```

for any campaign whose vault receives USDC only through these instructions. The API reports the second split directly, as `pendingLiability` and `withdrawableBalance` (§5.1.5). Escrowed money has already left for the claims vault, so it appears in neither. Reserved money is still in the vault but is excluded from the withdrawable balance until its verification resolves. Everything else is the brand's to withdraw at any time. A campaign whose vault falls below the configured floor (`Config.vault_floor`) is reported as underfunded and drops out of the active set for new intents (§5.2.5); conversions it has already won settle as usual.

Three details of the ledger matter in practice:

- **funded_net only moves with deposits and withdrawals.** Spending never reduces it, and $\text{spent_total} + \text{pending_total} \leq \text{funded_net}$ always holds, so `funded_net` bounds the campaign's lifetime net debit. Together with the campaign cap below, a campaign can never spend more than 5,000 USDC net of refunds.
- **The daily cap counts reservations.** A reservation is checked against today's verified spend plus every reservation outstanding, including any made on an earlier day. Verified spend is attributed to the UTC day of verification, not of purchase.
- **Refunds restore capacity.** A refund returns the debit to the vault and to `spent_total` at once. It reduces `spent_today` only when the campaign's current spend day is the day the conversion verified.

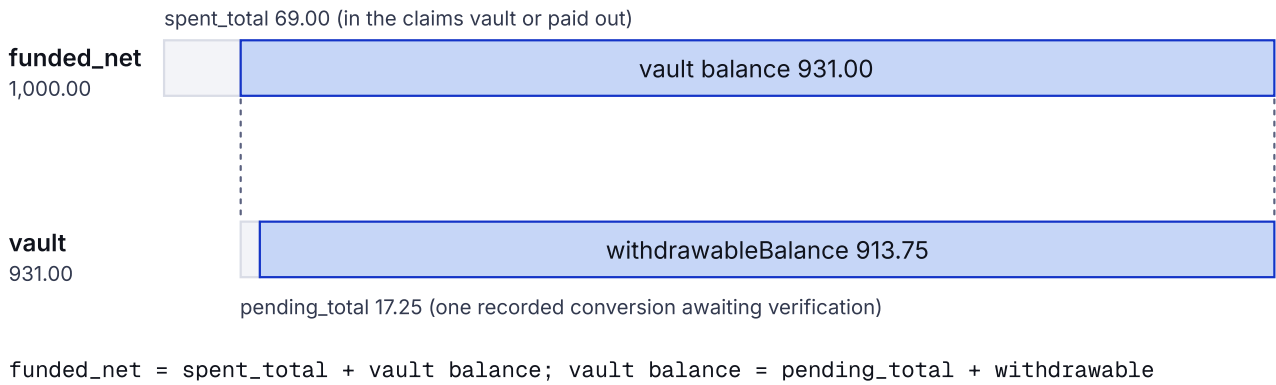


Figure 9.1 — The campaign of example 3 (§9.5) after four verified conversions and one reservation, drawn to scale. Only the accented part of the vault can leave by `withdraw_campaign`.

9.4 Caps in settlement

§5.1.2 defines the four caps. Settlement meets each of them here:

Cap	Value	Checked at settlement by
REBATE_CAP	50,000,000 base units (50 USDC) per conversion	<code>record_conversion</code> (T2), <code>oracle_post_conversion</code> (T3 user share)
<code>per_conversion_cap</code>	brand's choice, at most REBATE_CAP	<code>record_conversion</code> ; offers above it were already rejected at ingest
<code>daily_cap</code>	brand's choice, at most CAMPAIGN_CAP	the reservation in <code>record_conversion</code> and <code>requeue_conversion</code> , and the valid callback
CAMPAIGN_CAP	5,000,000,000 base units (5,000 USDC) of <code>funded_net</code>	indirectly, through $\text{spent_total} + \text{pending_total} \leq \text{funded_net}$

Both rebate caps limit the rebate alone; fees ride on top, so one conversion can debit at most 57.5 USDC. A campaign funded to the full 5,000 USDC can pay $\text{floor}(5,000 / 57.5) = 86$ conversions at the maximum rebate, a debit of 4,945 USDC; the remaining 55 USDC cannot cover an 87th. Because the cap applies to `funded_net`, which spending never lowers, a brand that wants more than 5,000 USDC of lifetime budget runs a second campaign.

A cap that blocks a reservation fails `record_conversion` with `CapExceeded` (or `Underfunded` for the funding conditions) and changes nothing on chain. The service keeps the receipt's conversion in `verification_pending` and retries the record with backoff until it fits or the intent's settlement deadline passes, after which the receipt is a terminal failure (§8.7). Within that window, a blocked receipt waits for the next UTC day, a refund, or new funding.

9.5 Worked examples

The amounts below are examples; the rates, caps and rounding are the protocol's.

Example 1: a standard conversion

For example, a pair of running shoes priced at 120 USDC carries a sealed offer with a 15 USDC rebate, and the shopper buys it.

Quantity	Base units	USDC
Receipt amount (purchase price, paid to the merchant)	120,000,000	120.00
rebate	15,000,000	15.00
$\text{sway_fee} = \text{floor}(15,000,000 \times 1000 / 10000)$	1,500,000	1.50
$\text{integrator_fee} = \text{floor}(15,000,000 \times 500 / 10000)$	750,000	0.75
brand_debit	17,250,000	17.25

At verification, 17,250,000 base units move from the campaign vault to the claims vault. With a 30-day lock (for example), `unlock_at` is 2,592,000 seconds after verification. If the shoes come back before then, the merchant's refund notice returns all 17,250,000 to the campaign vault and nobody is paid. Otherwise the claim pays 15,000,000 to the user's destination, 1,500,000 to the treasury and 750,000 to the integrator in one instruction. The shopper's net price for the shoes is 105 USDC, and the brand's cost is 14.375% of the price it was paid. No rounding occurs, because the rebate is a multiple of 20 base units.

Example 2: rounding

For example, a rebate of 12,345,679 base units (12.345679 USDC).

Quantity	Exact value	Recorded
sway_fee	1,234,567.9	1,234,567
integrator_fee	617,283.95	617,283
Fees together	1,851,851.85 (exactly 15%)	1,851,850
brand_debit		14,197,529

The brand pays 1.85 base units less than an exact 15%; the user receives all 12,345,679. At the small end, a rebate of 19 base units gives `sway_fee` = 1, `integrator_fee` = 0 and a debit of 20.

Example 3: a campaign over one UTC day

For example, a brand creates a campaign with `per_conversion_cap` 20 USDC, `daily_cap` 100 USDC and `lock_days` 30, funds it with 1,000 USDC, and every conversion carries a 15 USDC rebate, so each debit τ is 17.25 USDC. All steps happen on one UTC day.

Step	Event	Vault	Pending	spent_total	spent_today	Withdrawable
1	fund_campaign 1,000	1,000.00	0	0	0	1,000.00
2	four receipts recorded	1,000.00	69.00	0	0	931.00
3	four valid callbacks	931.00	0	69.00	69.00	931.00
4	fifth receipt recorded	931.00	17.25	69.00	69.00	913.75
5	sixth receipt: $69.00 + 34.50 = 103.50 > 100$, CapExceeded	931.00	17.25	69.00	69.00	913.75
6	fifth callback, valid	913.75	0	86.25	86.25	913.75
7	merchant refunds the second conversion	931.00	0	69.00	69.00	931.00
8	sixth receipt retried: $69.00 + 17.25 = 86.25 \leq 100$	931.00	17.25	69.00	69.00	913.75

Step 4 is the state drawn in Figure 9.1. At step 5 the sixth `record_conversion` fails, nothing changes on chain, and the service retries it (§9.4). It fits at step 8, after the refund at step 7 has lowered the day's spend; on the next UTC day it would fit regardless, because `today_spent` restarts at 0. At step 4 the brand could withdraw up to 913.75 USDC, and every withdrawal condition would still hold: with 900 withdrawn, the vault keeps $31.00 \geq 17.25$ pending, and `funded_net` falls to $100.00 \geq 69.00 + 17.25$.

9.6 When each party is paid

Party	Paid by	When
Brand (debited)	<code>verify_conversion_callback</code> , campaign vault $\rightarrow$ claims vault	At verification
User	<code>claim</code> , rebate $\rightarrow$ destination chosen by the claim key	At or after <code>unlock_at</code>
Sway	<code>claim</code> , <code>sway_fee</code> $\rightarrow$ treasury PDA	Same instruction as the user
Integrator	<code>claim</code> , <code>integrator_fee</code> $\rightarrow$ token account owned by <code>Intent.integrator</code>	Same instruction as the user

The brand's debit happens once, and a refund before `unlock_at` can reverse it entirely. The three payouts happen together in the claim that follows, so the fee split is settled atomically with the user's rebate. The treasury accumulates Sway's fees, and they leave it only through the buyback of \$9.8.

Brands see the same arithmetic through the API. Each conversion reports `rebateAmount`, `swayFee`, `integratorFee` and `totalCampaignDebit` (their sum for T2). `GET /v1/campaigns/:id/spend` returns daily points over an inclusive range of UTC dates, each with `rebateAmount`, `feeAmount`, `totalDebit` and `conversionCount` for conversions that are escrowed, claimed or disputed.

9.7 The cashback split

The cashback fallback (T3) inverts the arrangement (`cashbackSplit: { userBps: 7000, integratorBps: 1000, swayBps: 2000 }`). An affiliate commission, not a brand's offer, funds the conversion, and Sway divides it rather than adding fees on top. Sway's 20% and the integrator's 10% are each floored, and the user receives the remainder, so the fractional base units go to the user, the opposite of brand-direct rounding, and the whole commission is distributed. The oracle deposits the entire commission into the claims vault, no campaign is debited (`totalCampaignDebit` is 0), and the claim is the one of \$8.5. For example, a commission of 8.40 USDC splits into 5.88 USDC for the user, 0.84 for the integrator and 1.68 for Sway. §10.8 gives the formulas, and §10.6 the oracle's checks and the T3 lock.

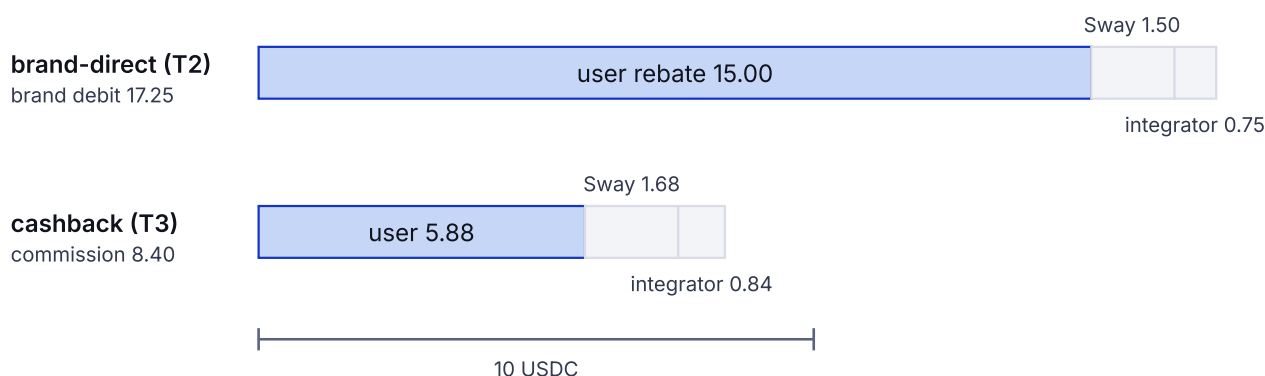


Figure 9.2 — Example amounts (USDC) on one scale. Brand-direct adds 15% on top of the rebate the user receives; cashback divides a commission 70/10/20.

9.8 SWAY: buyback and burn

\$SWAY is Sway's token. Its supply is fixed at 1,000,000,000 and was sold entirely in the public launch on equal terms: there is no presale, team, investor, airdrop or reserved allocation, and nothing vests. Every amount inside the protocol stays in USDC. What connects SWAY to the protocol is the treasury: Sway's whole share of every fee buys SWAY on the market, and the SWAY bought is burned.

Parameter	Value
Name and ticker	Sway, SWAY (<code>settlement.token.ticker</code>)
Supply	1,000,000,000, fixed; no further minting (<code>supply, fixedSupply: true</code>)
Launch	100% to the public launch (<code>launch: "fair"</code>)
Funded by	<code>sway_fee</code> of every T2 claim (1,000 bps of the verified brand-direct rebate); Sway's 2,000 bps of each T3 commission joins it once cashback runs
Use of funds	Market buy of SWAY, then burn (<code>feeShare: "buyback-burn"</code>)
SWAY mint	<code>Config.sway_mint</code> , mirrored by the services as <code>SWAY_MINT</code>
Treasury outflow	<code>buyback_begin</code> only, in a transaction whose last instruction is <code>buyback_finish</code>
Signer	<code>Config.buyback_authority</code> , set to the relayer at initialize
Trigger	Treasury balance $\geq$ <code>BUYBACK_MIN_USDC</code> (default 10 USDC), at most once per <code>BUYBACK_INTERVAL_SECONDS</code> (default 3,600)
Route	Jupiter ExactIn quote, USDC $\rightarrow$ SWAY, slippage <code>BUYBACK_SLIPPAGE_BPS</code> (default 100)

The treasury fills at claim time. `claim` pays the user, the integrator and the treasury PDA in one instruction (§9.6), so the treasury fills with the Sway fees of claimed conversions as they are claimed. Over any period, the USDC that goes to the burn is

$$\text{burn_usdc} = \sum_{\text{T2}} \text{floor}(\text{rebate} \times 1000 / 10000)$$

summed over the conversions claimed in that period. Once cashback runs, each claimed T3 conversion adds $\text{floor}(\text{commission} \times 2000 / 10000)$ to the sum. A refunded conversion never reaches `claim`, so its fee never enters the sum.

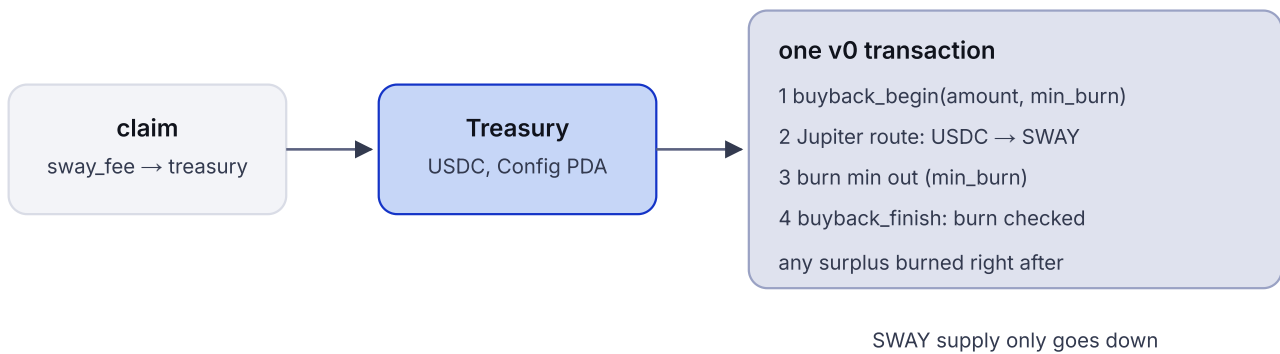


Figure 9.3 — The buyback. Fees reach the treasury at claim; one transaction moves treasury USDC through a Jupiter route into SWAY, burns it, and ends with `buyback_finish`, which checks the burn.

The buyback is one v0 transaction that the worker's keeper builds and signs. Its key is `Config.buyback_authority`, which initialize sets to the relayer and which is the only key `buyback_begin` and `buyback_finish` accept (§12.5). The Config admin has no instruction that moves treasury USDC; `buyback_begin` is the treasury's only debit. Once the treasury holds at least `BUYBACK_MIN_USDC` and `BUYBACK_INTERVAL_SECONDS` have passed since the last run, the keeper asks Jupiter (`lite-api.jup.ag`) for an `ExactIn` quote of the whole treasury balance from USDC to SWAY at `BUYBACK_SLIPPAGE_BPS`, with the route capped by `maxAccounts` so the transaction fits. It sets `min_burn` to the quote's minimum output (`otherAmountThreshold`) and compiles, in order:

1. compute-budget instructions for the unit limit and price;
2. `buyback_begin(amount, min_burn)`, which records the keeper's USDC balance, the SWAY mint's supply, `min_burn` and the current slot in the `BuybackState` PDA `["buyback", config]`, then moves amount from the treasury to the keeper's USDC account;
3. Jupiter's setup, swap and cleanup instructions, which deliver SWAY to the keeper's SWAY account;
4. an SPL burn of `min_burn` SWAY from that account;
5. `buyback_finish()`.

Before it moves anything, `buyback_begin` requires `Config.sway_mint` to be set, `amount` and `min_burn` to be positive and `amount` to be at most the treasury balance. Through the instructions `sysvar` it requires that it is a top-level instruction and that the transaction's last instruction is `buyback_finish` over the same keeper, `Config`, `BuybackState`, keeper USDC account and SWAY mint. The keeper refuses a route that invokes Sway, that needs any signer other than the keeper, or that writes to `Config`, the treasury or `BuybackState`.

`buyback_finish` enforces the outcome. It must be the last instruction, it must run in the slot recorded by `buyback_begin`, the keeper's USDC balance must be back at or below its balance before the transfer, so the whole amount went into the route, and the SWAY supply must be lower by at least `min_burn`. It then emits `BuybackBurned { usdc_in, sway_burned }` and closes `BuybackState` to the keeper. If any check fails, the transaction fails as a whole, the treasury transfer included, so the transaction either burns at least the quoted minimum or does not happen at all. If the route fills

above the minimum, the keeper burns the surplus in a follow-up transaction and confirms from finalized state that its SWAY account is empty. GET /v1/protocol/buyback reports the SWAY mint, the treasury's finalized balance and the totals of USDC spent and SWAY burned, summed from finalized BuybackBurned events and surplus burns.

For example, a month in which 2,000 brand-direct conversions with a 15 USDC rebate are claimed sends $2,000 \times 1.50 = 3,000$ USDC to the burn. These figures are an example.

10. Cashback fallback and the affiliate oracle

The program's T3 instructions and the API's cashback routes described here are in place, and the path turns on when the deployment connects an affiliate network; until then `cashbackOnly` intents, tracking links and oracle batches answer `CONFIGURATION_REQUIRED`.

Brand-direct offers (§5–§9) need a funded campaign whose sealed rule admits the intent. When no such offer exists, or when the agent skips matching, Sway still pays the user. It acts as the publisher on an affiliate network, ties the network's commission to the intent through a random sub-ID, and escrows the commission on chain as a T3 conversion. The T3 path runs no MPC computation. It reuses the intent account, the claim key, the claims vault and the `claim` instruction of the brand-direct path, and replaces the merchant receipt with an attestation from the Sway affiliate oracle.

10.1 When an intent falls back

An intent reaches the cashback path in one of three ways. Each leaves an `Intent` account on chain with its claim key and integrator recorded, and that is all the T3 path needs.

Fallback reason	Trigger	On-chain <code>Intent.status</code>	API status
<code>COMPUTATION_EXPIRED_TWICE</code>	Attempt 1 expires after 180 slots, the agent calls <code>retryIntent</code> , and attempt 2 also expires. The worker then submits <code>retry_intent</code> again, which moves the intent to fallback.	2 (fallback)	<code>cashback_fallback</code>
<code>NO_ELIGIBLE_OFFERS</code>	Matching completes and no slot of the captured book admits the intent.	1 (matched)	<code>cashback_fallback</code>
<code>CASHBACK_ONLY</code>	The agent posts with <code>cashbackOnly: true</code> . The service registers the intent on chain and queues no matching job.	0 (queued, never matched)	<code>cashback_fallback</code>

A cashback-only intent is still encrypted on the device, and its nonce still commits the purchase deadline (§4). `post_intent` runs unchanged: it fixes the purchase window and the claim key on chain and consumes the same one-use (`sender`, `nonce`) guard as a matched intent. The mode is immutable for an intent ID: repeating `postIntent` with the other mode returns `IDEMPOTENCY_CONFLICT`. The service accepts `cashbackOnly` only when the affiliate integration is configured, and otherwise returns `CONFIGURATION_REQUIRED`. The `find_offers` MCP tool (§13.8) forwards the same option and performs the single retry on its own: when a poll returns `expired` on attempt 1, it calls `retryIntent` once.

10.2 The public cashback table

`GET /v1/cashback/rates` is public and paginated (limit 1–100, default 50, optional shard filter). Only rates with a merchant domain are listed.

Field	Meaning
id	Rate identifier, quoted when requesting a tracking link
merchantName, merchantDomain	The advertiser and the exact hostname that destinations must use
shard	Category shard the rate serves
commissionBps	Network commission on the purchase, 0–10,000 basis points
userShareBps, integratorShareBps, swayShareBps	Fixed at 7000, 1000 and 2000 (cashbackSplit)
status	available, paused or expired
validUntil	The rate's expiry
sample	Marks an illustrative entry

A rate is expired once its `expiresAt` has passed. It is available only when it is enabled, belongs to the configured network (`AFFILIATE_NETWORK`), has reached its `effectiveAt`, carries an expiry, and names a merchant domain listed in `AFFILIATE_ALLOWED_DESTINATION_HOSTS`. A `sample` rate is available only when `AFFILIATE_ALLOW_SAMPLE_RATES` is true. Every other rate is paused.

The table is public, so reading it tells Sway nothing about the user's preferences; on this path Sway learns only the merchant the agent asks to link to. The published shares let the agent estimate the user's cashback before it links:

```
expected_cashback = price × commissionBps / 10000 × userShareBps / 10000
```

For example, a \$120 order at an assumed 7% commission yields an \$8.40 commission and \$5.88 of cashback, the example of §9.7.

10.3 Tracking links

`POST /v1/cashback/tracking-links` takes `{intentId, rateId, destinationUrl}`. It requires a `SwayAgent` signed request from a key with the `cashback:link` scope (§13.3) and an `Idempotency-Key`. The service checks, in order:

1. The intent belongs to the calling agent key, and the rate exists.
2. The rate's shard equals the intent's shard, and the rate is available.
3. The destination uses `https:`, carries no credentials, port or fragment, and its hostname equals the rate's `merchantDomain` and appears in the allowlist.
4. The intent account is finalized, not settled, and records the same claim key as the service's row.

5. The intent's purchase deadline has not passed.
6. An intent has at most one link per rate. Repeating the pair with a different destination returns `IDEMPOTENCY_CONFLICT`.

```
link.expiresAt = min(intent.purchase_deadline, rate.expiresAt, now + AFI
```

The sub-ID is the intent's external ID, the random identifier the agent generated for `postIntent`. It is never a wallet, an agent key or an integrator ID. The affiliate adapter builds the network URL from the rate's advertiser ID, the sub-ID and the destination. Nothing is written into the user's browser. Attribution travels in the link the agent follows at checkout.

10.4 Signed postbacks

The affiliate gateway reports each network event to `POST /v1/affiliate/postbacks`. Two independent signatures protect it. The HTTP request carries a `SwayOracle` authorization from an allowlisted oracle key, with its own nonce and rate bucket (§13.3). The body carries an `Ed25519` attestation over the event itself, made by a configured oracle key:

```
"Sway affiliate postback v1" || 0x0A || canonical_json({ network, ...event fields exc
```

Event field	Meaning
<code>networkEventId</code>	The network's identifier for the commission event
<code>intentId</code> , <code>trackingLinkId</code>	The sub-ID and, when the intent has several links, the link
<code>orderHash</code>	SHA-256 of the order reference, 32 bytes as hex
<code>commissionAmount</code>	Commission in USDC base units
<code>purchaseAmount</code>	Optional purchase price in base units
<code>purchaseOccurredAt</code>	When the purchase happened
<code>status</code> , <code>occurredAt</code>	Network state (pending, validated, paid, reversed) and when the network entered it
<code>payoutReceivedAt</code>	When Sway received the commission

The service rejects an event unless all of these hold:

- `purchaseOccurredAt` is present (or already stored) and is not later than `occurredAt`.
- `occurredAt` is not in the future. `payoutReceivedAt` is not in the future and not before the purchase. A paid event carries `payoutReceivedAt`, and pending or validated events do not.

- For a paid event, the user share computed in §10.8 is positive and at most 50 USDC.
- The event resolves to exactly one tracking link of the intent on the configured network, and `purchaseOccurredAt` falls in `[link.createdAt, link.expiresAt)`.

The two timestamps have separate jobs. `purchaseOccurredAt` is when the user bought, and it is immutable for the event. `occurredAt` is when the network moved the event to its current state. A commission paid weeks after the purchase therefore keeps its purchase inside the intent's purchase window, and the settlement grace (§10.6) admits the late payment.

The first report of an event fixes its terms: tracking link, order hash, commission, purchase amount and purchase time. A later report with different terms returns `IDEMPOTENCY_CONFLICT`. State moves forward only. `pending` may become anything. `validated` may become `validated`, `paid` or `reversed`. `paid` may become `paid` or `reversed`. A batched event may only become `reversed`. `reversed` and `rejected` are final. Any other move returns `REPLAY_DETECTED`. Each event is processed under a PostgreSQL transaction lock keyed `affiliate-event:<network>:<networkEventId>`, the same lock the batch worker takes before it signs a deposit (§10.7). The service stores the signed payload, the signature and its SHA-256 for audit, and fixes `validatedAt` and `commissionReceivedAt` at their first occurrence.

10.5 Oracle batches and per-event outcomes

The oracle settles paid commissions with `POST /v1/oracle/batches`, body `{eventIds}` with 1–32 unique event IDs. Admission runs in one transaction:

- The finalized `OracleAuthority` account must name the configured oracle key and the configured network return days, and the key must be allowlisted. Otherwise the call returns `CONFIGURATION_REQUIRED`.
- Every event must be paid, have `commissionReceivedAt`, and belong to no batch (`FORBIDDEN` otherwise).
- No two events may share an intent, and no other cashback conversion may exist for those intents. One intent settles at most one cashback conversion (`REPLAY_DETECTED`).
- The events are reserved by writing the batch ID onto them, and a `cashback_batch` job is queued.

Every event has a global on-chain identity that does not depend on the intent:

```
event_hash = SHA-256( "sway/affiliate-event/v1" || 0x00 || canonical_json(
```

The worker processes each event of the batch in order:

1. It rejects an event whose stored attestation, order hash or purchase time is missing (`INVALID_ATTESTATION`).
2. It computes the split of §10.8 and enforces the cap (`CAP_EXCEEDED`).

3. It reads the intent's cashback conversion account. If an earlier signed deposit exists, it drives that attempt to finality before deciding anything else.
4. With no deposit on chain, it checks the finalized intent: it must exist (`INTENT_CLOSED`), must not be settled (`INTENT_SETTLED`), the purchase must precede `purchase_deadline` (`PURCHASE_OUTSIDE_WINDOW`), and chain time must not exceed `settlement_deadline` (`SETTLEMENT_EXPIRED`).
5. Under the event lock, it allocates the conversion row (tier T3, slot 255).
6. It submits `oracle_post_conversion` through the relay outbox (§13.5), with the oracle co-signing. Just before the signed attempt is persisted, it re-checks under the event lock that the event has not been reversed.
7. After finality, it compares every field of the conversion account with the expected values: tier, both hashes, purchase amount, integrator payout wallet, rebate and both fees (`CHAIN_BINDING_MISMATCH` on any difference). The conversion becomes escrowed with its `unlockAt`, and the event becomes batched.

Outcome	Meaning	Event afterwards
posted	The deposit finalized and matched every expected field	batched
pending	A signed deposit or a chain read is awaiting finality	unchanged; the job runs again
failed	A terminal check failed, with its failure code	rejected, unless already reversed
released	The deposit attempt ended without landing, and no signed attempt remains pending (<code>DEPOSIT_REJECTED</code>)	leaves the batch and may join another
cancelled	The event was reversed before any deposit was signed	allocation withdrawn

The batch stays running while any outcome is pending. It ends failed if any outcome is failed or released, and succeeded otherwise. `GET /v1/oracle/batches/:id` reports the original event IDs, the per-event outcomes and the status. One bad event never blocks the others.

10.6 T3 escrow on chain

`oracle_post_conversion(event_hash, purchase_amount, commission, purchase_timestamp)` needs two signers: the fee payer and the oracle key recorded in `OracleAuthority` (`has_one = oracle`). It creates the `Conversion` at seeds `["conversion", intent, [255]]` and the `OracleEvent` at `["oracle_event", event_hash]`. The source must be a classic SPL token account of the configured USDC mint whose authority is the oracle. The program requires:

- `event_hash` is not zero, `commission > 0`, and the intent is not settled (`status < 3`).
- The three cashback shares sum to 10,000 basis points, and the user's rebate is positive and at most `REBATE_CAP` (50,000,000 base units).

- `intent.created_at ≤ purchase_timestamp < intent.purchase_deadline` and `purchase_timestamp ≤ now`.
- `now ≤ intent.settlement_deadline`, where the settlement deadline is `purchase_deadline + Config.settlement_grace_secs`, fixed when the intent was posted.

It then transfers the entire commission from the oracle's source into the claims vault and writes the conversion:

Conversion field	T3 value
slot, proof_tier, status	255, 3, 1 (escrowed at creation)
campaign	the <code>OracleAuthority</code> address, in place of a campaign
order_hash, receipt_hash	both event_hash
rebate, sway_fee, integrator_fee	the split of \$10.8
integrator	copied from <code>Intent.integrator</code>
signer_epoch	<code>OracleAuthority.epoch</code> at posting
snapshot_hash	zero; no sealed result is involved
unlock_at	<code>now + network_return_days × 86400</code>

`OracleEvent` stores the conversion key, the source token account and the full commission, and `Intent.open_conversions` grows by one. Two seeds close every replay route. The conversion seed ends in the fixed slot 255, so one intent holds at most one cashback conversion. The event seed is the global event_hash, so one network event funds at most one conversion across all intents, before or after an oracle rotation. The database separately keeps (`network`, `networkEventId`) unique permanently.

`OracleAuthority` is a singleton at seeds [`"oracle_authority"`]. The Config admin creates it with `initialize_oracle` and replaces the oracle key or the return days with `rotate_oracle`, which increments epoch. The service also refuses to start a batch unless the finalized authority matches its own configuration, and refuses to start at all if `SETTLEMENT_GRACE_SECONDS` is shorter than `AFFILIATE_NETWORK_PAYOUT_LAG_SECONDS`, so that the grace covers the network's payout delay.

10.7 Reversals and refunds

A reversed postback queues a `cashback_refund` job. The event lock orders the reversal against the deposit:

- If the reversal commits before any deposit was signed, the worker cancels the allocation. The conversion row becomes rejected with no account address, and no money moves.
- If a signed deposit exists, the worker first drives it to finality and then refunds it.

`oracle_refund_conversion` takes the current oracle as signer and requires a T3 conversion at slot 255 in status 1 with `now < unlock_at`. The amount returned, `rebate + sway_fee + integrator_fee`, must equal `OracleEvent.commission`, and the destination must be the source recorded in `OracleEvent`. Every escrowed base unit therefore returns to the account that deposited it, even after the oracle key has rotated. The conversion moves to status 2 and the intent's open count drops by one. A reversal reported at or after `unlock_at` is recorded with failure code `REFUND_WINDOW_CLOSED`, and the escrow stays claimable. A reconciler re-arms refund work for every reversed event whose conversion is still pending or escrowed.

10.8 Claim and the 70/10/20 split

In a brand-direct conversion the brand pays the rebate plus 15% on top (§9.1). In a cashback conversion the commission is the whole pool, and the fees come out of it. The program floors both fees and gives the remainder to the user, so every base unit of the commission is accounted for:

```
sway_fee      = floor(commission × 2000 / 10000)
integrator_fee = floor(commission × 1000 / 10000)
rebate        = commission - sway_fee - integrator_fee
```

Commission (base units)	Sway (20%)	Integrator (10%)	User (remainder)
8,400,000 (\$8.40, the example of §10.2)	1,680,000	840,000	5,880,000
1,234,567 (for example)	246,913	123,456	864,198
71,428,571 (largest admissible)	14,285,714	7,142,857	50,000,000

The second row shows the rounding rule: an exact 70% would be 864,196.9, and the user receives 864,198. The third row is the cap: at 71,428,572 base units the remainder would be 50,000,001, above `REBATE_CAP`, so both the API and the program reject any larger commission.

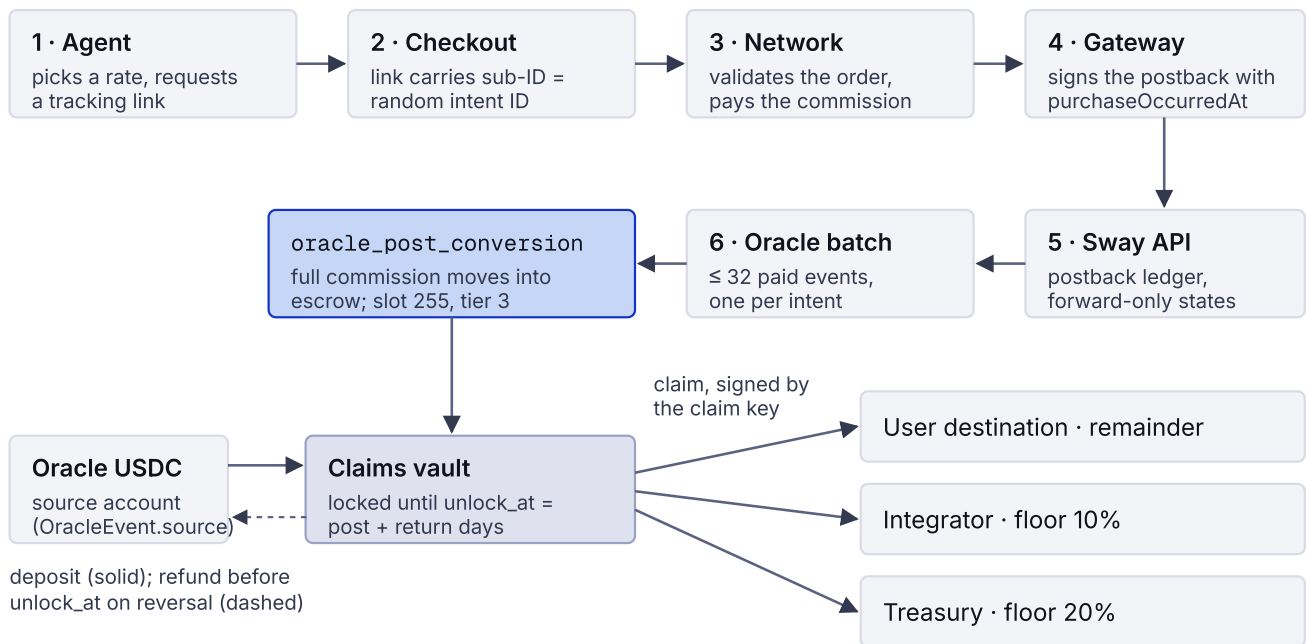


Figure 10.1 — The T3 path. Steps 1–6 attribute and attest the commission off chain; the program escrows the full commission and pays it out in one atomic claim, or returns it to its source on a reversal inside the lock.

Claiming a T3 conversion is identical to claiming a T2 one (§8.5). After `unlock_at`, the claim key signs the 128-byte `SWAYCLM1` message naming the conversion, a destination wallet, an expiry and a nonce. The relayer submits `claim`, which pays `rebate` to the destination, `sway_fee` to the treasury and `integrator_fee` to a token account owned by `Intent.integrator`, all in one instruction. When the intent's last open conversion is claimed, the intent becomes settled. After the settlement deadline, `close_oracle_conversion` returns the rent of a refunded or claimed T3 conversion and its `OracleEvent` to the original payer.

In REST responses a T3 conversion shows `campaignId` and `slot` as null, `totalCampaignDebit` as 0 and `refundAllowed` as false. A postback without a purchase price is stored on chain as `purchase_amount = 0`, and the API reports `purchaseAmountKnown: false` rather than presenting a free order.

11. Merchant bonds and disputes

A T2 conversion rests on the merchant's signed receipt (§7). Two mechanisms stand behind that signature. Before a merchant can sell through Sway it posts a USDC bond into a token account the program controls. If a user's purchase goes unrecorded, is recorded wrongly or is refunded without cause, the holder of the intent's claim key can open a dispute, and an admin resolution can pay compensation out of that bond. Only two instructions can debit a bond: `release_bond`, which returns it to the merchant once every liability has expired, and `slash_bond`, which pays a dispute award under a unique case hash.

11.1 Registration and receipt signers

A merchant signs in with its wallet (§13.3) and calls `POST /v1/merchant` with a display name, its receipt signer keys and its return window in days (0–65,535). The service returns a prepared `register_merchant` transaction. The wallet signs those exact bytes, and the service submits them through `POST /v1/transactions/:id/submit`. The instruction takes `receipt_signer` (any non-default Ed25519 key) and `return_window_days`, and creates three accounts in one transaction:

Account	Seeds	Initial contents
Merchant	<code>["merchant", wallet]</code>	wallet, receipt signer, bond address, return window, <code>signer_epoch = 0</code> , <code>status = 0</code> , <code>pending_count = 0</code> , <code>liability_until = 0</code> , and <code>brand_hash</code> , the identifier hash of the wallet address (§12.2)
Merchant bond (SPL token account)	<code>["merchant_bond", merchant]</code>	USDC mint, token authority = the Merchant PDA
ReceiptSignerEpoch 0	<code>["signer_epoch", merchant, 0u32 LE]</code>	the signer, <code>valid_from = now</code> , <code>valid_until = i64::MAX</code>

Because the bond's token authority is the Merchant PDA, no wallet can move bond funds directly. Only the program can, and only by signing with the seeds `["merchant", wallet, bump]` inside `release_bond` or `slash_bond`.

The return window is a public commitment. Every campaign the merchant creates must lock its conversions for at least that many days (`lock_days ≥ return_window_days`, error `LockTooShort`), and the campaign snapshots the value at creation. A merchant may also register a P-256 key, a 65-byte uncompressed SEC1 point. It stays in the service as dispute-evidence metadata, because AP2 checkout receipts are ES256-signed [S24]. The program holds no P-256 key.

The merchant replaces its receipt key with `rotate_receipt_signer`, prepared from `POST /v1/merchant/receipt-signers` and signed by the merchant wallet. Rotation closes the current signer epoch and opens the next one at the same instant (§7.5). Rotation matters to liability because it does not cancel anything: a receipt is judged against the epoch that was valid at its purchase timestamp,

so sales made under the old key still record after the rotation, while refund notices must come from the current key (§8.3).

11.2 Posting the bond and activation

`post_bond(amount)` is signed by the merchant wallet and moves `amount > 0` USDC from a token account the wallet owns into the bond account. It can be repeated to top up. The service prepares it from `POST /v1/merchant/bond`.

`activate_merchant` is signed by the Config admin. The admin lists pending merchants with `GET /v1/admin/merchants/pending` and calls `POST /v1/admin/merchants/:id/activate`. Both require a SwayAdmin signed request (§13.3) from the key that is also the on-chain Config admin, and the second returns a prepared transaction for the admin wallet to sign. The instruction reads the bond's token balance and requires `bond.amount ≥ Config.minimum_bond` (Underfunded otherwise), then sets `status = 1`. The minimum is fixed at `initialize`, and the admin can change it with `update_config`, which only accepts a positive value. It is a deployment parameter, listed under `requiredConfiguration` as `minimumMerchantBondUsdc` (§3.5). The service mirrors it as `MINIMUM_MERCHANT_BOND_USDC` and refuses to prepare a campaign unless the merchant is active and its bond meets the minimum.

11.3 What the merchant status gates

Instruction	0 pending	1 active	2 suspended
<code>create_campaign</code>	no	yes	no
<code>submit_offer</code>	no	yes	no
<code>ingest_offer_callback</code> accepting an offer	no	yes	no
<code>fund_campaign</code> , <code>withdraw_campaign</code>	yes	yes	yes
<code>record_conversion</code> , <code>verify_conversion_callback</code> , <code>refund_conversion</code> , <code>claim</code>	yes	yes	yes
<code>release_bond</code>	no	no	yes, once liabilities expire
<code>slash_bond</code>	yes	yes	yes

Matching respects suspension too. When the service captures an intent's effective book, it marks a slot active only if the merchant is active and the campaign vault holds at least the vault floor (§5.2.5). A suspended merchant's offers therefore stop appearing in new top-three results. Settlement does not depend on the status: receipts are checked against the signer epoch that was valid at the purchase timestamp, so conversions already owed by a suspended merchant still record, verify, refund and claim normally.

11.4 Liability windows

Two fields of Merchant decide how long the bond must stay in place.

- `pending_count` is the number of recorded receipts still awaiting verification. `record_conversion` and `requeue_conversion` add one. `verify_conversion_callback`, with either outcome, and `cancel_pending_conversion` subtract one.
- `liability_until` is a timestamp that only moves forward:

```
on verify_conversion_callback (valid): liability_until = max(liability.  
on suspend_merchant: liability_until = max(liability.
```

`release_bond(amount)` is signed by the admin and requires all of the following, failing with `BondLocked` otherwise:

1. The merchant is suspended (`status == 2`).
2. `pending_count == 0`: no receipt is waiting for verification.
3. `now ≥ liability_until`: every escrowed conversion has passed its unlock time plus the dispute window, and the dispute window has also elapsed since suspension.
4. $0 < \text{amount} \leq \text{bond balance}$, and the destination token account is owned by `Merchant.wallet`.

For example, a conversion verified at time T on a campaign with `lock_days = 30` unlocks at $T + 2,592,000$, and the bond cannot be released before $T + 2,592,000 + \text{dispute_seconds}$. Every verified sale therefore holds the bond in place until its own dispute window has closed.

The operator suspends merchants and releases bonds from the operator page `/admin`, which lists active and suspended merchants with `GET /v1/admin/merchants?status=active|suspended`. **Suspend** calls `POST /v1/admin/merchants/:id/suspend`, and **Release bond** calls `POST /v1/admin/merchants/:id/bond/release`. Both are SwayAdmin signed requests from the deployed Config admin, and both return a prepared transaction that the admin wallet signs as fee payer and admin, then sends through `POST /v1/transactions/:id/submit` or `/report` (§13.5). A release takes no amount and no destination from the request. The service reads the finalized merchant and computes both: the whole bond balance, paid to the USDC associated token account of `Merchant.wallet`, which the same transaction creates if it does not exist. While the program would refuse the release, because the merchant is not suspended, a receipt is pending or `liability_until` has not passed, the service answers `BOND_LOCKED` instead of preparing it.

11.5 Opening a dispute

The holder of an intent's claim key, in practice the user's agent device, opens a case with `POST /v1/disputes`:

Field	Content
intentId	The intent's external ID
targetMerchantId, targetCampaignId, targetSlot	The brand, campaign and book slot the purchase went to
conversionId	The T2 conversion in question, or null when none was recorded
reason	MISSING_CONVERSION, INCORRECT_AMOUNT or UNEXPECTED_REFUND
evidenceHash	SHA-256 of the evidence, 64 hex characters
encryptedEvidenceReference	An opaque: handle to encrypted evidence, at most 512 characters
payoutAddress	The wallet that will receive any compensation
authorization	A claim-key signature over a one-use open_dispute challenge

The authorization follows the challenge scheme of §13.3. The challenge's request hash is the SHA-256 of the canonical JSON of every field above except authorization, and its resource list is [intentId]. The payout address is part of the signed body, so it is bound when the case opens and no later party, including the admin, can redirect the award. The service then checks:

1. The intent exists and the authorization key is the intent's claim key.
2. The signature is valid. The challenge is consumed in the same database transaction that creates the case.
3. The merchant and campaign exist, the campaign belongs to the merchant, a slot is named, and the campaign's finalized account sits under that merchant's PDA.
4. With a conversionId: the conversion belongs to this intent, merchant, campaign and slot, is tier T2, and its finalized account carries the same intent, campaign, slot and receipt hash.
5. Without one: the reason must be MISSING_CONVERSION, the slot must have held this campaign as an active slot in the intent's captured book, the finalized intent (if still open) must map that slot to the same campaign account, and the agent must already have filed a purchase report (POST /v1/purchases) with the same intent, slot and evidence hash.
6. The window of §11.6 is still open (DISPUTE_WINDOW_CLOSED otherwise).
7. No other case exists for the same intent and slot, whatever its outcome (REPLAY_DETECTED). The service stores the SHA-256 of the authorization as a unique column, so one signed authorization opens at most one case.

The evidence itself never reaches Sway. The agent keeps the checkout receipt, for example an AP2 checkout receipt, on its device and submits only its digest and an opaque handle. The case can be read back with POST /v1/disputes/:id/query, authorized by a claim_read challenge bound to [disputeId].

11.6 The dispute window anchor

The window is measured from a time taken from finalized chain accounts, never from a timestamp the user supplies:

```
anchor = conversion.unlock_at                                if the conversion is verified
        = conversion.created_at + lock_days × 86400          if it is recorded as a verified conversion
        = intent.purchase_deadline + lock_days × 86400      for MISSING_CONVERSIONS

open is accepted while chain_now ≤ anchor + DISPUTE_WINDOW_SECONDS
```

For example, take a missing conversion on a campaign with `lock_days = 30`, an intent whose purchase deadline is T , and a dispute window of 604,800 seconds (seven days). The anchor is $T + 2,592,000$, and the case can be opened until chain time $T + 3,196,800$, 37 days after the purchase deadline. A verified conversion on the same campaign is anchored at its own `unlock_at` instead, which falls 30 days after the verification callback.

`lock_days` comes from the finalized campaign. For a missing conversion, the purchase deadline comes from the finalized intent, or, if the intent account has already been closed, from the service's mirror of that deadline, which is written only from finalized intent state. Filing a later purchase report cannot move the anchor. The service applies `DISPUTE_WINDOW_SECONDS` and the program applies `Config.dispute_seconds`. Both come from the deployment's single `disputeWindowSeconds` policy (§3.5), so the API stops accepting cases for a verified conversion at the same moment that conversion stops holding the bond.

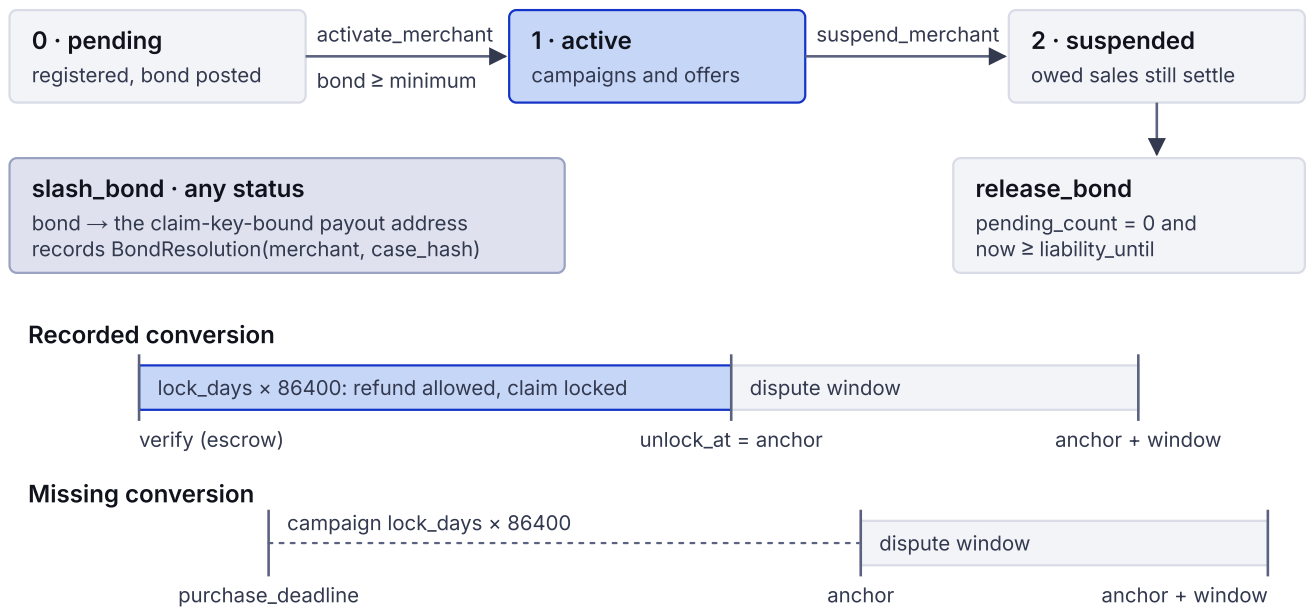


Figure 11.1 — Top: merchant status and the two instructions that can debit a bond. Bottom: the dispute window for a recorded conversion and for a missing one. The verified conversion also raises `liability_until` to `unlock_at + dispute_seconds`, the end of its window.

11.7 Resolution and bond compensation

The operator adjudicates cases on the operator page `/admin`. `GET /v1/admin/disputes` lists the open and under-review cases, each with its brand, opening time, evidence digest and bound compensation address. The decision is sent as `POST /v1/admin/disputes/:id/resolve`. The request must carry a SwayAdmin signed authorization from the Config admin key (§13.3), and the body is `{outcome: "user" | "merchant" | "reject", bondDebit, resolutionHash}`. `bondDebit` is in USDC base units, and `resolutionHash` is a 64-hex-character digest that identifies the decision; the operator page sets it to the SHA-256 of the operator's resolution note.

Outcome	Rules	Resulting status
user	Requires the configured compensation policy <code>direct-bond-payout</code> , a bound payout address, and $0 < \text{bondDebit} \leq 50,000,000$ base units (the per-conversion cap)	<code>under_review</code> , then <code>resolved_user</code> once the debit finalizes
merchant	<code>bondDebit</code> must be 0	<code>resolved_merchant</code> , immediately
reject	<code>bondDebit</code> must be 0	<code>rejected</code> , immediately

Resolution runs under a per-case session lock (§13.6) and an `admin:<publicKey>` idempotency scope. A case in a final state cannot be reopened (`IDEMPOTENCY_CONFLICT`). `merchant` and `reject` are recorded by the signed request alone. A user outcome is accepted only from the deployed Config admin (`FORBIDDEN` otherwise). It moves the case to `under_review` and turns the decision into a bond debit with a deterministic case hash:

```
case_hash = SHA-256( canonical_json({ amount, disputeId, evidenceHash, i
```

The response carries the case and a prepared `slash_bond` transaction in its `transaction` field. The admin wallet is its fee payer and its only signer: the operator page has it sign the transaction and sends it through `POST /v1/transactions/:id/submit` or `/report` (§13.5). The transaction first creates the USDC associated token account of the payout address if it does not exist, then calls `slash_bond(case_hash, amount)`. The program checks that the signer is the Config admin, that the bond belongs to the merchant, that `case_hash` is non-zero and that $0 < \text{amount} \leq \text{bond balance}$. It then transfers the amount from the bond to the destination token account and creates a `BondResolution` at seeds `["bond_resolution", merchant, case_hash]`, recording the merchant, case hash, amount, destination token account and `resolved_at`. Because the account is created with `init`, a second debit under the same case hash fails, and the same decision cannot pay twice.

The case becomes `resolved_user` only after the reconciler reads the finalized `BondResolution` and confirms that its merchant, case hash, amount and destination all equal the terms the service derived. The reconciler then records `bondDrawn` and takes `resolvedAt` from the chain's `resolved_at`. It also refreshes the merchant's bond balance and status mirrors. If the debit transaction fails or expires, the admin sends the request again with unchanged terms and receives a fresh transaction to sign; a request repeated while the earlier transaction is still prepared or submitted returns that transaction. Any request that changes the terms of a case under review is refused. A dispute is paid from the bond and leaves the conversion account it refers to unchanged: an escrowed conversion still follows its own refund and claim rules (§8).

Dispute status	Meaning	Entered through
<code>open</code>	The case is recorded and awaits adjudication	<code>openDispute</code>
<code>under_review</code>	A user outcome is authorized and its bond debit is in flight	<code>resolveDispute</code> with outcome <code>user</code>
<code>resolved_user</code>	The finalized <code>BondResolution</code> matches the decided terms; compensation has been paid	the dispute reconciler
<code>resolved_merchant</code>	Decided for the merchant; nothing is debited	<code>resolveDispute</code> with outcome <code>merchant</code>
<code>rejected</code>	Dismissed; nothing is debited	<code>resolveDispute</code> with outcome <code>reject</code>

A case read returns its status, reason, evidence hash, bound payout address, resolution hash, the amount actually drawn from the bond (`bondDebit`, zero until the debit finalizes), and the open and resolution times. While a user debit is in flight, a failed or expired transaction appears as a retryable failure on the case, which tells the admin that the same terms can be signed again.

12. The on-chain program

`sway_exchange` is a single Anchor program that holds every Sway balance and enforces every money rule. The deployed binary is built with the `arcium-financial` feature. That build takes its own program ID and the ID of the Arcium coordinator program that delivers its computation callbacks from the deployment's identity (`SWAY_FINANCIAL_PROGRAM_ID`, `SWAY_ARCIUM_COORDINATOR_PROGRAM_ID`) and compiles both in. It settles in plain SPL USDC through the classic SPL Token program, with six decimals and no Token-2022 extension. This chapter is the reference for its accounts, token accounts, instructions and invariants. The signed message layouts it verifies are described in §7 and §8 and listed byte by byte in §17.

12.1 Compiled constants

The protocol's fixed numbers are compiled into the binary. A generator writes the ten protocol constants from `PROTOCOL_PARAMETERS` into `generated_protocol.rs`, so the program, the API and the site read the same values (§3.4). The program adds three 8-byte domain tags for its signed messages, `SWAYRCP1`, `SWAYREF1` and `SWAYCLM1`, and a build script compiles in `PINNED_USDC_MINT` (§12.7). Values that differ between deployments, such as the vault floor, the minimum bond and the windows, live in `Config` instead (§12.3).

12.2 Account map

Every program account begins with an 8-byte Anchor discriminator, the first eight bytes of SHA-256("account:<Name>"). The sizes below include it. All of these accounts are owned by `sway_exchange`. The four token accounts in §12.4 are owned by the SPL Token program, with a program PDA as their token authority.

Account	PDA seeds	Bytes	Created by	Closed by
Config	"config"	266	initialize	—
Merchant	"merchant", wallet	132	register_merchant	—
ReceiptSignerEpoch	"signer_epoch", merchant, epoch u32 LE	93	register_merchant, rotate_receipt_signer	—
Campaign	"campaign", merchant, id16	157	create_campaign	—
OfferBook	"book", SHA-256(shard name)	10,120	create_production_book	—
OfferSubmission	"offer", campaign, id16	444	submit_offer	—
Intent	"intent", id16	1,309	post_intent	close_intent
IntentBook	"intent_book", intent	3,672	post_intent	close_intent
InputReplay	"input", x25519 sender, nonce16	9	post_intent, submit_offer	close_intent (intent inputs only)
Conversion	"conversion", intent, slot u8	296	record_conversion, oracle_post_conversion (slot 255)	close_conversion, close_oracle_conversion
OrderGuard	"order", merchant, order_hash	41	record_conversion	close_conversion
OracleAuthority	"oracle_authority"	47	initialize_oracle	—
OracleEvent	"oracle_event", event_hash	81	oracle_post_conversion	close_oracle_conversion
BondResolution	"bond_resolution", merchant, case_hash	121	slash_bond	—
BuybackState	"buyback", config	145	buyback_begin	buyback_finish

Every close sends the rent to the payer recorded inside the account (`Intent.payer`, `Conversion.payer` or `BuybackState.keeper`). The caller cannot name another recipient.

Four of these sizes belong to the deployed build. It appends an 8-byte identifier hash to `Merchant`, `Campaign` and `OfferBook`, and a sealed copy of each slot's offer to `IntentBook` (§12.3). An identifier hash is the first eight bytes, read little-endian, of `SHA-256("sway/" || domain || "/v1" || 0x00 || value)`, the same hash the encrypted intent and offer carry (§4.2.1). The program computes it itself, so these identities are derived on chain rather than supplied by a caller.

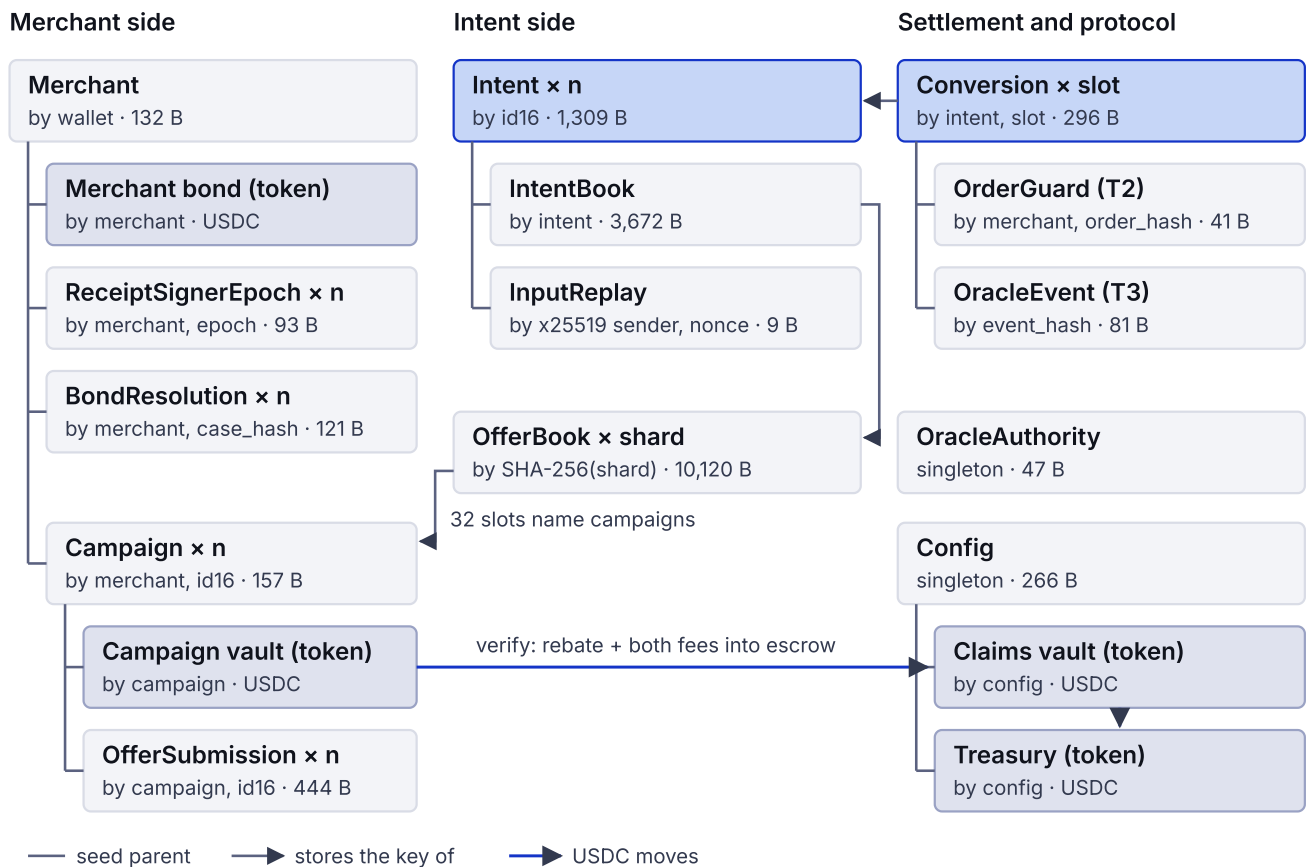


Figure 12.1 — Accounts grouped by owner. Spines join each PDA to the account in its seeds. A Conversion's campaign field names a Campaign for T2 or OracleAuthority for T3; its token flows are listed in §12.4.

12.3 Account layouts

Offsets are byte offsets into the account data, discriminator included. Integers are little-endian. Pubkey and hash fields are 32 bytes.

Config (266 bytes). bump u8 @8 · admin @9 · mint @41 · mxe @73 (the MXE callback authority) · claims_vault @105 · treasury @137 · vault_floor u64 @169 · minimum_bond u64 @177 · dispute_seconds u32 @185 · sway_bps u16 @189 · integrator_bps u16 @191 · rounding_policy u8 @193 · purchase_window_secs u32 @194 · settlement_grace_secs u32 @198 · sway_mint @202 (the SWAY mint the buyback burns) · buyback_authority @234 (the key that signs the buyback).

Merchant (132). bump @8 · wallet @9 · receipt_signer @41 · bond @73 · return_window_days u16 @105 · signer_epoch u32 @107 · status u8 @111 (0 pending, 1 active, 2 suspended) · pending_count u32 @112 · liability_until i64 @116 · brand_hash u64 @124 (identifier hash of the wallet address in base58, domain brand).

ReceiptSignerEpoch (93). bump @8 · merchant @9 · signer @41 · epoch u32 @73 · valid_from i64 @77 · valid_until i64 @85 (i64::MAX while current).

Campaign (157). bump @8 · id [16] @9 · merchant @25 · vault @57 · per_conversion_cap u64 @89 · daily_cap u64 @97 · funded_net u64 @105 · spent_today u64 @113 · spent_total u64 @121 ·

spent_day i64 @129 (UTC day index, unix_time div 86400) · lock_days u16 @137 · return_window_days u16 @139 · pending_total u64 @141 · campaign_hash u64 @149 (identifier hash of the campaign UUID text, domain campaign).

OfferBook (10,120, zero-copy, repr(c)). bump @8 · padding [7] @9 · shard [32] @16 · revision u64 @48 · campaigns [32][32] @56 · encrypted [32][8][32] @1080 · nonces [32][16] @9272 · counts [32] @9784 · slot_versions [32] u64 @9816 · active_mask u32 @10072 · padding [4] @10076 · mxe_sender [32] @10080 · category_hash u64 @10112 (identifier hash of the shard name, domain category). The book is zero-copy, with explicit padding, so that it is read in place and never deserialized onto the SBF stack. Its 10,120 bytes stay under the 10 KiB limit on account growth within one instruction [S20].

OfferSubmission (444). bump @8 · id [16] @9 · campaign @25 · book @57 · payer @89 · slot u8 @121 · sender [32] @122 · encrypted [8][32] @154 · nonce [16] @410 · input_count u8 @426 · status u8 @427 (0 pending, 1 accepted, 2 rejected) · expires_at_slot u64 @428 · slot_version u64 @436.

Intent (1,309).

Offset	Field	Type	Offset	Field	Type
8	bump	u8	796	mxe_result	[9][32]
9	id	[16]	1084	user_result_nonce	[16]
25	shard	[32]	1100	mxe_result_nonce	[16]
57	agent	Pubkey	1116	payer	Pubkey
89	claim_key	Pubkey	1148	book	Pubkey
121	integrator	Pubkey	1180	book_revision	u64
153	input_x25519	[32]	1188	book_active_mask	u32
185	user_x25519	[32]	1192	input_commitment	[32]
217	encrypted	[8][32]	1224	snapshot_hash	[32]
473	nonce	[16]	1256	result_count	u8
489	input_count	u8	1257	open_conversions	u32
490	attempt	u8	1261	mxe_sender	[32]
491	status	u8	1293	purchase_deadline	i64
492	expires_at_slot	u64	1301	settlement_deadline	i64
500	created_at	i64			
508	user_result	[9][32]			

status is 0 queued, 1 matched, 2 fallback, 3 settled. integrator is the integrator's payout wallet, the owner that receives the integrator fee.

IntentBook (3,672, zero-copy). bump @8 · padding [7] @9 · intent [32] @16 · campaigns [32][32] @48 · active_mask u32 @1072 · padding [4] @1076 · encrypted [32][2][32] @1080 · nonces [32][16] @3128 · counts [32] @3640. At post_intent it copies the book's 32 campaign keys and mask together with each slot's two sealed offer elements, nonce and element count, so the intent keeps a fixed copy of the sealed book it was posted against. It is never modified afterwards.

Conversion (296). bump @8 · intent @9 · campaign @41 (the OracleAuthority address for T3) · integrator @73 · receipt_hash @105 · order_hash @137 · purchase_amount u64 @169 · rebate u64 @177 · sway_fee u64 @185 · integrator_fee u64 @193 · created_at i64 @201 · unlock_at i64 @209 · slot u8 @217 · proof_tier u8 @218 (2 or 3) · status u8 @219 · signer_epoch u32 @220 · snapshot_hash [32] @224 · payer @256 · settlement_deadline i64 @288. status is 0 reserved (verification pending), 1 escrowed, 2 refunded, 3 claimed, 4 rejected, 5 cancelled. Locked and claimable are not stored states; they are read from unlock_at against chain time.

Small accounts. OrderGuard (41): bump, conversion @9. InputReplay (9): bump only, since its existence is the guard. OracleAuthority (47): bump, oracle @9, network_return_days u16 @41, epoch u32 @43. OracleEvent (81): bump, conversion @9, source @41, commission u64 @73. BondResolution (121): bump, merchant @9, case_hash @41, amount u64 @73, destination @81, resolved_at i64 @113. BuybackState (145): bump, keeper @9, keeper_usdc @41, sway_mint @73, usdc_before u64 @105, supply_before u64 @113, usdc_in u64 @121, min_burn u64 @129, slot u64 @137. It lives only between buyback_begin and buyback_finish inside one transaction.

12.4 Token accounts and fund movement

Token account	Seeds	Token authority	Credited by	Debited by
Campaign vault	"campaign_vault", campaign	Campaign PDA	fund_campaign, refund_conversion	verify_conversion_callback, withdraw_campaign
Claims vault	"claims_vault", config	Config PDA	verify_conversion_callback, oracle_post_conversion	claim, refund_conversion, oracle_refund_conversion
Treasury	"treasury", config	Config PDA	claim (Sway fee)	buyback_begin only
Merchant bond	"merchant_bond", merchant	Merchant PDA	post_bond	release_bond, slash_bond

All four are created with token::mint = Config.mint, and every instruction that touches a token account re-checks its mint. Every debit is a CPI signed with the authority's own seeds: ["config", bump], ["campaign", merchant, id, bump] or ["merchant", wallet, bump]. No instruction accepts a caller-chosen authority. One claims vault serves all campaigns and all cashback conversions. Every

credit and every debit of it moves exactly one conversion's stored total, `rebate + sway_fee + integrator_fee`, so its balance always covers the sum of those totals over escrowed conversions.

`Campaign.funded_net` counts only deposits made through `fund_campaign`. USDC sent straight to a vault raises its balance but not its registered commitment, so it cannot lift a campaign past `CAMPAIGN_CAP`. A withdrawal must leave every pending reservation covered by the vault and by `funded_net` (the three conditions of §5.1.4).

12.5 Instructions

An instruction's data is the 8-byte discriminator `SHA-256("global:<name>")[0..8]` followed by its Borsh-encoded arguments. The program defines one Anchor event, `BuybackBurned { usdc_in, sway_burned }`, emitted by `buyback_finish`. Every other effect is a change of account state, and the services observe it by reading finalized accounts (§13.6). Error names are those of the program's `ExchangeError` enum (§17 C.2).

Administration and governance

Instruction	Signers	Checks	Effects
<code>initialize(mxe, vault_floor, minimum_bond, dispute_seconds, rounding_policy, purchase_window_secs, settlement_grace_secs, sway_mint, buyback_authority)</code>	admin (payer)	§12.7 bootstrap; mint is PINNED_USDC_MINT with 6 decimals; all values positive; rounding_policy == 1; mxe is the coordinator's financial_authority PDA; mxe and buyback_authority non-default; sway_mint is not the USDC mint	Creates Config, claims vault and treasury; copies the fee bps from the constants; stores the SWAY mint and the buyback authority
<code>handoff_production_admin(new_admin)</code>	current admin, still the upgrade authority	§12.7 bootstrap check again; new_admin non-default and different	Config.admin = new_admin, once, at the end of bootstrap
<code>rotate_admin</code>	admin and new admin	has_one = admin	Config.admin = new_admin
<code>update_config(mxe, vault_floor, minimum_bond, dispute_seconds, sway_mint, buyback_authority)</code>	admin	all amounts positive; mxe is the coordinator's financial_authority PDA; mxe and buyback_authority non-default; sway_mint is not the USDC mint	Overwrites those six fields only
<code>initialize_oracle(oracle, network_return_days) / rotate_oracle(...)</code>	admin (and payer)	oracle non-default	Creates or rewrites OracleAuthority; rotation increments epoch
<code>activate_merchant / suspend_merchant</code>	admin	bond ≥ minimum_bond for activation	status 1 or 2; suspension raises liability_until (§11.4)
<code>create_production_book(shard_text)</code>	payer, admin	name of 1–80 bytes	OfferBook at ["book", SHA-256(shard_text)] with its category_hash
<code>clear_book_slot(slot)</code>	admin	slot < 32	Zeroes the slot, clears its mask bit, increments slot_versions[slot] and revision
<code>release_bond(amount) / slash_bond(case_hash, amount)</code>	admin	§11.4, §11.7	Bond to merchant wallet / to dispute payout, with BondResolution

Treasury buyback

Instruction	Signers	Checks	Effects
buyback_begin(amount, min_burn)	Config.buyback_authority (keeper, payer)	Config.sway_mint set and not the USDC mint (ConfigurationRequired); amount > 0, min_burn > 0 (InvalidAmount); amount $\leq$ treasury balance (Underfunded); keeper USDC account owned by the keeper; through the instructions sysvar: top-level, not last, and the last instruction is buyback_finish with the same keeper, Config, BuybackState, keeper USDC account, SWAY mint and sysvar (BuybackIncomplete)	Creates BuybackState with the keeper's USDC balance, the SWAY supply, amount, min_burn and the slot; treasury to the keeper's USDC account
buyback_finish	Config.buyback_authority	the last top-level instruction; same slot as buyback_begin; keeper USDC balance $\leq$ the recorded balance; SWAY supply fell by at least min_burn (BuybackIncomplete)	Emits BuybackBurned { usdc_in, sway_burned }; closes BuybackState to the keeper

The two instructions bracket the swap and the burn in one transaction (§9.8). buyback_begin checks the closing instruction before it moves any USDC, and buyback_finish checks the result, so treasury USDC cannot stay with the keeper past the end of the transaction.

Merchants, campaigns and offers

Instruction	Signers	Checks	Effects
<code>register_merchant(receipt_signer, return_window_days)</code>	wallet	signer non-default	Merchant, bond account, epoch 0 (§11.1)
<code>rotate_receipt_signer(receipt_signer)</code>	wallet	previous epoch PDA is the current one	Bounds the old epoch, creates epoch $n + 1$
<code>post_bond(amount)</code>	wallet	$\text{amount} > 0$; source owned by wallet	Source to bond
<code>create_campaign(id, per_conversion_cap, daily_cap, lock_days)</code>	wallet	merchant active; $0 < \text{cap} \leq \text{REBATE_CAP}$; $0 < \text{daily_cap} \leq \text{CAMPAIGN_CAP}$; $\text{lock_days} \geq \text{return_window_days}$	Campaign and vault; $\text{spent_day} = \text{today}$
<code>fund_campaign(amount)</code>	wallet	$\text{amount} > 0$; $\text{funded_net} + \text{amount} \leq \text{CAMPAIGN_CAP}$	Source to vault; $\text{funded_net} += \text{amount}$
<code>withdraw_campaign(amount)</code>	wallet	the three conditions of §5.1.4; destination owned by wallet	Vault to destination; $\text{funded_net} -= \text{amount}$
<code>submit_offer(id, slot, sender, encrypted, nonce, input_count)</code>	wallet	$\text{slot} < 32$; $\text{input_count} == 2$ (the packed offer, §5.3); canonical x25519 sender; merchant active; slot empty or already this campaign's	OfferSubmission (status 0, current slot_version, expires_at_slot = slot + 180); InputReplay(sender, nonce)
<code>ingest_offer_callback(valid, encrypted, nonce, element_count, sender)</code>	MXE authority	submission pending and unexpired; if valid: merchant active, $1 \leq \text{element_count} \leq 8$, canonical sender equal to the book's MXE sender once any slot is active, slot_version unchanged, slot empty or same campaign	Invalid: submission status 2. Valid: writes the slot, sets its mask bit, increments slot_versions[slot] and revision, status 1

Intents and matching

Instruction	Signers	Checks	Effects
<code>post_intent(id, shard, encrypted, nonce, input_count, input_x25519, user_x25519, claim_key, integrator)</code>	payer, agent	<code>input_count == 8</code> (the packed intent, §4.2); claim key and integrator non-default; <code>input_x25519 == user_x25519</code> , canonical (InputKeyMismatch); with <code>deadline = u64 LE nonce[0..8]: now < deadline ≤ now + purchase_window_secs</code> ; book shard matches	Intent (attempt 1, status 0, <code>expires_at_slot = slot + 180</code> , <code>settlement_deadline = deadline + settlement_grace_secs</code> , book revision and mask), IntentBook copy of the 32 campaign keys, InputReplay, input_commitment
<code>retry_intent</code>	agent	<code>now < purchase_deadline</code> ; status 0; <code>slot ≥ expires_at_slot</code> (NotExpired)	Attempt 1 → 2 with a fresh 180-slot window; attempt 2 → status 2
<code>match_intent_callback(attempt, input_commitment, result_count, user_result, mxe_result, user_nonce, mxe_nonce, sender)</code>	MXE authority	<code>now < purchase_deadline</code> ; status 0, same attempt, unexpired (StaleCallback); commitment equal (SnapshotMismatch); <code>1 ≤ result_count ≤ 9</code> ; unused elements zero	Stores both sealed copies, status 1, snapshot_hash

The program computes both commitments that tie matching to settlement from its own stored bytes: `input_commitment` when the intent is posted (§4.5.4) and `snapshot_hash` from the sealed MXE copy when the callback lands (§6.8). It recomputes `snapshot_hash` again at `record_conversion` and at verification instead of accepting a caller's value. §17 A.11 lists both constructions.

Conversions and settlement

Instruction	Signers	Checks	Effects
<code>record_conversion(slot, receipt, rebate, snapshot)</code>	payer, MXE authority	$\text{slot} < 32$; intent matched; now $\leq \text{settlement_deadline}$; IntentBook maps the slot to this campaign; snapshot equals the stored and recomputed hash; receipt has domain, zero padding, this intent, slot and program, and the epoch of the ReceiptSignerEpoch account seeded from receipt bytes 89–92; Ed25519 precompile by that epoch's signer (§12.6); $\text{created_at} \leq \text{ts} < \text{purchase_deadline}$, $\text{ts} \leq \text{now}$, ts within the epoch; $0 < \text{rebate} \leq \text{per_conversion_cap}$, $\leq \text{REBATE_CAP}$	Fees floored from rebate; reserves the total against vault balance, funded_net and the daily cap; Conversion status 0; OrderGuard; open_conversions += 1; pending_count += 1
<code>verify_conversion_callback(receipt_hash, valid)</code>	MXE authority	status 0; hash equal; intent matched; snapshot unchanged	Releases the reservation. Invalid: status 4. Valid: vault $\geq \text{total}$, $\text{spent_total} + \text{total} \leq \text{funded_net}$, $\text{day spend} + \text{total} \leq \text{daily_cap}$; vault to claims vault; status 1; created_at = now; unlock_at = now + lock_days × 86400; raises liability_until
<code>refund_conversion(notice)</code>	fee payer; receipt signer via precompile	status 1, now < unlock_at; notice equals SWAYREF1 message for this conversion, receipt hash and current epoch, signed by the current receipt signer	Claims vault to campaign vault (full total); reverses spent_total and same-day spent_today; status 2; open_conversions -= 1
<code>cancel_pending_conversion</code>	admin or MXE authority	status 0	Releases the reservation; status 5
<code>requeue_conversion</code>	admin or MXE authority	now $\leq \text{settlement_deadline}$; status 4 or 5; intent matched; snapshot unchanged	Reserves the same stored total again; status 0

Instruction	Signers	Checks	Effects
<code>claim(authorization)</code>	fee payer; claim key via precompile	status 1 (AlreadyConsumed); now $\geq$ <code>unlock_at</code> (ClaimLocked); SWAYCLM1 message signed by <code>Intent.claim_key</code> , expiry $\geq$ now, destination wallet equals the destination token account's owner; integrator account owned by <code>Intent.integrator</code>	Rebate to destination, Sway fee to treasury, integrator fee to integrator, atomically; status 3; intent status 3 when no conversion stays open
<code>oracle_post_conversion /</code> <code>oracle_refund_conversion</code>	payer and oracle / oracle	\$10.6, \$10.7	T3 escrow at slot 255 / full return to source

The verification callback carries a boolean and a hash, not an amount. The rebate and both fees are fixed when the receipt is recorded, so a callback can approve or reject a conversion but cannot change what it pays. Reservations stay counted across UTC day changes until they are verified, rejected or cancelled.

Cleanup and utility

Instruction	Signers	Checks	Effects
<code>close_intent</code>	agent	<code>open_conversions == 0</code> ; now > <code>settlement_deadline</code>	Closes Intent, IntentBook, InputReplay to <code>Intent.payer</code>
<code>close_conversion</code>	any fee payer	tier 2, status 2–5, now > <code>settlement_deadline</code>	Closes Conversion and OrderGuard to <code>Conversion.payer</code>
<code>close_oracle_conversion</code>	any fee payer	tier 3, status 2–3, now > <code>settlement_deadline</code>	Closes Conversion and OracleEvent to <code>Conversion.payer</code>
<code>request_nonce(nonce16)</code>	wallet	—	None (§12.10)

A closed InputReplay cannot reopen an old input. The nonce inside it carries an expired purchase deadline, so `post_intent` rejects the same tuple with `PurchaseWindowClosed`. A conversion closes without needing its intent, because it stores its own payer and settlement deadline.

Who signs what

Every authority the program recognizes is pinned in an account, and each one can do only the instructions listed for it.

Key	Pinned in	Authorizes
Config admin	Config.admin, which initialize requires to be the upgrade authority	handoff_production_admin (once), rotate_admin (with the new admin), create_production_book, update_config, initialize_oracle, rotate_oracle, activate_merchant, suspend_merchant, clear_book_slot, release_bond, slash_bond, cancel_pending_conversion, requeue_conversion
Buyback authority	Config.buyback_authority, the relayer at initialize, changeable by update_config	buyback_begin, buyback_finish
MXE authority	Config.mxe, the ["financial_authority"] PDA of the Arcium coordinator program, which signs by CPI	ingest_offer_callback, match_intent_callback, record_conversion (as co-signer), verify_conversion_callback, cancel_pending_conversion, requeue_conversion
Oracle	OracleAuthority.oracle	oracle_post_conversion from its own USDC account; oracle_refund_conversion
Merchant wallet	the Merchant PDA seed	register_merchant, rotate_receipt_signer, post_bond, create_campaign, fund_campaign, withdraw_campaign, submit_offer
Receipt signer	ReceiptSignerEpoch.signer, Merchant.receipt_signer	receipts and refund notices, through the precompile only
Agent	Intent.agent, the relayer for REST intents	post_intent, retry_intent, close_intent
Claim key	Intent.claim_key	claim authorizations, through the precompile only
Integrator	Intent.integrator	nothing; it owns the account that receives the integrator fee
Fee payer	recorded as payer where rent must return	fees and rent for any instruction it joins

Two of these keys never sign a Solana transaction at all. The receipt signer and the claim key sign 128-byte messages that reach the program inside the Ed25519 precompile, so a merchant's checkout service and a user's device need no SOL and no transaction tooling.

12.6 Signature introspection

Three instructions depend on an Ed25519 signature, and each expects a different signer:

Instruction	Message	Expected signer
record_conversion	SWAYRCP1 receipt	ReceiptSignerEpoch.signer of the epoch the receipt names
refund_conversion	SWAYREF1 notice	Merchant.receipt_signer, the current key
claim	SWAYCLM1 authorization	Intent.claim_key

The runtime's Ed25519 precompile verifies the signature, and the transaction fails before the program runs if it is wrong. The shared function `verify_ed25519` then reads the instructions sysvar to establish which key and which 128 bytes the precompile verified. The precompile must sit immediately before the Sway instruction, carry exactly one signature, and take its signature, key and message from its own data (all three instruction indexes `0xFFFF`). Its key and message must equal the expected signer and the expected message byte for byte. §7.6 walks through the checks, and §17 A.5 gives the precompile header field by field.

12.7 How the program pins Config

Config lives at the singleton PDA `["config"]`, so `initialize` succeeds exactly once. Before creating it, `verify_bootstrap` proves that the signer is the program's upgrade authority:

- The program account is the declared ID, executable, owned by the upgradeable BPF loader, 36 bytes long, with state tag 2 (Program) pointing to the ProgramData PDA derived from `[program_id]` under the loader.
- The ProgramData account is owned by the loader, has state tag 3, an upgrade authority present (`data[12] == 1`), and that authority (`data[13..45]`) equals the signing admin.

No one but the deployer can therefore create the Config that every other instruction reads.

Bootstrap ends with `handoff_production_admin`, which repeats the same upgrade-authority check and moves `Config.admin` to the operating admin key in one step. It can run only while the signer is both the Config admin and the upgrade authority, and after it every admin change needs `rotate_admin` with both signatures.

The deployed build also fixes who may sign callbacks. `initialize` and `update_config` accept only one value for `Config.mxe`: the program address derived from `["financial_authority"]` under the Arcium coordinator program compiled into the binary. Only that program can sign for its PDA, so every instruction in §12.5 that requires the MXE authority as signer admits only callbacks that arrive through the coordinator.

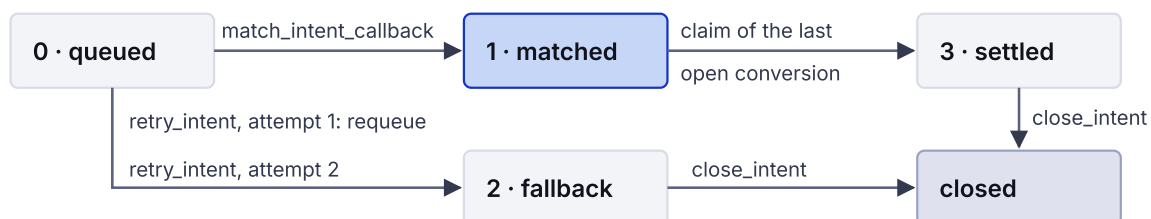
What the Config pins, and who can change it:

Field	Set at	Changeable by
mint	initialize, equal to the compiled PINNED_USDC_MINT with 6 decimals	Nothing; there is no setter
sway_bps, integrator_bps, rounding_policy	initialize, from the constants and policy 1 (floor to a base unit)	Nothing
purchase_window_secs, settlement_grace_secs	initialize	Nothing; each intent also snapshots its own deadlines
claims_vault, treasury	initialize, as PDAs	Nothing
admin	initialize, the upgrade authority	handoff_production_admin once, while the signer is still the upgrade authority; afterwards rotate_admin, signed by both the old and the new admin
mxe	initialize, the coordinator's financial_authority PDA	update_config (admin), and only to that same PDA
vault_floor, minimum_bond, dispute_seconds	initialize	update_config (admin), positive values only
sway_mint, buyback_authority	initialize, with the relayer as buyback authority	update_config (admin); the authority non-default, the SWAY mint never the USDC mint

A normal build reads the mint from the SWAY_PINNED_USDC_MINT environment variable at compile time and refuses to compile without it, so a deployed binary accepts exactly one mint. The services pin the same facts from the other side. Before preparing any money transaction, the chain adapter checks that the Config account is 266 bytes, owned by the program and carries the Config discriminator. It also checks that the mint and admin equal the configured USDC_MINT and ADMIN_MULTISIG_ADDRESS, that both windows equal the configured values, and that the ProgramData upgrade authority equals the configured PROGRAM_UPGRADE_AUTHORITY, the deployer key. Before a buyback, the worker also requires the finalized sway_mint and buyback_authority to equal SWAY_MINT and its own relayer key. A successful check is cached for CHAIN_CONFIG_CACHE_TTL_MILLISECONDS. A failed one is never cached.

12.8 State machines

Intent.status



Conversion.status

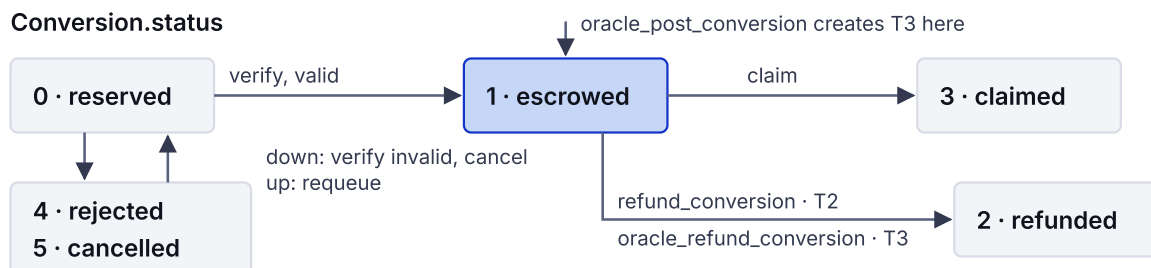


Figure 12.2 — Status transitions and the instruction behind each. T2 conversions enter at 0 through record_conversion. Terminal states close after the intent's settlement deadline: T2 states 2–5 with close_conversion, T3 states 2–3 with close_oracle_conversion. close_intent applies to any intent status once no conversion is open.

12.9 Instruction-to-flow mapping

Flow	Off-chain trigger (§13)	Instructions, in order	Transaction signers
Merchant onboarding (§11)	POST /v1/merchant, /v1/merchant/bond, /v1/admin/merchants/:id/activate	register_merchant → post_bond → activate_merchant	merchant wallet; Config admin
Campaign funding (§5)	POST /v1/campaigns, /fund, /withdraw	create_campaign → fund_campaign ⇌ withdraw_campaign	merchant wallet
Offer placement (§5.5)	POST /v1/offers, then the ingest job	submit_offer → ingest_offer_callback	merchant wallet; MXE authority
Intent and matching (§4, §6)	POST /v1/intents, then the post and match jobs	post_intent → match_intent_callback, or retry_intent	relayer as payer and agent; MXE authority
Receipt and verification (§7.7)	POST /v1/receipts, then the settlement job	coordinator verify_conversion → record_certified (after the Ed25519 precompile), which calls record_conversion and verify_conversion_callback (§7.12)	relayer and MXE authority
Refund (§8.3)	POST /v1/conversions/:id/refund	Ed25519 precompile → refund_conversion	relayer
Claim (§8.5)	POST /v1/claims	token-account creation if needed → Ed25519 precompile → claim	relayer
Cashback (§10)	POST /v1/oracle/batches; reversed postbacks	oracle_post_conversion; oracle_refund_conversion	relayer and oracle
Dispute award (§11.7)	POST /v1/admin/disputes/:id/resolve from the operator page	token-account creation → slash_bond	Config admin wallet, as fee payer and admin
Suspension and bond release (§11.4)	POST /v1/admin/merchants/:id/suspend, /bond/release from the operator page	suspend_merchant; token-account creation → release_bond	Config admin wallet, as fee payer and admin
Buyback and burn (§9.8)	the worker's buyback job	compute budget → buyback_begin → Jupiter route → SPL burn → buyback_finish; a surplus burn after it	relayer, as Config.buyback_authority
Pending recovery	admin or MXE authority	cancel_pending_conversion, requeue_conversion	admin or MXE authority

Flow	Off-chain trigger (§13)	Instructions, in order	Transaction signers
Rent recovery	reconcilers	close_intent; close_conversion or close_oracle_conversion	relayer
Governance	admin	initialize, handoff_production_admin, create_production_book (once per shard), rotate_admin, update_config, initialize_oracle, rotate_oracle, clear_book_slot	Config admin

For relayed REST intents, the relayer signs `post_intent` as both payer and agent, so the relayer is also the key that later signs `close_intent`. The user's own keys, the x25519 key and the claim key, never sign a transaction. The claim key signs the 128-byte claim message, which reaches the program through the precompile.

12.10 Transaction composition, size and lookup tables

Every transaction the services prepare has the same shape:

```
[ associated-token-account creation, when a payout account may not exist ]
[ Ed25519 precompile, for record, refund and claim ]      ← must sit immediately b
[ the Sway instruction ]
[ request_nonce(16 random bytes) ]
```

The buyback (§9.8) is the one transaction with a different shape. It is always a version-0 message, compiled against the lookup tables of its Jupiter route and the Sway table, opens with compute-budget instructions, ends with `buyback_finish` and carries no `request_nonce`, because `buyback_finish` must be its last instruction. A buyback that exceeds 1,232 bytes is never prepared.

`request_nonce` does nothing on chain. Two otherwise identical actions approved by the same wallet against the same blockhash would compile to identical bytes and an identical signature, and the runtime would reject the second as already processed. Sixteen random bytes make every prepared transaction distinct. When a wallet signs a prepared transaction, the service forwards it only if its instructions are exactly the prepared ones, in order. The wallet may add `ComputeBudget` instructions, which carry no accounts, and Lighthouse guard instructions after the prepared ones, but it may not add writable accounts or signers (§13.5, §15.5).

A Solana transaction is limited to 1,232 bytes [S18]. The table gives the serialized size of each relayed transaction, and of the admin wallet's bond debit, as a legacy message, computed from the account lists and argument sizes the chain adapter uses. Keys count the unique accounts including program IDs. Data counts the Sway instruction's bytes including the discriminator.

Transaction	Keys	Signatures	Data	Bytes
post_intent	8	1	457	855
match_intent_callback	5	2	682	1,043
submit_offer	9	1	330	760
ingest_offer_callback	8	2	314	774
record_conversion with precompile	15	2	177	1,112
verify_conversion_callback	11	2	41	599
refund_conversion with precompile	12	1	136	907
claim with precompile	12	1	136	907
claim with two token-account creations	17	1	136	1,087
oracle_post_conversion	12	2	64	656
slash_bond with token-account creation, admin wallet as payer	12	1	48	583
close_intent	6	1	8	338

The adapter prefers the legacy format. When a Sway address lookup table is configured (SOLANA_LOOKUP_TABLE_ADDRESS) and the legacy serialization fails or exceeds 1,100 bytes, it compiles a version-0 message against that table instead. The margin below 1,232 bytes leaves room for the compute-budget and guard instructions a wallet may append. A transaction that still exceeds 1,232 bytes is never prepared. In a version-0 message, an account loaded through the table costs a one-byte index instead of its 32-byte key. Accounts that never change from one transaction to the next are the ones to load this way: Config, the claims vault, the treasury, OracleAuthority, the USDC mint, the SPL Token and System programs, the instructions sysvar, and the static Arcium accounts that every queued computation references. Program IDs invoked at the top level stay in the static key list, as the format requires. Of the transactions above, record_conversion at 1,112 bytes is the one that crosses the 1,100-byte threshold.

13. Off-chain services

Sway's off-chain side is one TypeScript backend deployed as two services that share one PostgreSQL database. The API authenticates and validates requests, prepares transactions and records work. The worker holds the relayer key, runs durable jobs, signs and broadcasts transactions, queues Arcium computations and reconciles the database against finalized chain state. Neither service holds a user key or sees a plaintext intent or offer rule. The program's accounts are the record of money. The database records work, authorizations and mirrors of finalized accounts, and a mirror never authorizes a payout.

13.1 The API and the worker

Both services are built from the same code and differ only in their entry point (§16.1). Every route, body, parameter, query and response schema comes from the OPERATIONS table of the shared contract package, the same source that generates `openapi.json` and the MCP tool schemas.

	API	Worker
Keys held	the relayer's public key only	the relayer keypair (RELAYER_KEYPAIR), which is also <code>Config.buyback_authority</code>
Chain access	finalized reads; prepares unsigned transactions	signs, broadcasts and observes transactions
Arcium access	publishes the MXE encryption key	queues <code>ingest</code> , <code>match</code> and <code>verify</code> computations and reads their status
Concurrency	stateless between requests; shared state lives in PostgreSQL	exactly one, enforced by a PostgreSQL session lock on <code>sway:single-worker</code> ; a second worker refuses to start
Health	GET /health reports the database and configured capabilities	GET /health on its own port

The worker runs two schedulers that never wait on each other. The job scheduler fires every `WORKER_POLL_MILLISECONDS` and runs up to `WORKER_JOBS_PER_TICK` jobs. The reconciliation scheduler fires every `WORKER_RECONCILE_INTERVAL_MILLISECONDS` (§13.6). Arcium computation offsets are deterministic, for example the first eight bytes, read little-endian, of SHA-256 over a match job's identity `match:<intent>:<attempt>`. A repeated queue request therefore names the same computation, and a computation still pending surfaces as "work pending" rather than as a failure.

The worker is also the SWAY buyback keeper (§9.8). Each tick re-arms one buyback job per interval, and the job runs under the session lock `buyback:<programId>`. It first finishes any run still in flight. Otherwise it starts a new one when the last run is at least `BUYBACK_INTERVAL_SECONDS` old (default 3,600) and the finalized treasury holds at least `BUYBACK_MIN_USDC` (default 10000000, 10 USDC). A new run creates the keeper's USDC and SWAY token accounts if needed, quotes the whole treasury balance from USDC to SWAY on Jupiter with `BUYBACK_SLIPPAGE_BPS` (default 100), stores the

compiled transaction and relays it through the outbox. After the finalized BuybackBurned event matches the stored amounts, it burns any surplus SWAY. `SWAY_MINT` names the mint, and the keeper runs only while `SWAY_MINT` and the relayer key equal the finalized `Config.sway_mint` and `Config.buyback_authority`.

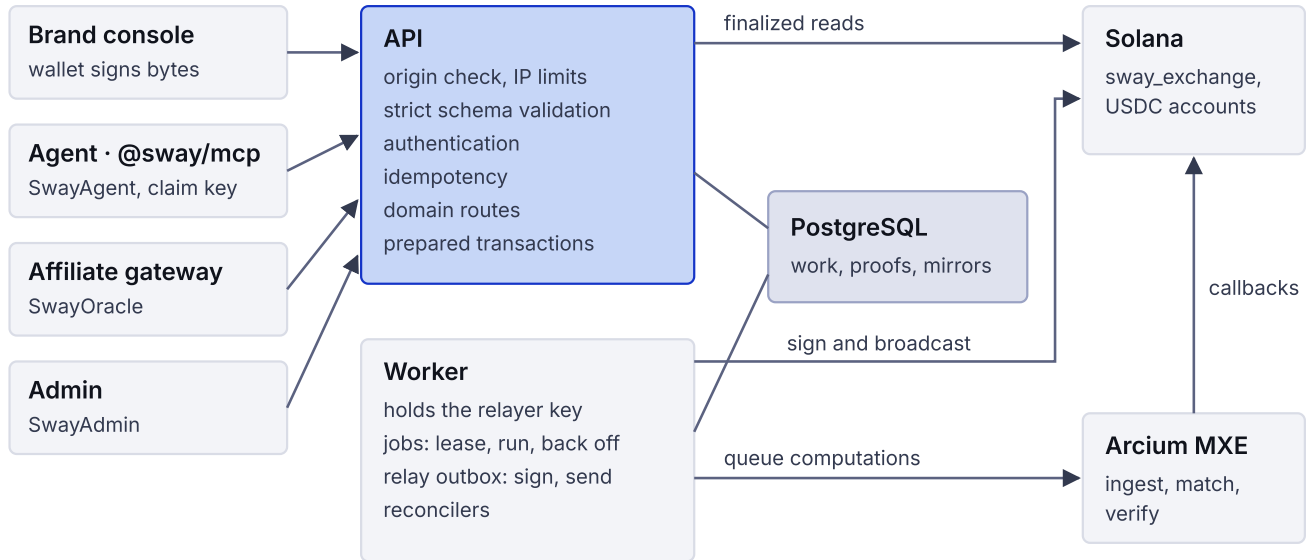


Figure 13.1 — Clients reach only the API. The worker is the only process that signs relayed transactions and queues computations. Both services coordinate through PostgreSQL leases and locks, and both read money state from finalized chain accounts.

13.2 Request pipeline

Each HTTP request passes the same stages in order. §15.9 gives the exact CORS, input and rate-limit rules.

1. **Origin.** A request whose `Origin` is not one of the exact origins in `CORS_ORIGINS` is refused with `403 ORIGIN_NOT_ALLOWED` before any other work.
2. **IP limits.** Authentication routes and the early-access form charge a salted per-IP bucket before the body is parsed against its schema, so malformed floods still count.
3. **Validation** against the operation's schema: at most 16,384 bytes, no type coercion, no unknown properties.
4. **Idempotency key**, required on every mutating operation (§13.4).
5. **Authentication** by the operation's role (§13.3), which also charges the per-key rate bucket of agents, the oracle and admins.
6. **Domain handler**, which returns `{data, meta: {mode, scenario, requestId}}` or `{error: {code, message, retryable, details?}, meta}`. The API error codes are listed in §17 C.1.

Rate buckets are rows of `RateLimitBucket`, keyed by the SHA-256 of the bucket name and updated by one atomic upsert, so every API replica shares them. A bucket's count is capped at its limit plus one, so a flood cannot grow the stored count without bound. IP buckets use HMAC-SHA256 under

IP_HASH_SALT, a secret of at least 32 characters, and raw addresses are never stored. Logs record the route template, method, status, request ID and mode, and nothing from the body.

Role	Credential	Used by
public	none	encryption parameters, cashback rates, buyback totals, challenges, integrator registration
merchant	Authorization: Bearer <session token>	merchant, campaign, offer, conversion and transaction routes
agent	SwayAgent signed request	intents, vouchers, purchase reports, tracking links
integrator	SwayIntegrator signed request	integrator profile and agent keys
claim	claim-key signature over a one-use challenge, inside the body	claimables, claims, disputes
receipt	the 128-byte receipt or refund notice itself, Ed25519-signed by the merchant's signer	receipts and refunds; anyone may relay them
oracle	SwayOracle signed request	postbacks, affiliate events, oracle batches
admin	SwayAdmin signed request	merchant, integrator and dispute lists, approvals, activation, suspension, bond release, dispute resolution

13.3 Authentication

Wallet challenges

Wallets and claim keys prove control by signing a one-use challenge that the server issues for one specific request. `POST /v1/auth/challenges` takes {purpose, publicKey, resourceIds, requestHash} and returns the nine-line Sway authorization v1 message shown in §15.3. That message binds the audience origin, the purpose, the key, a 32-byte random nonce, the issue and expiry times (AUTH_CHALLENGE_TTL_SECONDS), the request hash and the hash of the sorted resource IDs. The request hash is the SHA-256 of the canonical request body with authorization removed.

Purpose	Signed by	Resources	Authorizes
merchant_session	merchant wallet	none	a session
integrator_registration	integrator authority	none	creating the integrator
agent_registration	integrator authority	[integratorId]	adding an agent key
claim_read	claim key	none, or [claimId] or [disputeId]	reading claimables, a claim or a dispute
claim	claim key	[claimableId]	one claim
open_dispute	claim key	[intentId]	one dispute case

A seventh purpose, `admin_resolution`, is defined for admin decisions. Dispute resolution itself is authorized by a `SwayAdmin` signed request (below). A proof `{challengeId, publicKey, signatureBase64}` is accepted only for the same key, purpose, request hash, audience and resource set, before expiry, with a valid Ed25519 signature over the exact UTF-8 message. A mutation consumes the challenge by compare-and-set of `consumedAt` inside the same database transaction that performs the change. An exact idempotent read-back may re-verify a consumed proof, but no new mutation can reuse it.

Canonical JSON, used for every request hash in Sway, serializes strings, booleans, null and finite numbers with `JSON.stringify`, arrays in order, and plain objects with keys sorted by code-unit order and no whitespace. Any other value is refused.

Merchant sessions

A merchant signs a `merchant_session` challenge whose request hash covers `{action: "merchant_session", publicKey}` with no resources. In one transaction the server consumes the challenge and creates a session. The token is 32 random bytes in base64url (43 characters), returned once and stored only as its SHA-256. It is never written into an idempotency record. Sessions last `SESSION_TTL_SECONDS`, are sent as `Bearer` tokens and end with `DELETE /v1/auth/merchant/session`. A session belongs to a wallet: it follows that wallet to its merchant once registration finalizes, and a session whose wallet does not own the merchant fails with `WRONG_WALLET`.

Signed requests

Agents, integrators, the affiliate oracle and admins sign every request instead of presenting a bearer secret. The message, built by `createSignedRequestAuthorization` in the contract's `auth.ts` and rebuilt by the server, is nine lines joined by a single line feed (0x0A), with no trailing newline, encoded as UTF-8 and signed with Ed25519:

Sway signed request v1
Audience: <AUTH_AUDIENCE>
Principal: <SwayAgent | SwayIntegrator | SwayOracle | SwayAdmin>
Key: <key ID>
Timestamp: <Unix seconds>
Nonce: <nonce>
Method: <HTTP method, upper case>
Path: <path exactly as transmitted, including any query string>
Body-SHA256: <hex SHA-256 of the canonical JSON body; {} when there is none>

Authorization: <Principal> <key ID>:<timestamp>:<nonce>:<base64 signature>

The key ID is the agent key's or integrator's UUID for SwayAgent and SwayIntegrator, and the base58 public key itself for SwayOracle and SwayAdmin. The timestamp has 1–12 digits. The nonce matches [A-Za-z0-9_-]{16,128}, and the client library draws 24 random bytes encoded as 32 base64url characters. The signature is 64 bytes in standard base64.

Check	SwayAgent	SwayIntegrator	SwayOracle, SwayAdmin
Key	agent key not revoked; its integrator not revoked and approved	integrator not revoked; approved, except for reading its own profile	listed in ORACLE_PUBLIC_KEYS or ADMIN_MULTISIG_ADDRESS
Scope	the operation's scope	—	—
Clock	timestamp within SIGNED_REQUEST_MAX_AGE_SECONDS of server time, either side	same	same
Signature	by the agent key	by the integrator authority	by the listed key
Nonce	one use per principal, stored hashed until timestamp + max age + 1 s	same	same
Rate bucket	per agent key	—	per authority key

The nonce is reserved before the rate bucket is charged, so replaying a captured request cannot drain the victim key's allowance. A client retrying after 429 signs a new nonce and keeps its idempotency key. Agent keys carry least-privilege scopes: intent:write (post, retry, voucher), intent:read (intent and captured book), purchase:report and cashback:link.

An integrator registers publicly with a name, an authority key and a payout address, authorized by an integrator_registration challenge that the authority signs. The payout address must be an existing, on-curve, System-owned wallet with no data and a positive balance, so a token account is rejected. The integrator starts as pending. An admin lists pending integrators and approves one with

POST /v1/admin/integrators/:id/approve, and approval is checked again on every agent and integrator request. Agent keys are added with a SwayIntegrator request plus an agent_registration challenge bound to [integratorId]. A key is unique within its integrator and can be revoked.

13.4 Idempotency keys

Every mutating operation takes an Idempotency-Key of at most 128 characters, scoped to (principal, operation, key). The principal is the authenticated actor, for example agent:<id>, wallet:<publicKey>, claim:<claimKey>, oracle:<publicKey> or receipt:<merchantId>, so two actors cannot collide on a key. The service takes an advisory lock on the scope, compares the stored request hash with the SHA-256 of the canonical body, and either returns the stored response, refuses a changed body with IDEMPOTENCY_CONFLICT, or runs the operation and stores its response in the same transaction (§15.6). The domain work and the idempotency record therefore commit together: a crash between commit and response leaves a retry that returns the original result, and a crash before commit leaves no trace. Stored responses are allowlisted public or ciphertext projections and never include a session token.

Retry	Result
same key, same body	the original response, without repeating the work
same key, different body	409 IDEMPOTENCY_CONFLICT
new key, same domain action	the domain's own uniqueness rule decides; for example, a known receipt digest returns its existing conversion
after 429	same key, new signed-request nonce

13.5 The relay outbox

Sway sends two kinds of transactions. Wallet-signed transactions are prepared by the API and signed by a merchant or admin wallet. Relayed transactions are signed by the worker as fee payer, with the oracle key co-signing where the instruction requires it; the buyback is signed by the relayer alone, as Config.buyback_authority. Both are tracked as ChainOperation rows.

Wallet-signed transactions

The API stores the prepared bytes and their last valid block height with status prepared. The wallet returns the signed bytes to POST /v1/transactions/:id/submit, or broadcasts them itself and reports the signature to POST /v1/transactions/:id/report. Either way the service decompiles the transaction and applies the acceptance rules of §15.5: same blockhash, fee payer and signers, every prepared instruction unchanged and in order, and no new writable account. A submitted transaction is simulated with signature verification, its deterministic signature is committed with status submitted, and only then is it broadcast. A lost RPC response therefore leads to a lookup of that

signature, never to a second, different transaction. The operation is then tracked to `confirmed`, `failed` or `expired`, and here too expiry is decided only after a history lookup of the signature.

Relayed transactions

A relayed operation is identified by `(relayer:<publicKey>, kind, dedupeKey)`, for example `(..., claim, claim:<claimId>:<authorizationHash>)`, and commits to a request hash over `{kind, resourceId, request}`. The same identity with different terms is refused. The worker then:

1. Under a transaction lock on the identity, prepares the transaction, signs it, validates it, runs the operation's `beforePersist` hook (the reversal check of §10.7 is one), and inserts the row as submitted with the unsigned bytes, the signed bytes, the signature and the last valid block height. The signed bytes are committed before any broadcast.
2. Observes the signature with transaction history. Finalized without error means `confirmed`. Finalized with an error means `failed`. A signature that is visible but not yet finalized stays submitted.
3. If the signature is absent and the finalized block height has passed the last valid height, marks the attempt `expired`. History is always consulted before height: a transaction that landed stays landed after its blockhash expires.
4. Otherwise takes a broadcast lease with compare-and-set (owner and expiry, `WORKER_LEASE_SECONDS`), records `broadcastUncertain = true` before the network call, and sends the stored bytes unchanged. After a send error it observes again. Only the first fenced send, with a definite program or preflight rejection, can mark the attempt `failed`. Transport errors leave it pending.
5. Releases the lease.

A new attempt is created only when the latest one is `failed` or `expired`. It gets `attempt = n + 1`, `retryOfId = root` and a derived key `attempt:<SHA-256({root, attempt})>`, and `(retryOfId, attempt)` is unique. Earlier attempts and their signatures stay immutable. Domain code calls `requireFinalized`: `confirmed` continues, `failed` or `expired` raises `COMPUTATION_FAILED`, and anything else raises "work pending", which the queue treats as waiting rather than failing.

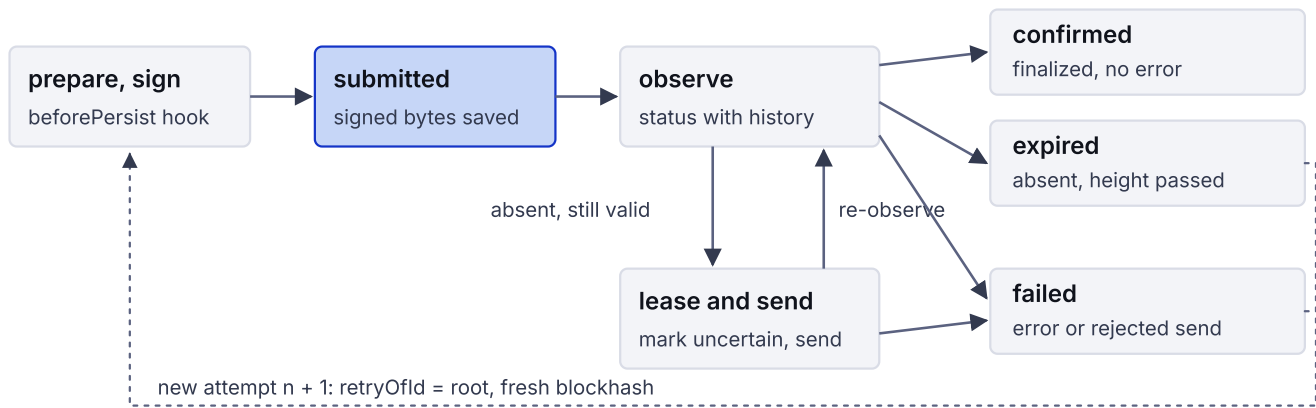


Figure 13.2 — One attempt of a relayed transaction. The signed bytes are saved before the first send, history is checked before expiry, and a new attempt starts only from a failed or expired one.

13.6 Durable jobs, locks and reconcilers

A BackgroundJob row carries a kind, a resource ID, a unique dedupe key, a status, an attempt count, a run time, a lease and a recovery fingerprint. A job is claimed with one statement that selects the oldest due row, queued or with an expired running lease, using `FOR UPDATE SKIP LOCKED LIMIT 1`, and sets a lease of `WORKER_LEASE_SECONDS`. The outcome decides what happens next:

- **Success:** the job becomes succeeded.
- **Work pending** (finality or a computation outstanding): the job returns to queued without consuming an attempt.
- **Any other error:** one attempt is consumed, the job backs off (§15.8), and it becomes failed after `WORKER_MAX_ATTEMPTS`. Only an allowlisted failure code is stored.

Recovery re-arms a failed job, or a succeeded one when asked, only if its state fingerprint changed or its backoff has elapsed. It never takes over a running lease.

Job kind	Work
exchange_post_intent, exchange_match, exchange_retry_intent	post the intent, capture its book, run or retry matching, fall back to cashback
exchange_ingest_offer	ingest a finalized offer submission into its book slot
settlement_conversion	verify a receipt and record and escrow the conversion
settlement_refund, settlement_claim	relay a signed refund notice or claim
cashback_batch, cashback_refund	post an oracle batch event by event (§10.5); refund a reversed commission (§10.7)
buyback	quote a Jupiter route, relay the buyback transaction, then burn any surplus SWAY (§9.8)

Two kinds of PostgreSQL advisory lock serialize work. Transaction-scoped locks (`pg_advisory_xact_lock`) guard idempotency keys, outbox identities, affiliate events, dispute slots and offer-slot allocation. Session-scoped locks guard longer work: `exchange-book:<shard>`, `settlement:<conversionId>`, `dispute:<id>` and `buyback:<programId>`. They run in a separate pool of one connection, so a waiting lock never holds a database connection the work itself needs, and a busy lock returns a retryable `DEPENDENCY_UNAVAILABLE` instead of queueing.

Reconcilers find work that a crash or a delayed finality left behind. Each scan reads one bounded, indexed page of due rows and handles each row in isolation (§15.8 gives the budgets). The relay reconciler runs on every tick, and so does the buyback index, which reads only signatures newer than the last one it indexed. The other domain reconcilers run only when the program's newest finalized transaction signature differs from the one seen on the previous tick.

Domain	Scans	Action
Relay	submitted outbox rows	resume observation and broadcast
Exchange	pending offer ingests; pending intents; intents past their settlement deadline with no open conversion	re-arm ingest or match jobs; expire attempts; <code>close_intent</code>
Settlement	T2 conversions awaiting verification; escrowed conversions with a live claim or refund authorization; terminal conversions past their deadline	re-arm settlement jobs; <code>close_conversion</code> or <code>close_oracle_conversion</code>
Wallet operations	submitted wallet transactions	refresh status and the merchant and campaign mirrors
Cashback	unfinished batches; reversed events whose conversion is pending or escrowed	re-arm batches; cancel unfunded allocations; re-arm refunds
Disputes	<code>under_review</code> user outcomes	confirm the finalized <code>BondResolution</code>
Buyback	the program's finalized transaction signatures since the last indexed one	record every <code>BuybackBurned</code> event, including one sent by another authorized keeper

An escrowed conversion with no claim or refund in flight is not scanned while its lock runs. Its state is read when someone asks for it.

13.7 The database model

PostgreSQL is accessed through Prisma. Money is stored as `Decimal(20,0)` with unsigned-u64 checks, carried as decimal strings in the API, and computed with `bigint`, so no amount ever passes through a floating-point number in the services.

Group	Tables	Holds
Identity	WalletChallenge, MerchantSession, Merchant, Integrator, AgentKey	challenges with hashed nonces, hashed session tokens, public keys, payout wallets, scopes, approval and revocation
Market	Campaign, OfferBook, Offer, BookRevisionSnapshot	chain mirrors and slot ownership; encrypted offers and their MXE copies; the encrypted book at each revision
Intents	Intent, IntentEncryptionGuard, Computation	ciphertext, the per-intent claim and x25519 public keys, sealed results, the captured book, deadlines; a permanent (sender, nonce) tombstone with no foreign key; computation attempts
Settlement	Conversion, Claim, Refund, PurchaseReport, Dispute	receipts and their hashes, signed claim and refund authorizations, order and evidence digests, dispute cases
Cashback	AffiliateRate, AffiliateTrackingLink, AffiliatePostback, OracleBatch	the public table, intent-bound links, signed events and their states, batches with per-event outcomes
Work	ChainOperation, BackgroundJob, HttpIdempotency	the outbox and its attempt lineage, leased jobs, stored responses
Buyback	BuybackRun, BuybackBurn, BuybackIndexCursor	each run's amounts and compiled transaction; finalized burns by signature; the position of the signature index
Limits	RateLimitBucket, SignedRequestNonce	hashed rate buckets; one-use request nonces

Unique constraints back the protocol's one-time rules off chain as well as on chain: one conversion per intent and slot, one per receipt digest, one per merchant and order digest, one claim per conversion, one dispute per intent and slot, one postback per (network, networkEventId), one tracking link per intent and rate, and one live offer per book slot (a partial index over pending and accepted offers). No table has a column for a plaintext preference, offer rule, price or winning slot before a conversion exists.

13.8 The MCP package

@sway/mcp is the agent-side runtime. It is a Node 22 process (sway-mcp) that speaks the Model Context Protocol over stdio. It keeps every private input on the device: it encrypts intents locally, decrypts sealed results locally, and signs with keys that never leave it. It exposes five tools whose schemas come from the contract package. Each tool returns {data, error, meta}, and a failure also sets isError. Only claimable is marked read-only, and only claim is marked destructive.

Tool	Input	What it does
<code>find_offers</code>	plaintext intent, optional <code>intentId</code> , optional <code>cashbackOnly</code>	encrypts and posts the intent, polls, retries once, decrypts the top three
<code>get_voucher</code>	<code>intentId</code> , <code>slot</code>	POST <code>/v1/intents/:id/voucher</code> , returning <code>{intentId, intentAccountAddress, slot, attachAtCheckout: "sway:<intentId>:<slot>"}</code>
<code>report_purchase</code>	<code>intentId</code> , <code>slot</code> , <code>orderHash</code> , <code>evidenceHash</code> , optional <code>encryptedEvidenceReference</code>	files the digests; the checkout receipt stays on the device as dispute evidence
<code>claimable</code>	<code>intentId</code>	signs a <code>claim_read</code> challenge with the intent's claim key and lists amounts, unlock times and states
<code>claim</code>	<code>intentId</code> , <code>claimableId</code> , <code>destination</code>	signs the SWAYCLM1 claim message and a claim challenge, then submits the claim

`find_offers` runs these steps:

1. It hashes the plaintext input. Reusing an intent ID with a different input fails with `IDEMPOTENCY_CONFLICT`.
2. It reads `GET /v1/encryption` for the MXE public key, the chain clock and the purchase window, and sets the purchase deadline to `chainUnixTimestamp + purchaseWindowSeconds` (§4.4).
3. It generates the intent's x25519 keypair and claim keypair, encrypts the intent, and stores the keys and the exact request body before anything is sent.
4. It posts with idempotency key `mcp-intent-<intentId>`, then polls up to 60 times (`SWAY_MCP_MAX_POLLS`) at 500 ms intervals (`SWAY_MCP_POLL_INTERVAL_MS`). On expired at attempt 1 it calls `retryIntent` once (`mcp-retry-<intentId>-2`).
5. On matched it decrypts the sealed result and checks its intent and category bindings. It then fetches the captured book (`GET /v1/intents/:id/book`), resolves each winner's campaign and brand hashes against the full slot map, and computes each score with the user's own brand bonuses. Sway never learns which slots won.

For `claim`, the tool reads the chain clock and builds the claim message with an expiry 120 seconds past chain time and 16 random nonce bytes. It signs the message with the claim key and saves the reservation before obtaining a `claim` challenge bound to [`claimableId`]. If a saved reservation has expired, the tool replaces it only after a fresh `claimable` read shows the conversion still `claimable`, which proves the earlier attempt ended without paying. It refuses to change the destination while an earlier authorization is still valid.

Keys live in the operating system's credential store through a keychain adapter, under the service name `sway-mcp-` followed by the first 24 hex characters of the SHA-256 of the API URL, so each API has its own set of keys; `SWAY_MCP_KEYCHAIN_SERVICE` overrides the name. Each entry is limited to 2,400 bytes of JSON, below the 2,560-byte credential limit of the Windows Credential Manager. The entries are the integrator authority and agent key seeds, the registration profile, `intent:<id>:keys` (claim key seed, x25519 secret, MXE public key, input hash), `intent:<id>:request` and `claim`:

<claimableId>. Ed25519 keys are stored as 32-byte seeds, and secret buffers are zeroed after each use. Outbound requests refuse redirects and time out after 15 seconds. Every agent call carries a fresh SwayAgent signature with its own timestamp and nonce. Stdout carries MCP JSON-RPC only, and errors are reported without payloads or key-store details.

14. Privacy model

Sway's privacy design gives each party the data its role requires, at the point in the flow where the role requires it, in the narrowest form that still lets the protocol work. Intents and offers are encrypted on the device and in the browser where they are written. Matching and verification run on secret shares inside the MXE. Results are sealed to a key created for a single intent. Documents that matter only for integrity, such as order references and checkout evidence, travel as SHA-256 digests. Settlement is public by design: when a conversion moves USDC, its amounts are written to the chain.

14.1 What each party learns

Each row describes what a party receives in its role. Every party, brands included, can also read the public chain. The public chain row lists what that reveals, and the other rows do not repeat it.

Party	Learns	Never learns
Losing brands	Nothing from the protocol: no result, notification or callback reaches them	Whether their offer was evaluated against an intent, whether it was eligible, how it ranked, or what the intent contained
Winning brand	The order, at purchase, as any merchant would; the intent account and slot named by the voucher	The user's other preferences, rival offers, the user's wallet
Sway (API, worker, relay)	Category shard, timing, agent key and its integrator, the slot named in a voucher request, conversion amounts, the claim destination	Intent contents, offer rules, the other entries of a top three and their order
Arcium MXE nodes	The public arguments of each computation and its revealed bits	Plaintext intents, offer rules or results; a node holds only shares
Shopper's agent	Its own intent; for each offer in its top three, the slot, price, rebate, campaign and brand	Offers outside its top three; the targeting rule of any offer
Public chain	Campaign spend per conversion; shard, timing and integrator of each intent; which campaigns occupy a book	Who received a rebate, unless the user links claims

	Intent contents	Offer rules	Top three	Chosen slot	Price and rebate	Payout address
Shopper's agent	writes it	top three: price, rebate	decrypts	chooses	own order	chooses
Winning brand		own rule		at checkout	signs receipt	
Losing brands		own rule				
Sway API, relay	ciphertext	ciphertext	ciphertext	at voucher	from receipt	at claim
Arcium MXE nodes	shares	shares	shares	at verify	at verify	
Public chain	ciphertext	ciphertext	ciphertext	after purchase	after purchase	at claim

plaintext, from the point shown
 ciphertext or secret shares
 not received

Figure 14.1 — Who sees each item, and in what form. Rows show what a party receives in its role; everything in the public chain row is readable by every party.

14.2 The shopper's agent

The agent is the only party that sees the intent in plaintext and the only party that can open the result. For each intent it creates an x25519 keypair and an Ed25519 claim keypair, keeps both secrets in the device keychain, and encrypts the intent under the shared secret of its x25519 key and the MXE's public key (§4). `match_intent` seals its output to the owner of the encrypted input (`input.owner.from_arcis`), so only the secret that encrypted the intent can open its result. The binding is enforced at three layers: the API rejects a request whose result key differs from the ciphertext's sender key, the program rejects it with `InputKeyMismatch`, and the circuit takes no separate recipient argument. A party that copies an intent ciphertext and reposts it under a key of its own therefore obtains nothing it can open, and reposting the original key and nonce is refused by the `InputReplay` account and by the API's encryption guard.

The result tells the agent about at most three offers: for each, the slot, price, rebate, campaign hash and brand hash, together with the number of matches and the category and intent hashes that bind the result to this intent. The agent checks those bindings and resolves the campaign and brand hashes against the public book map from `GET /v1/intents/:id/book`. That endpoint always returns all 32 captured slots, so the request reveals nothing about which entries the agent is resolving. Targeting rules, meaning size ranges, widths, region and the new-to-brand flag, are never part of the output, and offers that missed the top three never reach the agent in any form.

14.3 Losing brands

Matching reads every slot of the book inside the MXE and writes its output only to the intent account, sealed. No callback, event or message reaches any brand at auction time, including the brand that ranked first; notifying brands would cost callback bytes and would leak win rates. A losing brand therefore cannot tell whether its offer was evaluated against a given intent, whether it was eligible, or how it ranked. The intent itself appears to the brand exactly as it appears to every observer: an account in its shard that holds ciphertext.

14.4 The winning brand

A brand learns about a sale only when the sale happens, through its own checkout. The order shows what any merchant sees: the product, the size bought, the price and whatever delivery details its checkout collects. The voucher adds the intent account address and the slot, which the receipt signer needs. The receipt binds those to a SHA-256 hash of the order reference, so the order itself never reaches Sway or the chain. Because its offer converted, the brand can infer that the buyer met its eligibility rule.

The brand never sees the intent's maximum price, its bonuses for other brands, the brands the user already owns, the other entries of the top three, or any rival's terms. It never receives a wallet address from the protocol either. The voucher and the receipt carry none, and the user names a payout destination only at claim time, after the lock. That destination can be any address, including one used for nothing else.

14.5 Sway's API, worker and relayer

The service handles every request, so its view is the operator's view. It sees the shard named in each request, the timing of requests, the agent key that signed each one and the integrator behind that key. It stores intent and offer ciphertext, public keys, the sealed result and the MXE copy, and it holds no key that decrypts any of them: an `Enc<Shared>` input opens only with the agent's or brand's secret or with the MXE's key material, and `Enc<Mxe>` ciphertext opens only inside the MXE.

The service learns more at three points, each required for settlement. When the agent requests a voucher it names the chosen slot, so Sway learns which single offer the user intends to buy, but not the other entries of the top three or their order. When a receipt is relayed, Sway learns the purchase amount and the order hash, and with them the conversion's rebate and fees. When a claim is relayed, it learns the destination. The relayer is part of this service: it signs transactions as the fee payer, and the data it handles is the data in those transactions.

For relayed intents, `post_intent` uses the relayer key as both payer and agent signer. The agent key that authorized the HTTP request is known to Sway and absent from the chain.

14.6 The MXE nodes

The Arx nodes hold every encrypted input as secret shares and compute under Arcium's Cerberus protocol, which is secure with a dishonest majority: any node, or any coalition short of the whole cluster, holds shares that reveal nothing about the inputs. The nodes see in the clear only each computation's public arguments and the outputs it explicitly reveals.

Circuit	Public arguments	Revealed output
ingest_offer	Campaign per-conversion cap, expected brand hash, expected category hash	One validity bit
match_intent	Active-slot mask; campaign and brand hash of each slot; category hash; intent hash	Nothing: one output is sealed to the agent, the other to the MXE
verify_conversion	Claimed slot, purchase amount, rebate, campaign, brand, category and intent hashes	One bit

When `verify_conversion` runs, its slot, purchase amount and rebate are already public on the Conversion account, and its hashes identify the campaign, brand, category and intent that the conversion already names. Verification adds one bit to what the chain shows.

14.7 What is public on chain

Account or transaction	Public	Sealed
OfferBook	Shard hash, revision, campaign address of each slot, active mask, element counts, slot versions	Each slot's rule (Enc<Mxe>) and nonce
Merchant, Campaign	Receipt signer, bond account, return window, status; caps, lock days, funding, daily and total spend, pending reservations	—
Intent	Shard hash; relayer and integrator keys; claim and x25519 public keys; nonce, whose first eight bytes are the purchase deadline; deadlines, attempt and status; book revision and mask; input commitment and snapshot hash	The intent; the top three sealed to the agent; the MXE copy
IntentBook	The 32 campaign addresses and the active mask captured at posting	—
Conversion	Intent, campaign, slot, proof tier, purchase amount, rebate, both fees, receipt and order hashes, unlock time, status	—
OracleEvent	Network event hash, source token account, commission	—
claim transaction	Destination wallet and token account; the signed 128-byte claim message	—

The chain therefore shows campaign spend per conversion, and for each conversion the price and rebate of the offer that sold. Offers that never convert are never revealed. The intent result is a

fixed-size ciphertext whose element count does not depend on the number of matches, so the chain does not even show whether an intent found any eligible offer.

14.8 Linking intents and claims

Nothing in the protocol requires two intents of the same user to share an identifier. Each intent has its own random REST identifier, its own x25519 key and its own claim key, and the API refuses a claim key that already belongs to another intent (REPLAY_DETECTED). On chain, an intent names the relayer as its agent signer and the integrator's payout wallet as its integrator, which ties it to an agent product rather than to a person.

Claims are where linking becomes the user's choice. The claim key signs a message naming one conversion and one destination, the relayer pays the fee, and the destination needs no SOL and no earlier on-chain activity. A user who sends each claim to a fresh address keeps conversions unlinked; a user who sends several claims to one address links them, deliberately. The API does not act as a lookup service for balances: listing claimables requires a fresh signature from the claim key.

14.9 Data minimization in the API

- Every transport schema rejects unknown fields. The encrypted-input schema accepts only ciphertext elements, a nonce, a sender public key and the encoding name, so a plaintext attribute cannot be stored by accident.
- PostgreSQL holds ciphertext, nonces and public keys, with SQL checks on element and nonce lengths. There are no plaintext columns for rules, preferences, prices, rebates or winning slots before a conversion exists, and the sealed result is stored as opaque bytes.
- Purchase reports and disputes carry a SHA-256 evidence digest, the order digest and, optionally, an opaque reference to evidence the agent encrypted itself. The checkout receipt stays on the device.
- Session tokens contain 256 random bits and only their hashes are stored. Challenge nonces and messages, signed-request nonces and rate-limit keys are stored as hashes, and client IP addresses are HMAC-hashed with a deployment salt before storage.
- Idempotency records keep only allowlisted public or ciphertext response projections, and the database rejects secret-bearing property names.
- Logs record route templates, methods, statuses and request identifiers, never bodies, authorization headers, keys, emails or raw dependency errors. Failed jobs record allowlisted error codes only.
- Validation errors name the field and never echo the submitted value.
- The service never generates or receives a private key belonging to an agent, a user or a brand.

14.10 The cashback path

In the cashback path the affiliate network attributes the sale, so the purchase is visible to it as it would be for any affiliate sale. The rate table is public, so reading it reveals nothing about the user. When the agent requests a tracking link it names a rate and a destination URL, so Sway learns which merchant the user intends to buy from. The link's sub-ID is the random intent identifier, never a wallet or agent identifier. The oracle's signed postback gives Sway the order hash, the commission, the purchase time and, when the network supplies it, the purchase amount; without it, the conversion records zero with `purchaseAmountKnown = false`. On chain, the T3 conversion shows the commission, its split and the oracle's source account, and it is claimed with the same per-intent claim key as a brand-direct conversion (§10).

15. Security design

Sway moves USDC on the strength of signatures from several independent parties: the shopper's agent device, the brand's wallet and receipt signer, the Arcium MXE, the affiliate oracle, the Sway relayer and the Config admin. The security design gives each of those signatures exactly one meaning. A signature names its purpose in its first bytes, names the program and the accounts it acts on, carries its own freshness, and is consumed once. The `sway_exchange` program then decides every movement of funds from its own account state, and every off-chain step that leads there can be repeated without repeating its effect.

This chapter follows those mechanisms layer by layer: message construction (§15.1, §15.2), replay and cross-use guards (§15.3), key custody (§15.4), wallet transaction validation (§15.5), idempotency (§15.6), finality (§15.7), worker bounds (§15.8) and the HTTP boundary (§15.9). Exact byte layouts are collected in Appendix A (§17).

15.1 Domain separation

Every signed or hashed construction begins with a fixed prefix that no other construction shares, so a signature or digest produced for one purpose never verifies or matches under another.

Construction	Prefix	Produced by	Checked by
T2 receipt, 128 bytes	SWAYRCP1	Merchant receipt signer	API, record_conversion
Refund notice, 128 bytes	SWAYREF1	Merchant receipt signer	API, refund_conversion
Claim authorization, 128 bytes	SWAYCLM1	Per-intent claim key	API, claim
Wallet challenge	Sway authorization v1	Wallet, claim key or integrator authority	API (AuthService)
Signed HTTP request	Sway signed request v1	Agent, integrator, oracle or admin key	API (AuthService)
Affiliate postback attestation	Sway affiliate postback v1	Affiliate oracle gateway	API (affiliate adapter)
Intent input commitment	sway/intent-input/v1\0	post_intent	match_intent_callback
MXE snapshot hash	sway/mxe-snapshot/v1\0	match_intent_callback	record_conversion, verify_conversion_callback, API
Identifier hash	sway/<domain>/v1\0, domain $\in \{\text{brand, category, campaign, intent}\}$	Agent, brand, MXE	Circuits, agent
Plaintext packing tag	SWAYINT1, SWAYOFF1, SWAYTOP2 in the low 64 bits of metadata	Agent, brand, MXE	Circuits, agent

The three 128-byte messages share one shape: an 8-byte domain, fields at fixed offsets, and reserved bytes that must be zero. Decoders reject a message whose domain differs or whose reserved tail is non-zero (`requireDomain` in `settlement.ts`, `Receipt::decode` in `lib.rs`), so no byte of a signed message is left free to vary.

The packing tags do the same job inside ciphertext. The circuits' `valid_intent` and `valid_offer` and the agent's `unpackTopThree` reject a decrypted payload whose tag or reserved metadata bits differ from the expected type. An offer ciphertext cannot be posted as an intent, and a result cannot be read as an input. Unpacking is canonical as well: `canonicalUnpack` repacks the decoded fields and requires the identical field elements, so one plaintext has exactly one valid encoding.

15.2 Binding to program and accounts

Program id

Each 128-byte message carries the program id: receipt bytes 93–124, refund bytes 72–103, claim bytes 8–39. The program checks `r.program == crate::ID` for receipts, and rebuilds refund and claim messages with `crate::ID` before comparing them byte for byte. A message signed for one deployment never settles on another. The API applies the same check against its configured program id. Its chain adapter starts money operations only after the finalized `Config` account shows the configured USDC mint and admin, and the program's `ProgramData` shows that admin as upgrade authority.

Accounts

- **Receipt.** The receipt names the intent account (bytes 8–39) and a book slot (byte 40). `record_conversion` requires the intent key to equal `r.intent` and the instruction's `slot` argument to equal `r.slot`. The slot must be active in the intent's captured `IntentBook`, and the captured campaign at that slot must equal the campaign account passed in. The conversion PDA is derived from `["conversion", intent, slot]`, so a receipt can only ever create the conversion for its own intent and slot.
- **Refund.** The notice names the conversion account and the SHA-256 hash of the original receipt. `refund_conversion` compares it with `refund_message(conversion.key(), conversion.receipt_hash, merchant.signer_epoch)`.
- **Claim.** The authorization names the conversion account and a destination wallet. `claim` compares it with `claim_message(conversion.key(), destination.owner, expiry, nonce)`, where `destination.owner` is the owner recorded in the destination USDC token account. The relayer that submits the claim cannot redirect it: any other destination changes the message and fails the signature check. The integrator's share goes to a token account whose owner must equal `intent.integrator`, and the Sway fee goes to the treasury recorded in `Config`.

Ed25519 verification

The program verifies all three messages through Solana's Ed25519 precompile and instruction introspection (`verify_ed25519`). The precompile instruction must sit immediately before the Sway instruction (`index current - 1`), reference no accounts and carry exactly one signature. Its three instruction-index fields must equal `u16::MAX`, so signature, key and message are read from its own data, and the bytes at its key and message offsets must equal the expected signer and the expected 128-byte message (layout in Appendix A.5). A precompile that verified another key or message, or that borrowed bytes from another instruction, does not satisfy the check.

Encrypted inputs and results

`post_intent` requires the input's encryption key to equal the key that receives the result (`input_x25519 == user_x25519`, error `InputKeyMismatch`). Both keys must be canonical x25519 coordinates, non-zero and below $2^{255} - 19$ (`canonical_x25519`). The instruction then records a commitment to everything the computation depends on:

```
input_commitment = SHA-256("sway/intent-input/v1\0" || intent || SHA-256(i
```

`match_intent_callback` accepts a result only when the MXE presents the same commitment. A result computed over another book revision, ciphertext, claim key or integrator is refused with `SnapshotMismatch`. The callback stores the MXE copy of the top three with its `snapshot_hash`. `record_conversion` and `verify_conversion_callback` recompute that hash from the stored ciphertext before acting on it.

The plaintext binds itself too. The sealed top three carries identifier hashes of the category shard and the intent account address in its `category` and `intent` fields. The agent rejects a decrypted result whose hashes differ from its own intent and category, and the `verify_conversion` circuit requires the same hashes before it reveals a verdict.

The production build of the program (feature `arcium-financial`) fixes the identities those circuits compare on chain, with the same `sway/<domain>/v1\0` construction (`identifier_hash`). A merchant's brand hash is the identifier hash of its wallet address, a campaign's hash that of its UUID, and a book's category hash that of its shard text. Offers must carry exactly 2 ciphertext elements and intents exactly 8, so every input has the one shape the circuits decode.

Book slots

A slot belongs to one campaign while it is active: `submit_offer` and `ingest_offer_callback` refuse an active slot held by another campaign (`CampaignMismatch`). Each slot has a `slot_versions` counter that increments on every write or clear. An offer submission records the version it was made against, and an ingest callback for a superseded version is refused as `StaleCallback`. Intents capture the campaign map and active mask at posting time in their own `IntentBook` account, so later book changes never alter what a receipt for that intent may claim.

15.3 Replay and cross-use prevention

One-use challenges

Claim-key reads and actions, merchant sign-in, integrator registration, agent-key registration and dispute opening authenticate with a challenge. The client sends `{purpose, publicKey, resourceIds, requestHash}` and receives a nine-line message to sign:

```
Sway authorization v1
Audience: <exact service origin>
Purpose: <purpose>
Public-Key: <base58 key>
Nonce: <32 random bytes, base64url>
Issued-At: <ISO 8601>
Expires-At: <ISO 8601>
Request-SHA256: <hex>
Resources-SHA256: <hex>
```

requestHash is SHA-256 over the canonical JSON of the request body with the authorization field removed (keys sorted, no whitespace). Resources-SHA256 covers the sorted resource ids, such as the claimable id for a claim or the intent id for a dispute. The API stores the message hash and the nonce hash. It accepts a proof only when the stored purpose, key, request hash, resource set and audience all equal those of the request being authorized.

A challenge expires after AUTH_CHALLENGE_TTL_SECONDS. It is consumed by a conditional update (consumedAt IS NULL AND expiresAt > now) inside the same database transaction that performs the action, and a second use returns REPLAY_DETECTED. A proof for a claim read cannot authorize a claim, and a claim proof for one claimable cannot authorize another. Before signing, the agent checks that the message has exactly nine lines naming its configured audience, the purpose, its key, the request hash and the resource hash.

A merchant sign-in proof (purpose merchant_session) issues a session token of 32 random bytes. The API stores only its SHA-256 hash, and the token expires after SESSION_TTL_SECONDS.

Signed requests

Agents, integrators, the oracle and the admin sign each HTTP request instead of presenting a bearer secret. The signed text names the audience, the principal scheme (SwayAgent, SwayIntegrator, SwayOracle or SwayAdmin), the key id, a Unix timestamp, a nonce, the method, the path and the SHA-256 of the canonical body. The API accepts a timestamp within SIGNED_REQUEST_MAX_AGE_SECONDS of its clock and reserves the nonce for that principal until the window has passed. A reused nonce returns REPLAY_DETECTED. The nonce is reserved after the signature verifies and before the rate-limit bucket is charged, so a replayed copy of a valid request never spends the legitimate key's quota.

Nonces and expiries on chain

- **Intent input.** The 16-byte input nonce holds the purchase deadline in bytes 0–7 (u64 little-endian Unix seconds) followed by 8 random bytes. `post_intent` requires `now < deadline ≤ now + purchase_window_secs` and stores the deadline as `Intent.purchase_deadline`; `settlement_deadline` adds `settlement_grace_secs`. MXE results, receipts and oracle conversions are accepted only inside those bounds.

- **Ciphertext reuse.** `submit_offer` and `post_intent` both initialize an `InputReplay` PDA at `["input", sender_x25519, nonce]`, so a (key, nonce) pair enters the program once. The API mirrors this with an `IntentEncryptionGuard` row that binds the pair to one intent id and one ciphertext hash, and it refuses a claim public key already bound to another intent.
- **Claims.** A claim authorization carries an expiry (bytes 104–111, checked as `expiry ≥ now` against the chain clock) and a 16-byte random nonce (bytes 112–127). `@sway/mcp` sets the expiry from the chain clock reported by the API plus its claim lifetime, 120 seconds by default (`claimTtlSeconds`).
- **Prepared transactions.** Every transaction the API prepares ends with a `request_nonce` instruction carrying 16 random bytes. Two identical wallet actions, such as two equal deposits, become distinct messages with distinct signatures, and neither can pass as a retry of the other.

Uniqueness by account address

The program expresses each at-most-once rule as an account that can be initialized only once:

PDA seeds	Created by	Guarantees
<code>["intent", id16]</code>	<code>post_intent</code>	One intent per id
<code>["input", x25519, nonce]</code>	<code>post_intent</code> , <code>submit_offer</code>	One use per encrypted input
<code>["offer", campaign, id16]</code>	<code>submit_offer</code>	One submission per offer id
<code>["conversion", intent, slot]</code> , slot 255 for T3	<code>record_conversion</code> , <code>oracle_post_conversion</code>	One conversion per intent slot
<code>["order", merchant, order_hash]</code>	<code>record_conversion</code>	One conversion per merchant order
<code>["oracle_event", event_hash]</code>	<code>oracle_post_conversion</code>	One conversion per network event
<code>["bond_resolution", merchant, case_hash]</code>	<code>slash_bond</code>	One bond debit per dispute case
<code>["signer_epoch", merchant, epoch]</code>	<code>register_merchant</code> , <code>rotate_receipt_signer</code>	One signer per epoch

State flags complete the set. A conversion pays out only from status 1 (escrowed) and moves to 3 (claimed) or 2 (refunded) in the same instruction that moves the USDC, so a second claim fails with `AlreadyConsumed` and a second refund with `RefundWindowClosed`. An MPC callback must match the intent's current attempt, arrive before `expires_at_slot` (180 slots after queueing, `QUEUE_SLOTS`), and find the intent or submission still pending; anything else is a `StaleCallback`.

`verify_conversion_callback` also requires the conversion's stored `receipt_hash` to match the one it verified. Figure 15.1 follows one claim through every guard it meets.

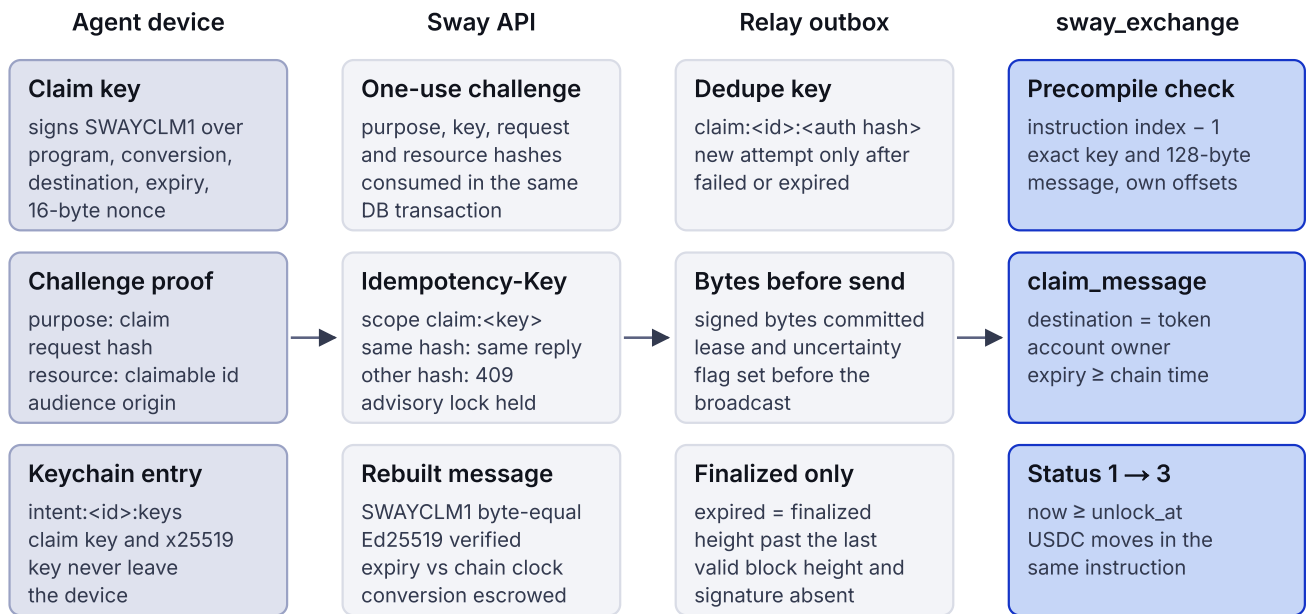


Figure 15.1 — The guards one claim passes, from the device keychain to the status change in sway_exchange. Each column checks the same authorization independently.

15.4 Key custody

Key	Held by	Signs or decrypts	Registered as	Rotation or revocation
Per-intent x25519 key	Agent device, OS keychain	Encrypts the intent; decrypts the sealed top three	<code>Intent.user_x25519</code>	Fresh per intent
Per-intent claim key (Ed25519)	Agent device, OS keychain	Challenges for <code>claim_read</code> , <code>claim</code> , <code>open_dispute</code> ; <code>SWAYCLM1</code>	<code>Intent.claim_key</code>	Fresh per intent; on intent per key
Agent request key (Ed25519)	Agent device, OS keychain	SwayAgent requests	Agent key with scopes under an integrator	<code>revokeAgentKey</code> ; revoked keys and keys of revoked integrators are refused
Integrator authority (Ed25519)	Agent developer	Registration challenges, SwayIntegrator requests	Integrator	Operator approval allowlist
Merchant wallet	Brand's browser wallet	Prepared transactions, <code>merchant_session</code> challenge	Seed of the Merchant PDA	Fixed per merchant account
Receipt signer (Ed25519)	Brand console's browser profile, or the store's own server	<code>SWAYRCP1</code> , <code>SWAYREF1</code>	<code>ReceiptSignerEpoch</code>	<code>rotate_receipt_sig</code>
MXE callback signer	Arcium cluster	<code>ingest_offer</code> , <code>match_intent</code> , <code>verify_conversion</code> callbacks	<code>Config.mxe</code>	<code>update_config</code> by the admin
Relayer	Sway service	Fee payer for relayed transactions; <code>buyback_begin</code> and <code>buyback_finish</code>	Required signer of relayed transactions; <code>Config.buyback_authority</code>	Service configuration; <code>update_config</code> by the admin for the buyback authority
Affiliate oracle	Sway oracle	<code>oracle_post_conversion</code> , <code>oracle_refund_conversion</code> , postback attestations	<code>OracleAuthority.oracle</code> with epoch; API allowlist	<code>rotate_oracle</code> by the admin
Config admin	The wallet the deployment names (<code>ADMIN_MULTISIG_ADDRESS</code>); a multisig wallet can serve	Activation, suspension, configuration, bond release and resolution	<code>Config.admin</code> , handed off from the upgrade authority at bootstrap	<code>rotate_admin</code> , signed by the old and the new admin

Device keys. @sway/mcp generates the per-intent x25519 key and claim key when it encrypts an intent, and stores both under one keychain entry per intent (`intent:<id>:keys`) together with the MXE key used and a hash of the plaintext intent. The agent request key and the integrator authority

have entries of their own. Entries are capped at 2,400 bytes to fit the 2,560-byte Windows credential limit, and seeds are zeroed in memory after each signature. The REST contract has no field that accepts a private key: `PostIntentRequest` carries ciphertext, the x25519 public key and the claim public key only. An intent id cannot be reused for a different plaintext, because the stored hash must match. Fresh claim keys keep separate rebates unlinkable unless the user claims them to the same address.

Receipt signer epochs. `register_merchant` opens epoch 0 for the merchant's receipt signer, valid from registration time with `valid_until = i64::MAX`. `rotate_receipt_signer` closes the current epoch at the current time, increments `Merchant.signer_epoch` and opens the next epoch account for the new key. A receipt names its epoch in bytes 89–92. The program loads `["signer_epoch", merchant, epoch]`, verifies against that epoch's signer, and requires the receipt timestamp to fall inside `[valid_from, valid_until]`. A purchase made before a rotation still settles with the old key, and a retired key cannot sign for anything dated after its retirement. Refund notices always use the current signer and the current epoch, so a retired key cannot reverse a conversion.

Admin and bootstrap. `initialize` succeeds only when its signer is the upgrade authority recorded in the program's `ProgramData` (`verify_bootstrap`). The mint must equal the USDC mint pinned into the build (`PINNED_USDC_MINT`, set from `SWAY_PINNED_USDC_MINT`) and have 6 decimals. Every policy must be explicit: `rounding_policy == 1` and a positive vault floor, minimum bond, dispute window, purchase window and settlement grace. In the production build, `Config.mxe` must be the `financial_authority` PDA of the Arcium coordinator program, so MPC callbacks can only arrive through Arcium, and only the admin can create a book (`create_production_book`). The deployer key keeps the upgrade authority, and the services check that the `ProgramData` names it (§12.7). The API accepts `SwayAdmin` requests only from the configured `Config` admin address (`ADMIN_MULTISIG_ADDRESS`) and `SwayOracle` requests only from the configured oracle keys. The MXE's encryption key is separate from its Ed25519 callback signer.

15.5 Wallet transaction validation

Brands sign Sway transactions in their own wallets. For each action the API builds the instruction with the account list the program expects, sets a finalized blockhash and appends `request_nonce`. It serializes a legacy transaction, or a version 0 transaction through the deployment's address lookup table when the legacy form would exceed 1,100 bytes. The unsigned bytes and the blockhash's `lastValidBlockHeight` are stored as a chain operation in `status prepared`.

Wallets do not always return those bytes unchanged: they set their own compute limit and priority fee, and Phantom appends Lighthouse assertion instructions that check the resulting account state. When the wallet returns the signed transaction (`POST /v1/transactions/:id/submit`), `validateSignedTransaction` decompiles both messages, resolving lookup tables at finalized commitment, and accepts the signed one only under these rules:

1. The payload is canonical standard base64, at most 1,232 bytes, and re-serializes to the identical bytes.

2. The recent blockhash and the fee payer equal the prepared ones. The required signers are the same keys in the same order, the signed-in wallet is among them, and every signature verifies over the signed message.
3. Every prepared instruction other than `ComputeBudget` appears unchanged and in order: same program, same data, same accounts with the same signer and writable flags. None is omitted.
4. `ComputeBudget` instructions may be added, replaced or dropped. They carry no accounts and may not follow the last prepared instruction.
5. After the last prepared instruction, only instructions of the Lighthouse program (`LIGHTHOUSE_PROGRAM_ID`, `L2TeXMFKdjpN9kozasaurPirfHy9P8sbXoAN1qA3S95`) may appear, and nothing else may follow them.
6. No account is writable unless the prepared message already writes it. A Lighthouse guard can read any account but gains no new write access.

A fresh submission is then simulated with signature verification on and the original blockhash kept, before the API writes an idempotency record or changes the operation; a failing simulation returns `TRANSACTION_SIMULATION_FAILED`. The API commits the signature and signed bytes before the first broadcast. A repeated submission resends the same bytes, and the API never builds a new message for an attempt whose outcome is still unknown.

A wallet that broadcasts the transaction itself reports the signature instead (`POST /v1/transactions/:id/report`). The API loads that transaction, rebuilds its signed bytes, applies the same six rules, requires the fee-payer signature to equal the reported one, and records the outcome; a transaction that landed with an error carries `TRANSACTION_FAILED`.

Transactions the relayer signs are built and signed by the service itself. They pass the same validation, with the relayer as the required fee payer, before their bytes enter the outbox (§15.6).

15.6 Idempotent operations

Operations that change protocol state are marked `idempotent: true` in `operations.ts` and require an Idempotency-Key header of 1 to 128 characters. The key is scoped to its principal and operation; principals include `wallet:<key>`, `agent:<id>`, `claim:<claim key>`, `receipt:<merchant>` and `oracle:<key>`. Under a PostgreSQL transaction-level advisory lock on that scope, the API looks for a prior record. The same request hash returns the stored response, a different request hash returns `IDEMPOTENCY_CONFLICT`, and an unfinished prior attempt returns a retryable `DEPENDENCY_UNAVAILABLE`. Stored responses hold only public data and ciphertext; session tokens are never cached.

Chain work goes through a durable outbox (§13). A chain operation is identified by (`principal`, `kind`, `dedupe key`) and carries a hash of its immutable terms; dedupe keys embed what makes the action unique, such as `claim:<claim id>:<authorization hash>` or `refund:<refund id>:<notice hash>`, and the same identity with different terms is refused. A new attempt, linked by `retryOfId`, exists only after the previous one is proven failed or expired. Signed bytes are committed before broadcast, and the broadcaster records `broadcastUncertain` under a lease before calling the network, so an interrupted send stays observable. Background jobs are unique by dedupe key, and

work on one book or one conversion runs under a non-blocking advisory lock that answers a busy resource with a retryable 503.

15.7 Finality

The API treats finalized chain state as the source of truth for money. Its chain adapter reads every account at `finalized` commitment and checks the owner (the Sway program), the exact data length and the Anchor discriminator before decoding. Merchant bonds, campaign balances, conversion states and unlock times shown through the API come from those finalized reads, whatever status a wallet transaction reports.

A relayed operation becomes confirmed only when its signature status is `finalized` without error, and `failed` when it finalized with an error. It becomes `expired` only when the finalized block height has passed its `lastValidBlockHeight` and the signature is absent from history, because a landed transaction can outlive its blockhash. A step that depends on an earlier operation calls `requireFinalized`: a confirmed operation lets it proceed, a failed or expired one ends it with `COMPUTATION_FAILED`, and anything else raises `WorkPending`, which requeues the job without counting a failed attempt.

Before relaying a receipt, the API re-reads the intent account and requires the ciphertext, sealed results, snapshot hash, claim key, integrator and book revision to equal its stored copies. Time-based decisions, including the purchase deadline, the refund window and claim unlock, use the chain's `clock` sysvar rather than the server clock.

15.8 Bounded worker scans

The worker runs as its own service, and exactly one runs per database: at start it takes the PostgreSQL session advisory lock `sway:single-worker` and exits if another worker holds it. Its work per tick is bounded:

- Every `WORKER_RECONCILE_INTERVAL_MILLISECONDS`, reconciliation first pages the relay outbox, then reads the newest finalized signature of the Sway program. The exchange, settlement, campaign, cashback and dispute scans run only when that signature has changed since the previous tick, so a quiet program triggers no domain scans.
- Each scan reads at most `WORKER_SCAN_BATCH_SIZE` rows, ordered by `reconcileAfter` and `id`, and selects only unfinished rows: submitted operations, pending ingests, pending intents and open conversions.
- After a visit, a row's `reconcileAfter` moves forward by `WORKER_RECOVERY_BACKOFF_MILLISECONDS` whether the visit succeeded or not, so an unavailable row rotates to the back instead of holding the page.
- The job runner executes at most `WORKER_JOBS_PER_TICK` jobs every `WORKER_POLL_MILLISECONDS`. Neither loop starts a tick while its previous tick runs, and both pause while the configured Sway or Arcium program is not deployed.

- A failing job is retried after `WORKER_POLL_MILLISECONDS × min(attempts, 10)` and marked failed after `WORKER_MAX_ATTEMPTS`. Only allowlisted error codes are stored, never exception text or remote payloads.

All of these budgets are required configuration and have no built-in defaults.

15.9 HTTP boundary

CORS. `CORS_ORIGINS` lists exact HTTP(S) origins. A request whose `Origin` header is not on the list is answered `403 ORIGIN_NOT_ALLOWED` before routing, validation or rate limiting. Allowed methods are GET, POST, DELETE and OPTIONS, and allowed request headers include Content-Type, Authorization and Idempotency-Key. Every response carries `cache-control: no-store`.

Input. Bodies are limited to 16,384 bytes. Every transport object sets `additionalProperties: false`, and validation neither coerces types nor strips fields, so a stray plaintext field is rejected rather than ignored. Validation errors name only declared field paths, with fixed messages.

Rate limits. Buckets are fixed windows kept by a single atomic PostgreSQL upsert and keyed by the SHA-256 of the bucket name, so every API process shares them.

Bucket	Key	Configuration
Authentication, per IP	HMAC-SHA256(IP_HASH_SALT, auth:<ip>)	AUTH_IP_RATE_LIMIT_REQUESTS, AUTH_IP_RATE_LIMIT_WINDOW_SECONDS
Early access, per IP	HMAC-SHA256 with IP_HASH_SALT	EARLY_ACCESS_RATE_LIMIT_REQUESTS, EARLY_ACCESS_RATE_LIMIT_WINDOW_SECONDS
Agent key	agent:<key id>	AGENT_RATE_LIMIT_REQUESTS, AGENT_RATE_LIMIT_WINDOW_SECONDS
Oracle and admin keys	authority:<scheme>:<key>	AUTHORITY_RATE_LIMIT_REQUESTS, AUTHORITY_RATE_LIMIT_WINDOW_SECONDS

A limited request receives `429 RATE_LIMITED` with a `Retry-After` header. Raw IP addresses are never stored. The client address is taken from forwarded headers only across `TRUST_PROXY_HOPS` hops whose addresses fall inside `TRUST_PROXY_CIDRS`. Request logs record the route template, status, request id and mode, never bodies, authorization headers or keys.

16. Infrastructure and deployment

Sway runs as four services around two Solana programs and one Arcium MXE: the web app, the API, a single worker, and PostgreSQL, around the financial program `sway_exchange` and its Arcium coordinator `sway_arcium`. This chapter describes how they are laid out, how a deployment is created from nothing, how each service starts, and how the system reads the chain without multiplying RPC traffic as users arrive.

16.1 Service layout

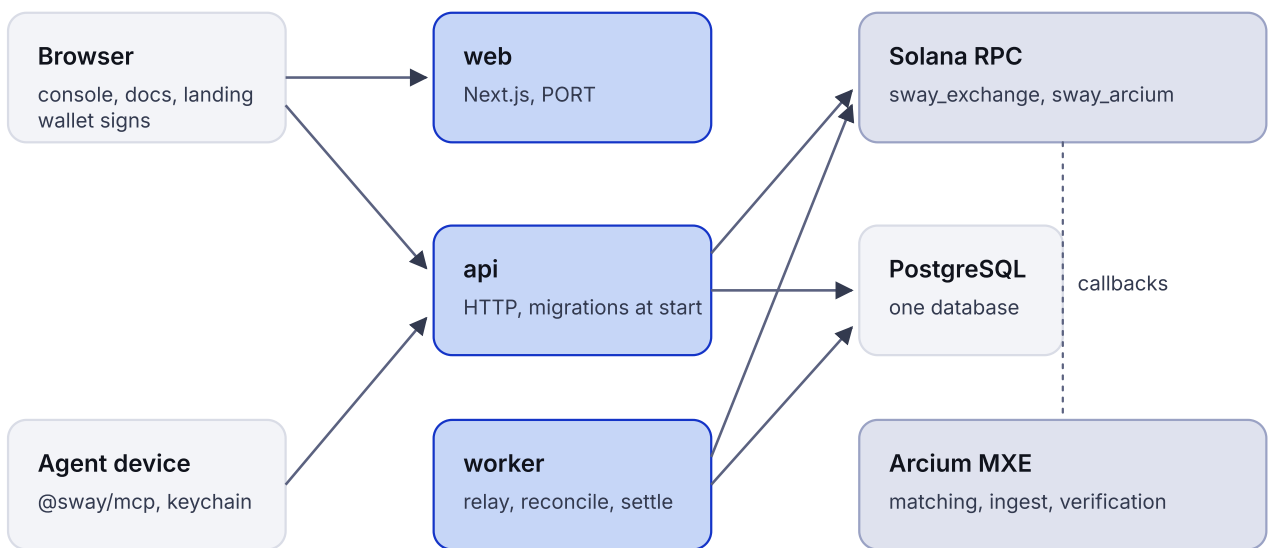


Figure 16.1 — The browser and the agent device talk to the API; only the API and the worker hold keys that act on chain, and only the worker writes background work.

Service	Role	Start	Health	Scale
web	The site, the brand console, the docs and the papers. Server components read the API; the browser calls the API directly.	next start on PORT	GET /	any number of instances
api	Every HTTP operation in Appendix B: sessions, signed requests, prepared transactions, reads.	prisma migrate deploy && node dist/src/main.js	GET /health	any number of instances
worker	The relay outbox, finality tracking, reconcilers, settlement and claim jobs, Arcium callback handling, the SWAY buyback keeper.	node dist/src/worker.js	GET /health	exactly one, held by a PostgreSQL advisory lock
PostgreSQL	The off-chain record: idempotency, sessions, outbox, mirrored chain state, evidence hashes.	managed	managed	one

The API and the worker are built from the same backend source and differ only in their entry point. Migrations run when the API starts, never during the build: the build environment does not reach the private database network, and running them at start keeps the schema and the code that expects it in one step. `prisma generate` runs at build time.

16.2 From nothing to a running deployment

A deployment needs a funded deployer key, an RPC endpoint and an Arcium cluster, and then two commands.

- 1. Programs.** `deploy:program` builds the financial program (with its `arcium-financial` feature), the Anchor coordinator and the Arcis circuits in a pinned Linux toolchain (Docker by default, WSL as an alternative; Rust 1.90.0, Agave 3.1.10, Anchor 1.0.2, Arcium 0.15.0) against the program IDs it is about to deploy. It deploys both programs with fixed buffer keypairs, so an interrupted upload resumes and the buffer's rent returns to the payer, and compares each finalized program account with the local build by SHA-256.
- 2. Protocol accounts and MXE.** The same command initializes `Config`: the admin, the relayer, the treasury, the USDC mint for the cluster, the purchase window, the settlement grace, the SWAY mint from `SWAY_MINT` and the relayer's key as buyback authority. It admits the `running-footwear` category shard, creates its 32-slot order book and the address lookup table the relayed transactions use, then initializes the MXE on the chosen Arcium cluster (offset 456 on devnet, 2026 on mainnet, a recovery set of four) and uploads and initializes each circuit's computation definition. `deploy:init` repeats this step alone.
- 3. Settings out.** It writes every public address and setting the services need into `deploy/<cluster>.env`, which is pasted into the hosting platform's variables, and the coordinator's IDL into `deploy/artifacts/`, which ships with the backend.

4. **Services.** The four services start from the settings above; the API applies migrations on its first start.

The settings include the worker's buyback keeper (§9.8, §13.1):

Setting	Default	Effect
SWAY_MINT	—	The SWAY mint, equal to <code>Config.sway_mint</code> ; the keeper runs while it is set
BUYBACK_AUTHORITY	written by <code>deploy:init</code> : the relayer's public key	Equal to <code>Config.buyback_authority</code> ; the worker signs with <code>RELAYER_KEYPAIR</code>
BUYBACK_MIN_USDC	10000000 (10 USDC)	Treasury balance a buyback needs
BUYBACK_INTERVAL_SECONDS	3600	Shortest time between buybacks
BUYBACK_SLIPPAGE_BPS	100 (1%)	Slippage of the Jupiter quote

Before it sends anything, the `deploy` command prints the SOL the deployer must hold: the program's rent plus the transaction fees, read from the Solana CLI's own balance check against the build being deployed.

16.3 Keys and where they live

Key	Holder	Used for
Deployer / upgrade authority	operator, offline after deploy	program deployment and upgrades
Config admin	operator's wallet	merchant activation and suspension, bond release, integrator approval, dispute resolution and bond debits on the operator page; Config updates
Relayer	the worker's settings; the api holds only its public key	paying fees for relayed instructions (<code>post_intent</code> , <code>record_conversion</code> , <code>claims</code>); signing the buyback as <code>Config.buyback_authority</code>
Oracle signer	worker settings, when cashback is enabled	attesting affiliate commissions (§10)
Merchant wallet	the brand's browser wallet	registration, bond, campaigns, funding, offers
Receipt signer	the brand's store (or the console's browser key)	the 128-byte receipt (§7)
Integrator authority	the agent developer's wallet	integrator registration, agent key registration
Agent key, per-intent keys, claim keys	the agent device's keychain	signed agent requests, intent encryption and result decryption, claims (§8)

Only public keys appear in the database. Service secrets are environment variables of the service that uses them, never shared variables.

16.4 Reading the chain economically

Every reader of the chain goes through one connection per process whose fetch middleware records each RPC call as a single structured log line with the method name; the keyed RPC URL never reaches a log.

- **Shared facts are cached per process.** The cluster's genesis hash is read once; whether the program is deployed is cached for 30 seconds with one read in flight; failures are not cached. Every API instance answers every browser tab from the same cached facts, so traffic from tabs does not multiply calls.
- **The worker scans only on change.** A reconcile pass first asks for the program's newest signature. If it is the same as at the last full read, nothing the program owns has changed and the pass only advances its clock; otherwise it reads the accounts it needs at a context slot at least as new as that signature.

- **Per-wallet reads follow the wallet.** The console reads a wallet's merchant, campaigns and conversions while the tab is visible and once right after the wallet's own finalized transaction.

16.5 Waiting for the program

Services can be started before the program is deployed. Until the program is found on the configured RPC, GET /health answers 200 {"status":"waiting"}, and chain routes answer one error code that the console shows as a waiting state. As soon as the program appears, the cached check turns over within 30 seconds and the same services serve the live protocol without a restart.

16.6 Upgrades

The deployer key is the upgrade authority of both programs: the deploy command passes it as --upgrade-authority for sway_exchange and the Arcium coordinator and confirms that each ProgramData names it, and the services check the same key as PROGRAM_UPGRADE_AUTHORITY (§12.7). Account layouts grow only into reserved space, and readers (API, worker, web) ship before a program upgrade that starts writing a new field, so no reader meets data it cannot decode.

17. Appendices

The appendices are reference material for implementers. Appendix A gives the exact byte layout of every message Sway signs, hashes or encrypts. Appendix B lists every REST operation in the contract, grouped by the role that calls it. Appendix C lists the API error codes, the program's error codes and the domain status codes. Appendix D defines the product's nouns.

A. Byte layouts

A.1 Conventions

Item	Encoding
Integers	Little-endian. Amounts are u64 USDC base units (6 decimals, 1 USDC = 1,000,000). Timestamps are i64 Unix seconds.
Public keys	32 raw bytes on chain and in signed messages; base58 in JSON.
Binary in JSON	Standard padded base64. 16 bytes are 24 characters, 32 bytes 44, 64 bytes 88 and 128 bytes 172.
Hashes in JSON	64 lowercase hex characters (HashSchema).
Amounts in JSON	Decimal strings of u64 base units, never floating point (AmountSchema).
Canonical JSON	Object keys sorted by code unit, no whitespace, JSON values only (canonicalJson, canonicalizeJson).
Discriminators	Instruction data starts with the first 8 bytes of SHA-256(global:<instruction>); account data starts with the first 8 bytes of SHA-256(account:<AccountName>).

A.2 T2 receipt (SWAYRCP1, 128 bytes)

Signed with Ed25519 by the merchant's receipt signer for the epoch named in the message.

Encoded by `encodeReceiptMessage` and `receipt_message`, decoded by `decodeReceiptMessage` and `Receipt::decode`.

Offset	Size	Field	Encoding	Checked by <code>record_conversion</code>
0	8	Domain	ASCII SWAYRCP1	Equals <code>RECEIPT_DOMAIN</code>
8	32	Intent account	Public key	Equals the intent account passed in
40	1	Slot	u8, 0–31	Equals the <code>slot</code> argument; active in the captured <code>IntentBook</code>
41	32	Order hash	32 bytes	Seeds the <code>["order", merchant, order_hash]</code> guard
73	8	Amount	u64 base units	Purchase price; <code>verify_conversion</code> compares it with the sealed price
81	8	Timestamp	i64 Unix seconds	$\text{created_at} \leq \text{ts} < \text{purchase_deadline}$, $\text{ts} \leq \text{now}$, inside the signer epoch
89	4	Signer epoch	u32	Seeds <code>["signer_epoch", merchant, epoch]</code> ; equals that account's epoch
93	32	Program id	Public key	Equals <code>crate::ID</code>
125	3	Reserved	Zero	Must be zero

The receipt hash is SHA-256 over the 128 bytes. It is stored as `Conversion.receipt_hash` and returned as `receiptHash` (hex).

A.3 Refund notice (SWAYREF1, 128 bytes)

Signed by the merchant's current receipt signer. The program compares the notice byte for byte with `refund_message(conversion, receipt_hash, merchant.signer_epoch)`.

Offset	Size	Field	Encoding
0	8	Domain	ASCII SWAYREF1
8	32	Conversion account	Public key
40	32	Receipt hash	SHA-256 of the original 128-byte receipt
72	32	Program id	Public key
104	4	Signer epoch	u32, the merchant's current epoch
108	20	Reserved	Zero

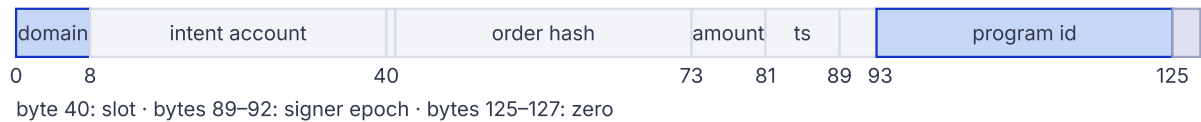
A.4 Claim authorization (SWAYCLM1, 128 bytes)

Signed by the intent's claim key. The program compares it byte for byte with `claim_message(conversion, destination.owner, expiry, nonce)`, reading `expiry` and `nonce` from the

message itself.

Offset	Size	Field	Encoding
0	8	Domain	ASCII SWAYCLM1
8	32	Program id	Public key
40	32	Conversion account	Public key
72	32	Destination wallet	Public key; must own the destination USDC token account
104	8	Expiry	i64 Unix seconds; <code>expiry</code> $\geq$ chain time at execution
112	16	Nonce	Random bytes

SWAYRCP1 receipt, signed by the merchant receipt signer



SWAYREF1 refund notice, signed by the current receipt signer domain conversion account receipt hash program id
zero 0 8 40 72 104 bytes 104–107: signer epoch · bytes 108–127: zero

SWAYCLM1 claim authorization, signed by the per-intent claim key domain program id conversion account
destination wallet expiry nonce 0 8 40 72 104 112 expiry: i64 Unix seconds against the chain clock · nonce: 16 random bytes

Figure 17.1 — The three 128-byte signed messages at a scale of 5 px per byte. Accent fields bind the message to its type and to one program deployment.

A.5 Ed25519 precompile instruction

`record_conversion`, `refund_conversion` and `claim` require this instruction immediately before them. The API builds it with `Ed25519Program.createInstructionWithPublicKey`, giving 240 bytes of data: a 16-byte header, the public key at offset 16, the signature at 48 and the message at 112. `verify_ed25519` accepts it only in this shape:

Offset	Size	Field	Required value
0	1	Number of signatures	1
1	1	Padding	0
2	2	Signature offset	≥ 16 , signature inside the data
4	2	Signature instruction index	0xFFFF (this instruction)
6	2	Public key offset	≥ 16 ; 32 bytes there equal the expected signer
8	2	Public key instruction index	0xFFFF
10	2	Message offset	≥ 16 ; 128 bytes there equal the expected message
12	2	Message size	128
14	2	Message instruction index	0xFFFF

The instruction must reference no accounts, and its program must be the Ed25519 program.

A.6 Instruction data carrying sealed or signed payloads

Offsets include the 8-byte discriminator. The ciphertext arrays have fixed size; `input_count` says how many elements are used. The production build requires exactly 2 elements for an offer and 8 for an intent.

submit_offer, 330 bytes:

Offset	Size	Field
0	8	Discriminator
8	16	Offer id (UUID bytes)
24	1	Slot
25	32	Sender x25519 public key
57	256	8 ciphertext elements of 32 bytes, zero-padded after <code>input_count</code>
313	16	Nonce
329	1	<code>input_count</code> , 1–8

post_intent, 457 bytes:

Offset	Size	Field
0	8	Discriminator
8	16	Intent id (UUID bytes), PDA seed
24	32	Shard hash, SHA-256 of the shard string
56	256	8 ciphertext elements of 32 bytes
312	16	Nonce (A.7)
328	1	<code>input_count</code> , 1–8
329	32	<code>input_x25519</code>
361	32	<code>user_x25519</code> , equal to <code>input_x25519</code>
393	32	Claim public key
425	32	Integrator payout wallet

match_intent_callback, 682 bytes:

Offset	Size	Field
0	8	Discriminator
8	1	Attempt, 1 or 2
9	32	Input commitment (A.11)
41	1	<code>result_count</code> , 1–9
42	288	User copy: 9 elements, sealed to the intent key, zero after <code>result_count</code>
330	288	MXE copy: 9 elements, zero after <code>result_count</code>
618	16	User copy nonce
634	16	MXE copy nonce
650	32	MXE sender x25519 key

The remaining payload-carrying instructions are short:

Instruction	Bytes	Layout after the discriminator
ingest_offer_callback	314	valid (1), 8 elements (256), nonce (16), element_count (1), sender (32)
record_conversion	177	slot (1), receipt (128), rebate u64 (8), snapshot hash (32)
verify_conversion_callback	41	receipt hash (32), valid (1)
refund_conversion	136	refund notice (128)
claim	136	claim authorization (128)
oracle_post_conversion	64	event hash (32), purchase amount u64, commission u64, purchase timestamp i64
request_nonce	24	random nonce (16)

A.7 Encrypted envelope

The agent and the brand send ciphertext in an `EncryptedInput` object. The sender derives a shared secret with x25519 from its secret key and the MXE public key, then encrypts each packed field element with the Rescue cipher keyed by that secret (`encryptFields`).

Field	Content
elements	1–8 strings, each 32 bytes of Rescue ciphertext in base64
nonceBase64	16 bytes
senderPublicKeyBase64	32-byte x25519 public key, canonical (non-zero, below $2^{255} - 19$)
encoding	rescue-32-v1

The intent nonce has a fixed format, announced as `inputNonceFormat`: "expiry-u64le-random8". Offer and result nonces are 16 opaque random bytes.

Nonce byte	Content
0–7	Purchase deadline, u64 Unix seconds; $\text{now} < \text{deadline} \leq \text{now} + \text{purchase_window_secs}$ at <code>post_intent</code>
8–15	Random bytes

A sealed result (`SealedResult`) has the same fields without the sender key, which is the MXE key, and holds 1–9 elements.

A.8 Intent plaintext (SwayIntentV1, 8 elements)

INTENT_FIELDS lists 23 unsigned integers. groupedPacker packs the first two into one element and every following group of three into one element with the Arcium createPacker descriptor, and canonicalUnpack requires the exact repacking on decode.

Element	Fields (bits)	Bits used
0	metadata (128), category (64)	192
1	maxPrice, brandIds[0], brandIds[1] (64 each)	192
2	brandIds[2], brandIds[3], brandIds[4]	192
3	brandIds[5], brandIds[6], brandIds[7]	192
4	brandIds[8], brandIds[9], bonuses[0]	192
5	bonuses[1], bonuses[2], bonuses[3]	192
6	bonuses[4], bonuses[5], bonuses[6]	192
7	bonuses[7], bonuses[8], bonuses[9]	192

category is the category identifier hash (A.11). maxPrice and bonuses are u64 base units. Brand preferences are sorted by brand identifier string, at most 10 (MAX_BRAND_PREFERENCES); brandIds[i] is the brand identifier hash, and unused entries are zero. A brand appears when it carries a non-zero bonus or is one the shopper already buys.

metadata bits	Width	Field
0–63	64	INTENT_TAG = 0x53574159494E5431 (ASCII SWAYINT1)
64–79	16	Size × 2, in half-size steps
80–81	2	Width: 0 narrow, 1 standard, 2 wide
82	1	Region flag, 1 = US
83–92	10	Existing-brand flags; bit 83 + i marks preference i as a brand the shopper already buys
93–96	4	Preference count, 0–10
97–127	31	Zero

A.9 Offer plaintext (SwayOfferV1, 2 elements)

Element	Fields (bits)	Bits used
0	metadata (128), category (64)	192
1	brand, price, rebate (64 each)	192

metadata bits	Width	Field
0–63	64	OFFER_TAG = 0x535741594F464631 (ASCII SWAYOFF1)
64–79	16	Minimum size × 2
80–95	16	Maximum size × 2, not below the minimum
96–98	3	Width mask: bit 96 narrow, 97 standard, 98 wide; non-zero
99	1	Region flag, 1 = US
100	1	New-to-brand only
101–127	27	Zero

ingest_offer accepts the rule only when the metadata is valid, $\text{rebate} \leq \text{cap}$, and brand and category equal the expected brand and category hashes supplied with the computation. The MXE then re-encrypts the same elements to itself for the book slot.

A.10 Top-three result (SwayTopThreeV2, 6 elements)

Element	Fields (bits)	Bits used
0	metadata (128), slots[0..2] (8 each)	152
1	prices[0..2] (64 each)	192
2	rebates[0..2]	192
3	campaigns[0..2], campaign identifier hashes	192
4	brands[0..2], brand identifier hashes	192
5	category, intent (64 each)	128

metadata holds RESULT_TAG = 0x53574159544F5032 (ASCII SWAYTOP2) in bits 0–63 and the number of filled ranks, 0–3, from bit 64. Filled ranks come first, ordered by score with the lower slot winning ties. An empty rank has slot 255 and zero price, rebate, campaign and brand. intent is the identifier hash of the intent account address, and category that of the category shard. match_intent writes

two copies: one sealed to the intent's x25519 key and one encrypted to the MXE for `verify_conversion`.

A.11 Hash constructions

Hash	Construction	Used as
Identifier hash	First 8 bytes, little-endian, of SHA-256(<code>sway/<domain>/v1</code> $\parallel$ <code>0x00</code> $\parallel$ value); domain is brand, category, campaign or intent; value at most 256 characters; zero is reserved. The production program computes the same hash on chain (<code>identifier_hash</code>) for the merchant wallet, campaign UUID and shard text	Packed category, brandIds, brand, campaigns, intent fields
Shard hash	SHA-256(shard string)	OfferBook seed [<code>"book"</code> , <code>shard_hash</code>]; <code>Intent.shard</code>
Input commitment	SHA-256(<code>sway/intent-input/v1</code> $\parallel$ <code>0</code> $\parallel$ intent $\parallel$ SHA-256(book account data) $\parallel$ <code>input_x25519</code> $\parallel$ nonce $\parallel$ <code>input_count</code> $\parallel$ ciphertext elements $\parallel$ claim key $\parallel$ integrator)	<code>Intent.input_commitment</code>
Snapshot hash	SHA-256(<code>sway/mxe-snapshot/v1</code> $\parallel$ <code>0</code> $\parallel$ <code>result_count</code> $\parallel$ MXE nonce $\parallel$ MXE elements)	<code>Intent.snapshot_hash</code> , <code>Conversion.snapshot_hash</code>
Receipt hash	SHA-256(128-byte receipt)	<code>Conversion.receipt_hash</code> , refund bytes 40–71
Affiliate event hash	SHA-256(<code>sway/affiliate-event/v1</code> $\parallel$ <code>0</code> $\parallel$ canonical JSON [<code>network</code> , <code>networkEventId</code>])	[<code>"oracle_event"</code> , <code>event_hash</code>]; T3 receipt_hash and order_hash
Dispute case hash	SHA-256(canonical JSON of amount, disputeId, evidenceHash, merchantId, outcome, payoutAddress, resolutionHash)	[<code>"bond_resolution"</code> , merchant, case_hash]
Request digest	SHA-256(canonical JSON), hex	Challenge Request-SHA256, Body-SHA256, idempotency request hash

A.12 Text messages

The wallet challenge is the nine-line message shown in §15.3. Its nonce is 32 random bytes in base64url (43 characters), and `Resources-SHA256` is the request digest of the sorted resource id array.

A signed request is nine UTF-8 lines joined by `\n` and signed with Ed25519:

Line	Content
1	Sway signed request v1
2	Audience: <service origin>
3	Principal: <SwayAgent, SwayIntegrator, SwayOracle or SwayAdmin>
4	Key: <agent key id, integrator id, or base58 key for oracle and admin>
5	Timestamp: <Unix seconds>
6	Nonce: <16-128 characters of A-Z, a-z, 0-9, _ and ->
7	Method: <method in upper case>
8	Path: <request path and query>
9	Body-SHA256: <request digest of the body, {} when empty>

It travels as Authorization: <Scheme> <keyId>:<timestamp>:<nonce>:<signatureBase64>, with a 1–12 digit timestamp.

An affiliate postback attestation signs Sway affiliate postback v1\n followed by the canonical JSON of the postback with network added and attestation removed.

B. API reference

B.1 Conventions

Every operation is defined once in `operations.ts`; the server routes, the OpenAPI document, the TypeScript client and the MCP tools all derive from that table. A success returns `{ data, meta }`; a failure returns `{ error, meta }`, where `error` is `{ code, message, retryable, details? }` and `meta` is `{ mode, scenario, requestId }`. Operations marked idempotent require an Idempotency-Key header of 1–128 characters (§15.6). List operations take `cursor` and `limit` (1–100) and return `Page<T> = { items, nextCursor, sync }`, where `sync` reports ready, indexing or failed and the data source.

Role	Credential
public	None. Registration and sign-in carry a challenge proof in the body.
merchant	Authorization: Bearer <session token> from verifyMerchantSession
receipt	No bearer secret. The body carries an Ed25519 signature by the merchant's registered receipt signer, so anyone may relay it.
integrator	SwayIntegrator signed request by the integrator authority
agent	SwayAgent signed request by an agent key holding the listed scope
claim	authorization field in the body: a challenge proof signed by the intent's claim key
oracle	SwayOracle signed request by a configured oracle key
admin	SwayAdmin signed request by the configured Config admin address (ADMIN_MULTISIG_ADDRESS)

In the tables below, "Idem." marks idempotent operations, and object names are the contract schema names without the Schema suffix.

B.2 Public

Method and path	Idem.	Request → response	Purpose
GET /v1/encryption		— → encryption parameters	MXE x25519 public key, adapter identity, rescue-32-v1, element budget 8, signed-request audience, chain clock, purchase window, settlement grace, allowed shards and nonce format
POST /v1/early-access		EarlyAccessRequest → EarlyAccessResult	Request early access as a brand or agent developer
POST /v1/auth/challenges		ChallengeRequest → Challenge	Issue a one-use challenge bound to purpose, key, resources and request hash
POST /v1/auth/merchant/session	Yes	SessionVerifyRequest → Session	Sign in with the merchant wallet; issue a session token
POST /v1/integrators	Yes	IntegratorRegistrationRequest → Integrator	Register an integrator authority and USDC payout wallet
GET /v1/cashback/rates		PageQuery + shard → Page<CashbackRate>	List the public cashback rates, filterable by shard
GET /v1/protocol/buyback		— → Buyback	The SWAY mint, the treasury's finalized USDC balance, and the USDC spent and SWAY burned by finalized buybacks (\$9.8)

B.3 Merchant (brand console)

Method and path	Idem.	Request → response	Purpose
GET /v1/auth/merchant/session		— → Session	Read the session and its wallet
DELETE /v1/auth/merchant/session	Yes	— → Session	End the session
GET /v1/merchant		— → Merchant	Profile, receipt signers, bond and status
POST /v1/merchant	Yes	MerchantRegistrationRequest → Transaction	Prepare register_merchant
POST /v1/merchant/receipt-signers	Yes	RotateSignersRequest → Transaction	Prepare rotate_receipt_signer
POST /v1/merchant/bond	Yes	AmountRequest → Transaction	Prepare a USDC bond deposit
GET /v1/campaigns		PageQuery → Page<Campaign>	List the merchant's campaigns
GET /v1/campaigns/:id		— → Campaign	Campaign with chain-indexed balances
POST /v1/campaigns	Yes	CreateCampaignRequest → Transaction	Prepare create_campaign with cap and lock checks
POST /v1/campaigns/:id/fund	Yes	AmountRequest → Transaction	Prepare a vault deposit within the campaign cap
POST /v1/campaigns/:id/withdraw	Yes	AmountRequest → Transaction	Prepare withdrawal of unreserved USDC
GET /v1/campaigns/:id/spend		SpendQuery → SpendSeries	Daily rebate and fee debits, inclusive UTC dates
POST /v1/transactions/:id/submit	Yes	SubmitTransactionRequest → Transaction	Validate, simulate and broadcast a wallet-signed prepared transaction
POST /v1/transactions/:id/report	Yes	ReportTransactionRequest → Transaction	Track a transaction the wallet sent itself, by signature
GET /v1/transactions/:id		— → Transaction	Follow prepared, submitted, confirmed, failed or expired
GET /v1/offers		PageQuery + campaignId → Page<Offer>	List encrypted offers with slot and ingest state
GET /v1/offers/:id		— → Offer	One offer's ingest result

Method and path	Idem.	Request → response	Purpose
POST /v1/offers	Yes	SubmitOfferRequest → SubmitOfferResponse	Reserve a slot and prepare submit_offer for a browser-encrypted rule
GET /v1/conversions		ConversionQuery → Page<Conversion>	Conversions on the merchant's campaigns
GET /v1/conversions/:id		— → Conversion	Lock time, fees and refund availability

B.4 Receipt signer

Method and path	Idem.	Request → response	Purpose
POST /v1/receipts	Yes	SignedReceiptRequest → Conversion	Relay a 128-byte T2 receipt for verification and escrow
POST /v1/conversions/:id/refund	Yes	SignedRefundRequest → Conversion	Relay a signed refund notice inside the lock window

B.5 Integrator

Method and path	Idem.	Request → response	Purpose
GET /v1/integrator		— → Integrator	The integrator and its approval or revocation state
GET /v1/integrator/agent-keys		PageQuery → Page<AgentKey>	List agent keys
POST /v1/integrator/agent-keys	Yes	RegisterAgentKeyRequest → AgentKey	Register a scoped agent public key
DELETE /v1/integrator/agent-keys/:id	Yes	— → AgentKey	Revoke an agent key

B.6 Agent

Method and path	Scope	Idem.	Request → response	Purpose
POST /v1/intents	intent:write	Yes	PostIntentRequest → Intent	Queue matching for an encrypted intent
GET /v1/intents/:id	intent:read		— → Intent	Poll status and the sealed top three
GET /v1/intents/:id/book	intent:read		— → IntentBookMetadata	The public book map captured for this intent
POST /v1/intents/:id/retry	intent:write	Yes	— → Intent	Retry an expired computation once
POST /v1/intents/:id/voucher	intent:write	Yes	VoucherRequest → Voucher	Issue the intent and slot voucher for checkout
POST /v1/cashback/tracking-links	cashback:link	Yes	TrackingLinkRequest → TrackingLink	Affiliate link with the intent id as sub-ID
POST /v1/purchases	purchase:report	Yes	ReportPurchaseRequest → PurchaseReport	Report an order digest; the receipt stays on the device

B.7 Claim key

Claim-key reads use POST so that proofs never appear in query strings.

Method and path	Challenge purpose	Idem.	Request → response	Purpose
POST /v1/claims/query	claim_read		ClaimablesRequest → Page<Claimable>	Amounts, unlock times and states
POST /v1/claims	claim	Yes	ClaimRequest → ClaimResult	Relay a destination-bound claim for one unlocked conversion
POST /v1/claims/:id/query	claim_read		SignedRead → ClaimResult	Claim relay and transaction status
POST /v1/disputes	open_dispute	Yes	OpenDisputeRequest → Dispute	Open a dispute with an evidence digest
POST /v1/disputes/:id/query	claim_read		SignedRead → Dispute	Read a dispute

SignedRead is { claimPublicKey, authorization }.

B.8 Affiliate oracle

Method and path	Idem.	Request → response	Purpose
POST /v1/affiliate/postbacks	Yes	AffiliatePostbackRequest → AffiliateEvent	Ingest an attested commission event once per network event id
GET /v1/affiliate/events		PageQuery → Page<AffiliateEvent>	Pending, validated, reversed, paid and batched events
POST /v1/oracle/batches	Yes	OracleBatchRequest → OracleBatch	Post up to 32 paid events as T3 conversions
GET /v1/oracle/batches/:id		— → OracleBatch	Batch status and per-event outcomes

B.9 Admin

Method and path	Idem.	Request → response	Purpose
GET /v1/admin/merchants/pending		PageQuery → Page<Merchant>	Merchants awaiting activation
GET /v1/admin/merchants		PageQuery + status (active or suspended) → Page<Merchant>	Active and suspended merchants
GET /v1/admin/integrators/pending		PageQuery → Page<Integrator>	Integrators awaiting approval
GET /v1/admin/disputes		PageQuery → Page<AdminDispute>	Open and under-review disputes with brand, evidence hash and compensation address
POST /v1/admin/merchants/:id/activate	Yes	— → Transaction	Prepare activate_merchant for the Config admin's wallet
POST /v1/admin/merchants/:id/suspend	Yes	— → Transaction	Prepare suspend_merchant for the Config admin's wallet
POST /v1/admin/merchants/:id/bond/release	Yes	— → Transaction	Prepare release_bond of the whole bond to the merchant wallet's USDC account; BOND_LOCKED while the program would refuse
POST /v1/admin/integrators/:id/approve	Yes	— → Integrator	Approve an integrator for the relayer
POST /v1/admin/disputes/:id/resolve	Yes	ResolveDisputeRequest {outcome, bondDebit, resolutionHash} → ResolveDisputeResult	Resolve a dispute; a user outcome returns the dispute with a prepared slash_bond for the Config admin's wallet

B.10 Service documents

Path	Content
GET /health	Service status and capabilities
GET /openapi.json	OpenAPI document generated from the contract
GET /v1/protocol	PROTOCOL_PARAMETERS (§3)
GET /v1/mcp/tools	MCP tool definitions: <code>find_offers</code> , <code>get_voucher</code> , <code>report_purchase</code> , <code>claimable</code> , <code>claim</code>

C. Error codes

C.1 API error codes

`retryable` is part of every error object. Statuses are the ones the live service returns; where two are listed, the route decides which one applies.

Code	Status	Meaning
VALIDATION_ERROR	400	The request does not match the contract; <code>details.fields</code> names declared fields only
UNAUTHORIZED	401	Missing, malformed, expired or revoked session, signed request, agent key or integrator
WRONG_WALLET	403	The signed-in wallet does not own the merchant record or its chain account
FORBIDDEN	403, 409	Missing scope, unapproved integrator, unconfigured authority key, or an action the resource state does not allow
NOT_FOUND	404	The resource does not exist for this principal; foreign ids are indistinguishable from absent ones
RATE_LIMITED	429	A rate-limit bucket is exhausted; retryable, with <code>Retry-After</code>
BOND_LOCKED	409	The merchant bond cannot be released while the program retains an outstanding liability: the merchant is not suspended, a receipt is pending, or <code>liability_until</code> has not passed
IDEMPOTENCY_REQUIRED	400	The operation needs an Idempotency-Key of 1–128 characters
IDEMPOTENCY_CONFLICT	409	The key, intent id, event id or operation already commits different terms
REPLAY_DETECTED	409	Consumed challenge, reused request nonce, reused encrypted input or claim key, or an earlier operation still awaiting finality
CONFIGURATION_REQUIRED	503	A deployment policy, key or adapter that the operation needs is not configured
INSUFFICIENT_FUNDS	409	The chain account lacks available funds, or a withdrawal exceeds unreserved vault funds
CAP_EXCEEDED	409, 422	A per-conversion, campaign or daily cap would be exceeded, or the book has no free slot
REFUND_WINDOW_CLOSED	409	The conversion is not escrowed or its lock has ended
DISPUTE_WINDOW_CLOSED	409	The configured dispute window has passed
CLAIM_LOCKED	409	The conversion is still locked or not escrowed
ALREADY_CLAIMED	409	The conversion has been claimed
INVALID_SIGNATURE	401	A signature, proof or binding check failed, including a receipt that does not match the sealed top three

Code	Status	Meaning
CHALLENGE_EXPIRED	401	Expired challenge, signed-request timestamp outside the window, or expired claim authorization
COMPUTATION_FAILED	409	An MPC computation or chain operation ended without the needed result, or the settlement grace ended for an unrecorded receipt
PROGRAM_NOT_DEPLOYED	503	The configured Sway or Arcium program is not available on the configured cluster
TRANSACTION_SIMULATION_FAILED	409	The signed wallet transaction failed simulation; prepare it again
TRANSACTION_FAILED	in <code>Transaction.failure</code>	A wallet-sent transaction landed with an error
DEPENDENCY_UNAVAILABLE	503	Database, RPC, resource lock or pending finality; retryable with the same request
ADDRESS_BLOCKED	403	The destination address is refused
ORIGIN_NOT_ALLOWED	403	The <code>Origin</code> header is not in <code>CORS_ORIGINS</code>
INTERNAL_ERROR	500	Unexpected failure; retryable

C.2 Program error codes

Anchor numbers custom errors from 6000 in declaration order (`ExchangeError`).

Code	Name	Meaning
6000	ConfigurationRequired	A required policy is unset or unsupported
6001	Unauthorized	The authority is not allowed, or a key is the default key
6002	WrongMint	The mint is not the pinned plain SPL USDC mint with 6 decimals
6003	InvalidAmount	An amount must be positive
6004	Overflow	Checked arithmetic overflowed
6005	CapExceeded	A rebate, campaign or daily cap is exceeded
6006	Underfunded	A campaign vault or merchant bond is underfunded
6007	LockTooShort	lock_days is below the merchant's return window
6008	InvalidInput	An input is outside protocol bounds
6009	NotExpired	The computation has not expired, so it cannot be retried
6010	StaleCallback	The callback is stale or its state was consumed
6011	InvalidSignature	The preceding Ed25519 precompile is missing or invalid, or a claim authorization is wrong or expired
6012	InvalidReceipt	A receipt or refund message is not bound to these accounts
6013	RefundWindowClosed	The refund window has closed or the conversion is not escrowed
6014	AlreadyConsumed	The conversion is already claimed, refunded, failed or unverified
6015	ClaimLocked	The claim is still locked
6016	InputKeyMismatch	The input owner must equal the result recipient
6017	SnapshotMismatch	The authenticated sealed snapshot or input commitment does not match the intent
6018	CampaignMismatch	The book slot is bound to another campaign
6019	IntentStillLive	The intent still has live computation or conversion liabilities
6020	BondLocked	The merchant bond still backs unresolved liabilities
6021	PurchaseWindowClosed	The purchase window has expired or is outside configured bounds
6022	SettlementWindowClosed	The intent's settlement grace has ended
6023	BuybackIncomplete	The buyback must spend the treasury debit and burn the minimum output in the same transaction, closed by buyback_finish

C.3 Domain status codes

Field	Values
Offer.rejectionCode	REBATE_OVER_CAP (the rule's rebate exceeds the campaign cap), COMPUTATION_FAILED
Intent.fallbackReason	COMPUTATION_EXPIRED_TWICE, NO_ELIGIBLE_OFFERS, CASHBACK_ONLY
OpenDisputeRequest.reason	MISSING_CONVERSION, INCORRECT_AMOUNT, UNEXPECTED_REFUND
Conversion.status	verification_pending, escrowed, locked, refunded, claimable, claimed, disputed, failed
Dispute.status	open, under_review, resolved_user, resolved_merchant, rejected

D. Glossary

Term	Meaning
Active mask	32-bit mask of a book's slots that take part in matching; captured into each intent's IntentBook
Admin	The wallet or multisig that holds Config.admin; the deployer hands it over at bootstrap and keeps the upgrade authority
Affiliate oracle	Sway's attesting key for network-validated commissions; posts T3 conversions
Agent	Software that shops for a user: it encrypts intents, decrypts results, buys and claims
Agent key	Ed25519 key registered under an integrator with scopes intent:write, intent:read, purchase:report OR cashback:link
Base units	USDC's smallest unit; 1 USDC = 1,000,000 base units
Basis points (bps)	Hundredths of a percent; 10,000 bps = 100%
Bond	A merchant's USDC deposit at ["merchant_bond", merchant], required for activation and drawn on in disputes
Book	Per-shard OfferBook account of 32 slots holding MXE-encrypted offers
Brand	The seller whose campaign funds rebates; on chain it is a merchant
Brand bonus	An amount the shopper adds to an offer's value for a preferred brand
Buyback	The one transaction that spends treasury USDC: buyback_begin, a Jupiter swap to SWAY, a burn and buyback_finish, signed by Config.buyback_authority (\$9.8)
Campaign	A brand's budget unit: vault, per-conversion cap (at most 50 USDC), daily cap, lock days and funding up to 5,000 USDC
Cashback	The fallback that pays from affiliate commissions, split 70% user, 10% integrator, 20% Sway
Cashback rate	A public table entry: merchant domain, shard, commission bps and the fixed split
Challenge	A one-use message the API issues for a wallet or claim key to sign (\$15.3)
Chain operation	A durable outbox record of one transaction: terms hash, signed bytes, signature, status
Claim	The transfer of an unlocked rebate to a destination chosen by the claim key
Claim key	A fresh Ed25519 key per intent, held on the agent device, that authorizes reads, claims and disputes
Claimable	The API view of a conversion as seen by its claim key
Claims vault	Config-owned USDC account that escrows rebates and fees until claim or refund
Computation	One queued MPC run; it expires after 180 slots and an intent may retry once

Term	Meaning
Config	The program's singleton account: admin, mint, MXE, claims vault, treasury, policies, SWAY mint and buyback authority
Conversion	The on-chain record of a verified purchase: rebate, fees, unlock time and status
Dispute	A claim-key-authorized case against a merchant, resolved by the admin with an optional bond debit
Escrow	The holding of rebate and fees in the claims vault between verification and claim
Integrator	The agent developer; registers an authority key and payout wallet and earns 5% of the rebate (500 bps) on brand-direct conversions and 10% of the commission (1,000 bps) on cashback
Intent	An encrypted purchase request: size, width, region, maximum price and up to 10 brand preferences, with a public shard
Intent book	The IntentBook account holding the campaign map and active mask captured when the intent was posted
Lock	The period from verification to <code>unlock_at</code> (<code>lock days × 86,400 s</code>), at least the merchant's return window
Merchant	The on-chain identity of a brand: wallet, receipt signer, bond, return window and status
MXE	The Arcium MPC execution environment that runs <code>ingest_offer</code> , <code>match_intent</code> and <code>verify_conversion</code>
Offer	A brand's sealed rule: category, brand, size range, widths, region, new-to-brand flag, price and rebate
Order guard	The <code>["order", merchant, order_hash]</code> account that lets one merchant order convert once
Proof tier	Evidence class of a conversion: T1 on-chain payment (not in v0), T2 merchant receipt, T3 affiliate commission
Purchase deadline	The time, carried in the intent nonce, by which the purchase must happen
Receipt	The 128-byte <code>SWAYRCP1</code> message a merchant signs for a purchase
Receipt signer	The brand's Ed25519 key that signs receipts and refund notices, versioned by signer epoch
Refund notice	The 128-byte <code>SWAYREF1</code> message that reverses an escrowed conversion inside its lock
Relayer	The Sway service key that pays fees and submits transactions for agents, receipts and claims
Score	<code>brand_bonus + rebate - price</code> , the value of an offer to the shopper
Sealed	Encrypted so that only the intended key holder (the intent key or the MXE) can read it
Settlement deadline	Purchase deadline plus <code>settlement_grace_secs</code> ; the last time a conversion may be recorded

Term	Meaning
Shard	The public category label, such as <code>running-footwear</code> , that selects a book
Signed request	An HTTP request signed per call by an agent, integrator, oracle or admin key (A.12)
Signer epoch	A numbered <code>ReceiptSignerEpoch</code> account binding one receipt key to a validity interval
Slot	Position 0–31 in a book, bound to one campaign while active
Top three	Up to three highest-scoring eligible offers, sealed to the intent's <code>x25519</code> key
SWAY	Sway's token: 1,000,000,000 fixed supply, sold entirely in the public launch; the treasury's fees buy it and burn it (\$9.8)
Tracking link	An affiliate link whose sub-ID is the intent id
Treasury	Config-owned USDC account that receives the Sway fee and is spent only by the buyback
Vault	A campaign's USDC token account at <code>["campaign_vault", campaign]</code>
Vault floor	Minimum vault balance for a campaign's slot to take part in matching
Voucher	The intent id, intent account address and chosen slot; the checkout string is <code>sway:<intent id>:<slot></code>