| | | | |
|---|---|---|---|
| Workgroup: | Network Working Group | | |
| Internet-Draft: | draft-solana-session-00 | | |
| Published: | 9 September 2026 | | |
| Intended Status: | Informational | | |
| Expires: | 13 March 2027 | | |
| Authors: | L. Galabru<br>*Solana Foundation* | Desormeaux<br>*Solana Foundation* | M. Assaf<br>*Moonsong Labs* |

# Solana Session Intent for HTTP Payment Authentication

## Abstract

This document defines the "solana" payment method implementation of the "session" intent registered by [I-D.payment-intent-session], for use within the Payment HTTP Authentication Scheme [I-D.ryan-httpauth-payment-01]. Sessions enable metered, streaming, or repeated-use access to resources through off-chain vouchers backed by an on-chain escrow. The client opens a payment channel by depositing into a channel program. The client either authorizes incremental spend directly with signed vouchers or presents a reusable bearer proof while the operator signs the corresponding vouchers. The channel settles on-chain when the session closes.

## Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 13 March 2027.

## Copyright Notice

## Table of Contents

# 1. Introduction

HTTP Payment Authentication [I-D.ryan-httpauth-payment-01] defines a challenge-response mechanism that gates access to resources behind payments. The "session" intent and its shared semantics — lifecycle operations, accounting invariants, request fields, and receipt shape — are registered and defined by [I-D.payment-intent-session]. This document defines how the "solana" payment method implements that intent.

The `session` intent establishes a unidirectional streaming payment channel using on-chain escrow and off-chain signed vouchers. This enables high-frequency, low-cost payments by batching many off-chain voucher updates into periodic on-chain settlement.

Unlike the `charge` intent, which settles a full on-chain transaction per request, the `session` intent allows clients to pay incrementally as service is consumed. This makes sessions suitable for streaming, metered APIs, and any use case where per-request on-chain settlement would be prohibitively expensive or slow.

## 1.1. Solana-Specific Capabilities

This specification leverages Solana-specific capabilities:

- **Escrow via channel program**: Deposits are held by an on-chain program (not the server), enabling trustless settlement and client-initiated forced close.
- **Atomic multi-instruction transactions**: Channel open can include the channel-PDA creation, escrow ATA creation, deposit transfer, and splits commitment in a single transaction. Similarly, cooperative close can bundle `settleAndSeal` and `distribute` so the merchant payout, payer refund, treasury sweep, and escrow-ATA closure all land atomically and immediately — no token movement is ever slot-gated (see Section 6.2.10).
- **Fee payer separation**: The server / operator can sponsor the cooperative on-chain operations it submits (open, topUp, settle, settleAndSeal, distribute, reclaim). The operator funds both the transaction fees AND the channel rent: it acts as the `rentPayer` that funds the

channel PDA and escrow ATA rent at `open` and recovers that SOL rent after close — via the terminal `distribute`'s fast path or a later permissionless `reclaim` (see Section 6.2.10). The client (`payer`) only ever moves stablecoin and never needs SOL during the normal session lifecycle. Because a SOL-free client cannot self-fund escape-route instructions (requestClose, seal, withdrawPayer), those instructions are permissionless or payer-authorized but **MAY** be submitted by the operator or any party; a client that does hold SOL **MAY** also submit them itself.

- **Ed25519 native verification**: Voucher signatures can be verified on-chain using Solana's native `ed25519` program, enabling trustless settlement without reimplementing signature verification in the channel program.
- **Passkey-compatible P256 verification**: Implementations can support delegated voucher signers using Solana's native `secp256r1` verification program, enabling WebAuthn/passkey-backed session authorization without requiring the funding key to sign each voucher.

## 1.2. Session Flow


Steps 6–9 are off-chain: the client signs a voucher authorizing cumulative spend, the server verifies the signature and serves the resource. No on-chain transaction occurs per request.

The diagram shows client-signed vouchers. With operator-signed vouchers, the client presents the proof in Section 8.1 and the operator meters service and signs each cumulative voucher.

Step 11 typically bundles `settleAndSeal` and `distribute` in the same transaction so the merchant payout, payer refund, treasury sweep, and escrow-ATA closure all land atomically and immediately. The channel PDA itself is deallocated in the same instruction when the epoch window of Section 6.2.10 has already elapsed; otherwise it is left `Distributed` and the operator's periodic `reclaim` sweep recovers the PDA rent after the window.

When fee sponsorship is enabled, the server co-signs as fee payer on steps 4 and 11 — the client never needs SOL for transaction fees.

### 1.3. Relationship to the Charge Intent

The "charge" intent (defined separately) handles one-time payments. The "session" intent handles metered, streaming, or repeated-use payments within a single channel. Both intents share the same `solana` method identifier and encoding conventions.

Like a zero-amount charge proof, the session bearer proof signs a domain-separated, challenge-bound message without moving funds. It is a reusable `session` proof and **MUST NOT** be processed as a `charge`.

### 1.4. Voucher Signer

Two voucher-signing modes are defined:

`client` The client controls `authorizedSigner` and signs cumulative vouchers. A valid voucher both authorizes payment and proves possession of the channel's voucher-signing key.

`operator` The operator controls `authorizedSigner`. The client authenticates metered requests with the reusable bearer proof defined in Section 8.1, and the operator signs the resulting cumulative vouchers.

`client` is the default when the challenge omits `voucherSigner`.

## 2. Requirements Language

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

## 3. Terminology

Payment Channel A unidirectional payment relationship between a payer and payee, consisting of an on-chain escrow account managed by a channel program and a sequence of off-chain vouchers. The channel is identified by a unique `channelId`.

Channel Program A Solana program that manages channel escrow accounts. It enforces deposit, settlement, and withdrawal rules. The program address is declared in the challenge so clients can verify they are interacting with the expected program.

Voucher A signed message authorizing a cumulative payment amount for a specific channel. Vouchers are monotonically increasing in amount.

Cumulative Amount The total amount authorized from channel open, not a per-request delta. For example, if the first voucher authorizes 100 and the second authorizes 250, the payee may claim up to 250 total, not 350.

Authorized Signer The key permitted to sign vouchers for a channel. Defaults to the payer unless the channel open binds a delegated signer in channel state.

Grace Period A time window after a client requests forced close, during which the server can still settle outstanding vouchers before funds are returned to the client.

Idle Timeout The maximum period without channel activity before the server initiates cooperative close. Channel activity is defined in Section 14.6.

## 4. Intent Identifier

The intent identifier for this specification is "session". It **MUST** be lowercase.

## 5. Encoding Conventions

This specification uses two distinct encoding regimes:

1. **HTTP envelope canonicalization.** Challenge payloads (`request` auth-param), credential payloads (`Authorization: Payment` header bodies), and receipts use the same encoding as the Solana charge intent: JCS-serialized [RFC8785] JSON, base64url-encoded [RFC4648] without padding.
2. **On-chain signed-payload encoding.** The bytes the channel's `authorizedSigner` signs to authorize spend are produced by Borsh-encoding the on-chain `Voucher` struct (see Section 9.3). These bytes are the exact message verified by Solana's native `ed25519` precompile and read back by the channel program via the Instructions sysvar. Using a fixed-layout binary encoding here removes the need to repack between the HTTP JSON shape and the precompile message, and makes the on-chain verification a single byte-equality check.

JCS produces deterministic JSON bytes for header canonicalization but is unnecessary for the inner signed payload: the on-chain Borsh layout is deterministic by construction.

## 6. Channel Program Interface

The channel program manages escrow accounts and enforces settlement rules. This section defines the logical interface that conforming channel programs **MUST** implement.

### 6.1. Channel State

Each channel is represented by an on-chain account (typically a PDA derived from payer, payee, mint, authorized signer, a salt, and the open slot) with the following logical fields. Field names use camelCase; tag and enum-variant values (`Channel`, `Open`, `Closing`, `Sealed`, `Distributed`) use PascalCase by convention, matching how they appear in Rust program source.

| Field | Type | Storage | Description |
|---|---|---|---|
| `discriminator` | u8 | Account state | Non-zero account-type tag (`Channel`); rejected when 0 so zero-initialized PDAs cannot impersonate a channel |
| `version` | u8 | Account state | Account-layout version; lets implementations evolve the account layout across program versions |
| `bump` | u8 | Account state | Canonical PDA bump |
| `status` | u8 | Account state | `Open` / `Closing` / `Sealed` / `Distributed` enum value. `Distributed` is terminal and fully drained — every token leg paid, escrow ATA closed — inert to every instruction except `reclaim` and holding only the PDA's own rent (see Section 6.2.10) |
| `salt` | u64 | Seed + Account state | PDA disambiguator. Persisted so the channel PDA can re-derive its own seeds for self-signed CPIs (refunds, distribution) without off-chain inputs |
| `deposit` | u64 | Account state | Total amount currently escrowed |
| `settled` | u64 | Account state | Cumulative amount authorized for distribution (voucher watermark) |
| `payoutWatermark` | u64 | Account state | Distribution watermark (`payoutWatermark <= settled`); `distribute` advances it to `settled` and pays cumulative floor deltas between the old and new watermark (see Section 10) |
| `closureStartedAt` | i64 | Account state | Unix timestamp when `requestClose` was called (0 if not set; cleared on `Sealed`) |

| Field | Type | Storage | Description |
|---|---|---|---|
| `payerWithdrawnAt` | i64 | Account state | Unix timestamp of the payer refund (0 if not yet); guards against double-refund when both `withdrawPayer` and `distribute` can pay the payer |
| `gracePeriod` | u32 | Account state | Non-zero seconds between `requestClose` and permissionless `seal`. Per-channel, set at open, so a single program deployment can host channels with differing dispute windows |
| `distributionHash` | [u8;32] | Account state | Hash digest of the canonical splits preimage committed at open; `distribute` **MUST** re-verify this hash before paying recipients |
| `payer` | Pubkey | Seed + Account state | Client who deposited funds |
| `payee` | Pubkey | Seed + Account state | Server authorized to settle; receives the implicit-remainder share on `distribute` |
| `authorizedSigner` | Pubkey | Seed + Account state | Voucher signer; **MAY** equal `payer` or a delegated signer |
| `mint` | Pubkey | Seed + Account state | SPL Token or Token-2022 mint. Stored (not seed-only) so refund / distribution CPIs can be validated without re-binding seeds |
| `rentPayer` | Pubkey | Account state | The operator / transaction submitter that funded the channel PDA and escrow ATA rent at open. Recorded so the terminal `distribute` (fast path) or the permissionless `reclaim` can drain the freed SOL rent to this account without an off-chain input. Distinct from `payer`: the client (`payer`) only moves stablecoin and never needs SOL |

| Field | Type | Storage | Description |
|---|---|---|---|
| `openSlot` | u64 | Seed + Account state | Client-supplied per-incarnation epoch, carried in the open instruction data and window-validated against the Clock sysvar (see Section 6.2.10). A PDA seed, so the channel address itself is per-incarnation and `channelId` alone identifies an incarnation; persisted for signer-seed reconstruction and as the reclaim-gate input |

*Table 1*

The `channelId` is the base58-encoded address of the channel account (PDA). Channel programs **MUST** derive the channel PDA deterministically from channel parameters and the program ID. At minimum, the seed set **MUST** bind the PDA to:

- the payer public key;
- the payee public key;
- the mint address (native SOL is unsupported; clients wishing to pay in SOL **MUST** wrap to wSOL before opening a channel);
- a client-chosen salt or nonce;
- the authorized signer public key (or payer if no delegation is used); and
- the client-supplied `openSlot` epoch, so that each incarnation of the same participant tuple derives a distinct address (see Section 6.2.10).

Once a channel is opened, vouchers for that channel **MUST** verify under the channel's `authorizedSigner`. No other signer is valid for that channel.

Clients and servers **MUST** derive the expected `channelId` from the channel program ID and the seed components above and **MUST** verify that the open transaction creates and funds exactly that PDA. Relying on a client-declared `channelId` string alone is NOT sufficient.

Channel programs **MUST** use Solana's canonical PDA derivation procedure and **MUST** reject non-canonical addresses or user-supplied bump values that do not match the canonical derivation for the channel seeds.

`Channel` state — `deposit`, `settled`, `payoutWatermark`, and `payerWithdrawnAt` — is authoritative for pending settlement value. The escrow ATA balance and the channel PDA's lamports can exceed those values, because third parties can prefund either address; the program does not record those surpluses in `Channel`. Off-chain consumers **MUST** derive spendable capacity and pending settlement from channel state, never from raw escrow ATA balances or PDA lamports.

The SOL rent backing the channel PDA and escrow ATA is funded at `open` by `rentPayer` (the operator / transaction submitter), not by the client. The freed SOL rent is returned to `Channel.rentPayer` at close — the escrow-ATA rent by the `Sealed` `distribute`, and the channel PDA's own rent either in the same instruction (fast path) or by a later permissionless `reclaim`

(see Section 6.2.10). The token `deposit` and any surplus PDA lamports are independent of this rent accounting: the token refund of `deposit - settled` always goes to the `payer`, while surplus PDA lamports drain to `rentPayer` at close.

## 6.2. Instructions

### 6.2.1. open

Creates the channel account, transfers the initial deposit from the payer, and commits a hash of the distribution splits preimage. The payer **MUST** be a signer.

| Parameter | Type | Description |
|---|---|---|
| `salt` | u64 | PDA disambiguator |
| `deposit` | u64 | Initial deposit in base units; **MUST** be non-zero |
| `gracePeriod` | u32 | Forced-close grace period in seconds; stored per-channel; encoded as `grace_period`; **MUST** be non-zero |
| `openSlot` | u64 | Client-supplied per-incarnation epoch; encoded as `open_slot`; window-validated against the Clock sysvar (see Section 6.2.10). A PDA seed: it is a derivation input for the channel address, and servers **MUST** include it when re-deriving or validating `channelId`. The open transaction **MUST** land within `OPEN_SLOT_WINDOW` (1,500 slots — ~10 min at 400 ms slots) of this slot; standard transactions are bounded tighter still by the 150-block blockhash validity, while durable-nonce transactions get the full window. A flow that misses the window **MUST** re-derive with a fresh `openSlot` — and therefore a fresh `channelId` — and re-sign; no other protocol message carries an on-chain deadline |
| `distributionSplits` | `(Pubkey, u16)[]` | Splits preimage; canonical encoding hashed into `distributionHash` (see Section 10) |

*Table 2*

The reference instruction-data layout after the instruction discriminator is:

```
salt (u64 LE) || deposit (u64 LE) ||
grace_period (u32 LE) || open_slot (u64 LE) ||
count (u32 LE) || entries (count × 34 bytes)
```

`open` takes the following leading accounts, in this exact order:

| Index | Account | Signer | Writable | Description |
|---|---|---|---|---|
| 0 | `payer` | Yes | Yes | Client depositing stablecoin; signs the deposit transfer |
| 1 | `rentPayer` | Yes | Yes | Operator / transaction submitter funding the SOL rent for the channel PDA and escrow ATA. **MUST** be the operator / fee-payer key already in scope (the same key that co-signs `open` as fee payer); a single operator signature satisfies both the fee-payer and `rentPayer` signer roles. Recorded into `Channel.rentPayer` so its rent can be reclaimed at close. There is no separate wire field for `rentPayer`; it is derived from the existing operator / fee payer |
| 2 | `payee` | No | No | Channel payee |
| 3 | `mint` | No | No | SPL Token / Token-2022 mint |
| 4 | `authorizedSigner` | No | No | Voucher signer bound into the PDA seeds |
| 5 | `channel` | No | Yes | Channel PDA being created |

*Table 3*

The remaining accounts (payer token account, escrow token account, token program, system program, rent, associated-token program, and program-internal accounts) follow `channel` in their fixed order. Verifiers that read `open` accounts by fixed index **MUST** account for `rentPayer` at index 1 and the resulting `+1` shift of every account after `payer`, and **MUST** verify that `accounts[1]` equals the operator / fee-payer key.

The client (`payer`) only ever moves stablecoin and never needs SOL: `rentPayer` funds all channel rent at `open` and recovers it after close, via the terminal `distribute`'s fast path or a later permissionless `reclaim` (see Section 6.2.10).

`open` **MUST** validate the client-supplied `openSlot` against the Clock sysvar: `openSlot <= clock.slot` and `clock.slot - openSlot <= OPEN_SLOT_WINDOW` (see Section 6.2.10). Future slots **MUST** be strictly rejected (reference error `OpenSlotOutOfWindow`, code 2003): a far-future `openSlot` would otherwise break the address-never-repeats argument of Section 17.6 and push the reclaim gate arbitrarily far out, permanently stranding the operator's PDA rent. `open` **MUST** reject any `distributionSplits` whose preimage is malformed, whose total share exceeds 10000 bps, which contains duplicate recipients, or which lists the derived channel PDA as a recipient. Mints carrying Token-2022 extensions outside the allow-list (see Section 17.13) **MUST** be rejected.

The `gracePeriod` parameter **MUST** be non-zero. Channel programs **MUST** reject `grace_period == 0`.

`open` does NOT curve-check `payee`; both on-curve and PDA payees are permitted (see Section 6.2.6).

`open` is prefund-tolerant. The channel PDA allocation and the escrow ATA creation are both idempotent: a prefunded but still-uninitialized channel PDA (a system-owned, data-empty account holding only lamports) or a pre-existing canonical escrow ATA is accepted rather than reverting. Prefunded balances are never credited to channel state — surplus PDA lamports drain to `rentPayer` at close, and surplus escrow tokens are swept to the treasury by the terminal `distribute`.

Servers **MUST** use a `salt` that keeps concurrent live channels between the same participants distinct. Reusing the full seed tuple — `(payer, payee, mint, authorizedSigner, salt, openSlot)` — of a live or still-`Distributed` channel reverts `open` (the PDA still holds an initialized `Channel`); resume a live channel instead of reopening it. Reopening a fully closed relationship is legal by design and creates a fresh channel at a **new address**: `distribute`'s fast path or `reclaim` removed the old PDA, and because `openSlot` is a PDA seed, the new incarnation (the same `salt` is fine) necessarily carries a new `openSlot` and therefore derives a different `channelId` (see Section 6.2.10). A lamport donation to a deallocated channel address cannot block reopening, because `open` is prefund-tolerant (Transfer + Allocate + Assign).

`open` does NOT carry an initial voucher; the first voucher is exchanged off-chain after confirmation.

#### 6.2.2. settle

Advances the on-chain `settled` watermark using a voucher signed by `authorizedSigner`. Permissionless; authority is the voucher signature.

`settle` takes no instruction-data arguments; the voucher is carried entirely by the preceding Ed25519 precompile instruction.

The submitter **MUST** bundle a Solana native `ed25519` precompile instruction immediately before `settle` in the same transaction. The program reads the verified message bytes via the Instructions sysvar, decodes the voucher (`magic`, `channelId`, `cumulativeAmount`, `expiresAt`) from them (see Section 9.3), asserts the `magic` prefix matches exactly (reference error `VoucherBadMagic`, code 238), asserts `channelId` equals the channel PDA address (reference error `VoucherChannelMismatch`, code 232 — because `openSlot` is a PDA seed, this address binding also covers cross-incarnation replay; no separate epoch check exists), and asserts the precompile-recorded signer equals `authorizedSigner`. The program then verifies `settled < cumulativeAmount <= deposit` and writes `settled = cumulativeAmount`. No token transfer occurs in `settle`, and `settle` is not slot-gated.

#### 6.2.3. topUp

Payer transfers additional funds to the escrow.

| Parameter | Type | Description |
|---|---|---|
| `amount` | u64 | Amount to add in base units; **MUST** be non-zero |

*Table 4*

`topUp` requires `status == Open` and **MUST** be rejected when `status == Closing`. Implementations of this specification do NOT clear `closureStartedAt` via `topUp`. The payer **MUST** be a signer.

#### 6.2.4. requestClose

Payer initiates a forced close. Sets `closureStartedAt = Clock::get().unix_timestamp`, `status = Closing`. Requires `status == Open`. The payer **MUST** be a signer.

#### 6.2.5. seal

Permissionless post-grace crank. Transitions `Closing -> Sealed` once `now >= closureStartedAt + gracePeriod`, clears `closureStartedAt`, and freezes `settled`. No token transfer occurs.

#### 6.2.6. settleAndSeal

Payee-initiated cooperative close. Optionally applies one final voucher (using the same precompile-verified path as `settle`), then transitions the channel to `Sealed`.

| Parameter | Type | Description |
|---|---|---|
| `hasVoucher` | u8 | `0` seals with no settlement (full refund); non-zero verifies and settles the voucher carried by the preceding Ed25519 precompile instruction (read via the Instructions sysvar) before sealing |

*Table 5*

The payee **MUST** be a signer. Callable from `Open` and from `Closing` while `now < closureStartedAt + gracePeriod`; after the grace deadline use `seal` instead. No token transfer occurs.

The `payee` **MAY** be an on-curve address or a program-derived address (PDA). Because cooperative close requires a transaction signer equal to `Channel.payee`, a PDA payee can use this path only when its owning program invokes `settleAndSeal` via CPI with signer seeds. The permissionless `settle`, `seal`, and `distribute` cranks need no payee signature.

#### 6.2.7. distribute

Pays the merchant-side pool out of escrow according to the splits preimage committed at `open`. Permissionless; authority is the on-chain hash commitment.

| Parameter | Type | Description |
|---|---|---|
| `distributionSplits` | `(Pubkey, u16)[]` | Splits preimage (see Section 10); rehashed and **MUST** equal `distributionHash` |

*Table 6*

`distribute` takes a fixed head of accounts (channel, `payer`, `rentPayer`, escrow token account, payer token account, payee token account, treasury token account, mint, token program, and program-internal accounts) followed by the dynamic recipient-ATA tail. The `rentPayer` account (writable, NOT a signer) **MUST** be positioned immediately after `payer` and **MUST** equal `Channel.rentPayer`; at the `Sealed` branch it receives the SOL rent freed by closing the escrow ATA, plus every lamport of the channel PDA when the fast path deallocates the account in place (see Section 6.2.10).

Recipient token accounts are supplied as the dynamic account tail, in the same order as the preimage entries. Each **MUST** be the canonical ATA for `(recipient, channel.mint, channel.tokenProgram)`. A `distribute` carrying enough recipient accounts to exceed the legacy transaction account-key budget — in practice at `MAX_DISTRIBUTION_RECIPIENTS` recipients (**RECOMMENDED** 32) — **MUST** be sent as a version-0 transaction with an address lookup table indexing the recipient ATAs.

Each beneficiary is paid a cumulative floor delta keyed to `payoutWatermark`:

- recipient `i`: `floor(settled * shareBps[i] / 10000) − floor(payoutWatermark * shareBps[i] / 10000)`;
- payee (implicit remainder): `floor(settled * (10000 − Σ shareBps) / 10000) − floor(payoutWatermark * (10000 − Σ shareBps) / 10000)`.

`distribute` then advances `payoutWatermark` to `settled`.

From `Open`, `distribute` requires `settled > payoutWatermark`, pays the cumulative floor deltas, leaves flooring-residual dust in the escrow ATA, advances `payoutWatermark` to `settled`, and keeps the channel `Open`; later distributions compute fresh deltas from the advanced watermark, so residual value remains claimable as a share's cumulative entitlement crosses the next whole unit. From `Sealed`, `distribute` additionally — when `payerWithdrawnAt == 0` — transfers the token refund `deposit - settled` to the payer, stamps `payerWithdrawnAt`, sweeps the final irreducible residual dust to the treasury ATA, and closes the escrow ATA. None of this token movement is slot-gated: the `Sealed` branch runs immediately and **MUST NOT** emit `ChannelCloseTooEarly`. In the same instruction, when `clock.slot > openSlot + OPEN_SLOT_WINDOW` already holds, the channel PDA is fully deallocated in place (fast path); otherwise the channel is set to the terminal `Distributed` status and its rent is recovered later by the permissionless `reclaim` (see Section 6.2.10). The freed SOL — the escrow ATA rent plus, at deallocation, every lamport of the channel PDA, including any prefund surplus — is drained to `Channel.rentPayer` (the operator), not the payer; the token refund still goes to the payer. `distribute` **MUST NOT** be callable from `Closing` or `Distributed`.

On a nonzero beneficiary share whose canonical ATA is unusable — missing or uninitialized, frozen, closed or malformed, carrying an unsupported Token-2022 account extension, or with a reassigned authority — that share is redirected to the treasury ATA, a `PayoutRedirected` event is emitted, and `payoutWatermark` still advances. The beneficiary permanently forfeits that share; repairing the ATA later does not reclaim it, because future deltas only cover newly settled amounts. The same redirect applies to the payer refund ATA at `Sealed` (the same instruction closes the escrow ATA, so no later crank could pay the refund). Malformed token-account data and wrong (non-canonical) accounts hard-fail rather than redirecting.

#### 6.2.8. withdrawPayer

One-shot payer refund in `Sealed` that does NOT close or deallocate the channel PDA and is NOT slot-gated. The program requires `status == Sealed` and `payerWithdrawnAt == 0`, transfers `deposit - settled` to the payer, and stamps `payerWithdrawnAt`. The payer **MUST** be a signer.

#### 6.2.9. reclaim

Permissionless rent-recovery crank. Deallocates a fully drained `Distributed` channel PDA and returns every remaining lamport to `Channel.rentPayer`. By the time a channel is `Distributed`, every token leg has been paid and the escrow ATA has been closed by the `Sealed` `distribute`; the only value left at the address is the PDA's own SOL rent, so delaying `reclaim` delays nobody's money.

`reclaim` takes no instruction-data arguments and no signers, and exactly two accounts:

| Index | Account | Signer | Writable | Description |
|---|---|---|---|---|
| 0 | `channel` | No | Yes | Channel PDA; **MUST** be `Distributed`. Deallocated; all lamports drained |
| 1 | `rentPayer` | No | Yes | **MUST** equal the recorded `Channel.rentPayer`; receives every remaining lamport |

*Table 7*

`reclaim` **MUST** require `status == Distributed`, **MUST** verify the supplied `rentPayer` account equals the recorded `Channel.rentPayer`, and **MUST** reject with reference error `ChannelCloseTooEarly` (code 2414) until `clock.slot > openSlot + OPEN_SLOT_WINDOW`; a rejected `reclaim` is safely retryable once the window elapses. The gate exists solely to keep the address occupied through the epoch window — the address-never-repeats invariant of Section 17.6 — not to sequence any payment.

Because `reclaim` needs only two writable accounts and no signers, operators **SHOULD** batch many `reclaim` instructions into a single periodic sweep transaction. `reclaim` is unnecessary for a channel whose terminal `distribute` ran after the window had already elapsed: the fast path deallocates directly (see Section 6.2.10).

#### 6.2.10. Channel Closure

Closure is two-phase: all value moves immediately, and only the recovery of the channel PDA's own rent waits for an epoch window.

**Phase 1 — drain (immediate).** The `Sealed` branch of `distribute` pays the merchant-side cumulative deltas, refunds `deposit - settled` to the payer (when not already withdrawn), sweeps residual dust to the treasury, and closes the escrow ATA, returning the escrow-ATA rent to `Channel.rentPayer`. None of this is slot-gated: the epoch window never delays a payout or a refund.

**Phase 2 — deallocate (window-gated).** After the drain, the channel holds only its own PDA rent. When `clock.slot > openSlot + OPEN_SLOT_WINDOW` already holds, `distribute` deallocates the PDA in the same instruction (fast path): every lamport — the rent funded at `open` plus any prefund surplus — is drained to `Channel.rentPayer` (the operator that funded the rent at `open`), not the payer, the account data is zeroed, and the runtime garbage-collects the account. Otherwise `distribute` sets `status = Distributed`: the channel is fully drained and inert to every instruction except `reclaim`, and its continued existence keeps the address occupied until the window elapses, when the permissionless `reclaim` deallocates it identically. Once deallocated, the address becomes reopenable as a new incarnation. `withdrawPayer` **MUST NOT** close or deallocate the channel.

Full deallocation is safe against voucher replay because the channel address is per-incarnation by construction: `openSlot` is a PDA seed, so `channelId` alone identifies one incarnation and an address can never host two channels. `openSlot` is a client-supplied per-incarnation epoch carried in the `open` instruction data and validated on-chain against the Clock sysvar:

```
openSlot <= clock.slot
clock.slot - openSlot <= OPEN_SLOT_WINDOW
```

where `OPEN_SLOT_WINDOW = 1500` slots (approximately 10 minutes at 400 ms slots; the window is measured in slots and **MUST** be at least the 150-block blockhash validity, so any deliverable transaction passes it at any slot duration). Future slots are strictly rejected (reference error `OpenSlotOutOfWindow`, code 2003): a far-future `openSlot` would otherwise break the uniqueness argument below and push the reclaim gate arbitrarily far out, permanently stranding the operator's PDA rent.

Only `reclaim` carries the slot gate: it **MUST** reject with reference error `ChannelCloseTooEarly` (code 2414) until `clock.slot > openSlot + OPEN_SLOT_WINDOW`, and a rejected `reclaim` is safely retryable. `distribute` **MUST NOT** emit `ChannelCloseTooEarly`; when the window has not yet elapsed, its `Sealed` branch simply leaves the channel `Distributed` instead of deallocating it. `withdrawPayer`, `settle`, `settleAndSeal`, `seal`, and both branches of `distribute` are NOT slot-gated; only the PDA deallocation — rent recovery — waits.

Together, the open window and the occupied address guarantee that a channel address never repeats: the address stays occupied — live, then `Distributed` — until some slot strictly greater than `openSlot + OPEN_SLOT_WINDOW`, and from then on the `openSlot` baked into the address's seeds is too stale for `open` to ever re-derive it. A voucher bound to an earlier incarnation can never settle against a later one, because the later incarnation lives at a different address. See Section 17.6 for the uniqueness argument and the constraint on evolving `OPEN_SLOT_WINDOW`.

Because the client chooses `openSlot`, it can derive the channel address (`openSlot` is one of the PDA derivation inputs) at transaction-build time and **MAY** construct and sign vouchers before the open transaction confirms; no post-open read-back is required to produce a voucher. The open-landing window and the reclaim unlock share the same `OPEN_SLOT_WINDOW` budget measured from the supplied slot — but the only thing the window delays is the operator's recovery of the channel PDA's own rent (roughly 2.7 million lamports per channel, for at most the window). Supplying the current slot maximizes landing safety at the cost of the full rent float; back-dating `openSlot` by `k` slots shrinks the landing window to `OPEN_SLOT_WINDOW - k` slots and shortens the rent float — a close landing after the dated window even takes `distribute`'s fast path and skips `reclaim` entirely. Back-dating never affects payout or refund latency, which is zero-wait either way.

Because `reclaim` takes only two writable accounts and no signers, operators **SHOULD** run a periodic sweep that batches many `reclaim` instructions per transaction across their `Distributed` channels.

Implementations **MUST NOT** treat a fee-payer signature as satisfying payer or payee authority checks on any authority-gated instruction above.

## 6.3. Grace Period

The grace period (**RECOMMENDED**: 15 minutes) protects the payee. If the payer calls `requestClose` while the payee has unsubmitted vouchers, the payee has until the grace period expires to call `settle` followed by `settleAndSeal` (or to bundle a voucher into `settleAndSeal` directly).

Without a grace period, the payer could `requestClose`, immediately call `seal`, and sweep funds before the server has time to settle.

## 6.4. Access Control

| Instruction | Caller | Gating |
|---|---|---|
| open | Payer | Payer signs the deposit transfer |
| settle | Anyone (permissionless crank) | Precompile-verified Ed25519 voucher from `authorizedSigner` |

| Instruction | Caller | Gating |
|---|---|---|
| topUp | Payer | Payer signs the additional transfer; rejected when `status != Open` |
| requestClose | Payer | Payer signer equals channel `payer` |
| seal | Anyone (permissionless crank) | `status == Closing` and elapsed grace period |
| settleAndSeal | Payee | Payee signer equals channel `payee` |
| distribute | Anyone (permissionless crank) | On-chain hash commitment to splits preimage; never slot-gated (the `Sealed` branch deallocates the PDA in place only when the epoch window has already elapsed; see Section 6.2.10) |
| withdrawPayer | Payer | Payer signer equals channel `payer` and `status == Sealed` |
| reclaim | Anyone (permissionless crank) | `status == Distributed`, supplied `rentPayer` equals the recorded `Channel.rentPayer`, and `clock.slot > openSlot + OPEN_SLOT_WINDOW` (see Section 6.2.10) |

*Table 8*

### 6.5. Account Shapes and Events

Every instruction takes an exact account list and rejects transactions with missing OR extra accounts. The only dynamic account tail is `distribute`'s recipient token accounts (one canonical ATA per active preimage entry, in preimage order). Conforming generated clients enforce the same shapes, so callers cannot pad an instruction with unexpected accounts.

The channel program declares two events in its IDL: `Opened` (emitted by `open`) and `PayoutRedirected` (emitted by `distribute` when a beneficiary share is redirected to the treasury; see Section 17.4). Each event carries an 8-byte discriminator so IDL-driven indexers can decode it without custom tooling.

## 7. Request Schema

### 7.1. Shared Fields

amount **REQUIRED**. Price per unit of service in the token's smallest unit, encoded as a decimal string.

unitType **OPTIONAL**. Unit being priced (for example, `"request"`, `"token"`, or `"byte"`).

suggestedDeposit **OPTIONAL**. Suggested initial channel deposit in base units. Clients **MAY** deposit less or more depending on expected usage.

minimumDeposit **OPTIONAL**. Hard floor on initial channel deposit in base units. Enforced at the HTTP layer (not on chain). Servers **MUST** reject `POST /channel/open` payloads with `depositAmount < minimumDeposit`. Implementations **SHOULD** set this to a minimum economically useful balance to avoid spam; channel rent is fully recovered at close (see Section 6.2.10), so the floor guards signature-verification and settlement overhead rather than storage cost.

recipient **REQUIRED**. Base58-encoded public key of the server's account that will receive settlement funds.

currency **REQUIRED**. Base58-encoded SPL token mint address. Native SOL is not supported; clients wishing to pay in SOL **MUST** wrap it to wSOL (`So11111111111111111111111111111111111111112`) before opening a channel.

description **OPTIONAL**. Human-readable description of the service or resource being paid for.

externalId **OPTIONAL**. Merchant reference for reconciliation or audit correlation.

## 7.2. Method Details

network **REQUIRED**. Solana cluster identifier. **MUST** be one of "mainnet", "devnet", or "localnet". There is no default; the challenge **MUST** state the cluster explicitly.

channelProgram **REQUIRED**. Base58-encoded address of the on-chain channel program, which **MUST** be the program explicitly deployed for the selected `network`. Clients **MUST** verify this matches their expected program for that cluster before depositing funds.

channelId **OPTIONAL**. Existing channel identifier to resume. When present, clients **SHOULD** verify the referenced channel is open and sufficiently funded before reuse.

recentBlockhash Conditionally **REQUIRED** when `channelId` is absent and **MUST** be absent when resuming an existing channel. Base58-encoded recent blockhash for the client to use when constructing the open transaction. The server **MUST** obtain it from the selected `network`, and the client **MUST** use it as the transaction message's recent blockhash.

recentSlot Conditionally **REQUIRED** when `channelId` is absent and **MUST** be absent when resuming an existing channel. Decimal-string u64 identifying the RPC context slot associated with `recentBlockhash`. It provides the reference from which the client selects the open credential's `openSlot` without making its own RPC request.

decimals Conditionally **REQUIRED**. Token decimal places (0–9). **MUST** be present when `currency` is a mint address.

tokenProgram **OPTIONAL**. Base58-encoded token program ID for the mint in `currency`. **MUST** be either the SPL Token Program or the Token-2022 Program when present. If omitted for a mint-based `currency`, clients **MUST** determine the correct token program from on-chain state before constructing token instructions.

feePayer **OPTIONAL**. If `true`, the server sponsors transaction fees for open, topUp, and close operations. When `true`, `feePayerKey` **MUST** also be present.

feePayerKey Conditionally **REQUIRED**. Base58-encoded public key of the server's fee payer account.

voucherSigner **OPTIONAL**. Party that signs cumulative vouchers. **MUST** be either `client` or `operator`. Defaults to `client`.

operator Conditionally **REQUIRED** when `voucherSigner` is `operator`. Base58-encoded Ed25519 public key that **MUST** equal the channel's `authorizedSigner`. The operator signs cumulative vouchers after authenticated requests are metered. This key **MAY** equal `feePayerKey`, but the fields describe separate roles.

minVoucherDelta **OPTIONAL**. Minimum amount increase between accepted vouchers.

ttlSeconds **OPTIONAL**. Suggested session duration in seconds. This field does not change the idle-timeout negotiation defined in Section 14.6.

idleTimeoutOptionsSeconds **OPTIONAL**. Server-supported inactivity thresholds for a new channel, in seconds. When present, this **MUST** be a non-empty, strictly increasing array of distinct integers from 1 through 2592000, inclusive (1 second through 30 days). When omitted, the server selects the effective timeout at its discretion and the client cannot request a specific value. For example, a server can advertise `[30, 600, 86400]`. The client selects one option in its `open` credential or lets the server choose. This field **MUST** be absent when `channelId` is present. See Section 14.6.

idleTimeoutSeconds Conditionally **REQUIRED** when `channelId` is present. Effective negotiated threshold for the resumed channel, as an integer from 1 through 2592000 seconds, inclusive (1 second through 30 days). This field **MUST** be absent from a challenge for a new channel. The value is server-side state and is not stored in the on-chain channel account.

gracePeriodSeconds Conditionally **REQUIRED**. Grace period for forced close when `channelId` is absent (**RECOMMENDED**: 900). Stored per-channel in `Channel.gracePeriod` at `open`. The value **MUST** be greater than zero.

distributionSplits **OPTIONAL**. Ordered list of `{recipient, shareBps}` entries the merchant proposes to bind into the channel at `open`. The payee receives the implicit remainder share `10000 − Σ shareBps`; the explicit list therefore covers only co-recipients, not the payee itself.

Each entry **MUST** have `shareBps > 0`. The list **MUST** satisfy `0 ≤ Σ shareBps ≤ 10000`. The list size is bounded by an implementation-defined `MAX_DISTRIBUTION_RECIPIENTS` (**RECOMMENDED**: 32).

When omitted, the channel behaves as a vanilla two-party channel in which the payee receives the full distributed pool.

For the `session` intent, `amount` specifies the price per unit of service, not a total charge. When `unitType` is present, clients can estimate cost before a session begins:

```
total = amount × units_consumed
```

# 8. Credential Schema

The credential payload uses a discriminated union on the `action` field. Five actions are defined.

These actions map to the abstract session lifecycle operations of [I-D.payment-intent-session] as follows:

| Abstract Operation | This Method's action |
|---|---|
| Open | `open` |
| Use | `voucher` for `client`; `use` for `operator` |
| Top-Up | `topUp` |
| Close | `close` |

*Table 9*

## 8.1. Session Bearer Proof

An operator-signed channel uses one payer-signed proof for `Open` and each subsequent `Use`. It authorizes access, not payment.

The client derives `channelId` from the channel program and open parameters, then constructs this JCS [RFC8785] object:

```
{
  "channelId": "<base58 channel address>",
  "domain": "mpp-session-auth-v1",
  "payer": "<base58 payer public key>",
  "sessionChallengeId": "<session challenge id>"
}
```

The client signs the UTF-8 bytes of the JCS serialization with the Ed25519 private key corresponding to `payer`. The resulting `authentication` object has this shape:

| Field | Type | Required | Description |
|---|---|---|---|
| `type` | string | **REQUIRED** | The string `"proof"` |
| `challengeId` | string | **REQUIRED** | Session challenge ID signed as `sessionChallengeId` |
| `payer` | string | **REQUIRED** | Base58 Ed25519 public key that produced the proof |
| `signature` | string | **REQUIRED** | Base58 encoding of the 64-byte Ed25519 signature |

*Table 10*

At open, the server **MUST** verify the session challenge and signature. The proof's `challengeId`, `channelId`, and `payer` **MUST** equal the opening challenge ID, declared and derived channel ID, and transaction payer. The verified challenge transitively binds its remaining policy. The server **MUST** bind the proof to exactly one channel; its verifier storage is implementation-defined.

For a subsequent `use` action, the outer Payment credential **MUST** echo the same session challenge that was bound at open and **MUST** carry the same `authentication` field values. The server **MUST** verify the proof against the bound channel state. The challenge's `expires` auth-param limits opening the binding, not later use by the open channel. The channel idle timeout still applies.

Repeated presentation is the expected bearer-proof behavior for the generic session `Use` operation. It **MUST NOT** be treated as fulfillment of a `charge` intent or rejected merely because it was previously presented. The proof remains valid while the channel is open, including when it temporarily has insufficient capacity. It becomes invalid when close begins or the channel otherwise becomes terminal.

The `authentication` object cannot by itself open, fund, top up, or settle a channel. The open transaction and cumulative vouchers remain the payment proofs and retain their normal replay protections.

## 8.2. Action: "open"

Opens a new payment channel.

| Field | Type | Required | Description |
|---|---|---|---|
| `action` | string | **REQUIRED** | `"open"` |
| `channelId` | string | **REQUIRED** | Base58 channel account address |
| `payer` | string | **REQUIRED** | Base58 public key of the depositor |
| `payee` | string | **REQUIRED** | Base58 public key of the channel payee (matches `recipient` in the 402 challenge) |

| Field | Type | Required | Description |
|---|---|---|---|
| `mint` | string | **REQUIRED** | Base58 SPL Token / Token-2022 mint (matches `currency` in the 402 challenge) |
| `authorizedSigner` | string | **REQUIRED** | Base58 public key bound into the PDA seeds as the voucher signer; **MAY** equal `payer` or a delegated signer |
| `salt` | string | **REQUIRED** | Decimal u64 PDA disambiguator |
| `depositAmount` | string | **REQUIRED** | Initial deposit in base units; **MUST** equal the decoded open deposit and satisfy `depositAmount >= minimumDeposit` (when the challenge sets one) |
| `gracePeriodSeconds` | integer | **REQUIRED** | Grace-period seconds bound into channel state at open; **MUST** be greater than zero and **MUST** match the challenge's `methodDetails.gracePeriodSeconds` |
| `idleTimeoutSeconds` | integer | **OPTIONAL** | Client-selected inactivity threshold; **MAY** be present only when the challenge advertises `idleTimeoutOptionsSeconds` and **MUST** exactly match an offered value; when omitted, lets the server select the effective value |
| `openSlot` | string | **REQUIRED** | Decimal u64 per-incarnation epoch encoded into the open instruction as `open_slot`; **MUST** be no greater than the challenge's `methodDetails.recentSlot` and **MUST** satisfy the on-chain window rule of Section 6.2.10 when the transaction executes. Also a PDA derivation input: servers **MUST** include it when re-deriving and validating `channelId` |
| `distributionSplits` | array | **OPTIONAL** | Splits preimage (see the challenge's `methodDetails.distributionSplits`); **MUST** byte-match the splits proposed in the 402 challenge |

| Field | Type | Required | Description |
|---|---|---|---|
| `authorizationPolicy` | object | **OPTIONAL** | Voucher signer policy. When present, **MUST** be consistent with `authorizedSigner` |
| `authentication` | object | Conditionally **REQUIRED** | Reusable proof from Section 8.1; **REQUIRED** for `operator` and **MUST** be absent for `client` |
| `transaction` | string | **REQUIRED** | Base64-encoded (standard alphabet, padded) signed or partially signed transaction |
| `capabilities` | object | **OPTIONAL** | Implementation-specific extensions |

*Table 11*

The `transaction` contains the open instruction(s). When `feePayer` is `true`, the client partially signs (transfer authority only) and the server co-signs as fee payer before broadcasting — same pattern as the charge intent's pull mode.

`Action: "open"` **MUST NOT** carry an initial voucher. The first voucher is exchanged off-chain in a subsequent metered request, after the channel is confirmed on-chain. This keeps the open path focused on channel construction and avoids burning on-chain compute on a signature for a single request's worth of authorization.

Clients **SHOULD** set `openSlot` to the challenge's `methodDetails.recentSlot` and **MUST** use the challenged `recentBlockhash` in the transaction. A client **MAY** choose an earlier `openSlot` to shorten the operator's post-close rent float. Because the client chooses `openSlot` — and it is one of the PDA derivation inputs — it can derive the channel address before the open transaction confirms and **MAY** pre-sign vouchers for the new channel immediately; no post-open read-back is required. Back-dating `openSlot` by `k` slots shrinks the transaction-landing window to `OPEN_SLOT_WINDOW - k` slots and shortens the operator's post-close rent float (see Section 6.2.10); it has no effect on payout or refund latency, and a future slot is always rejected on-chain.

`Action: "open"` **MUST NOT** carry a `bump` field. The channel PDA's canonical bump is derived on-chain via `find_program_address` and validated by the program's direct address check, so any wire-supplied bump is redundant. Servers **MUST** reject open envelopes that include a `bump` field using the `malformed-credential` problem type. Silently accepting and ignoring a wire `bump` is forbidden because a client whose derivation is buggy can compute a wrong bump that nonetheless pairs with the canonical PDA address — a mismatch the on-chain address check cannot catch.

Servers **MUST** treat the decoded `transaction`, not the HTTP envelope, as the authoritative open request before signing, paying fees, or broadcasting. Servers **MUST** reject `Action: "open"` credentials when the challenge, HTTP payload, decoded transaction, derived PDA, escrow ATA, token program, or confirmed on-chain state disagree. See Section 14.1 for the required decoding and validation sequence.

For `client`, the client controls `authorizedSigner`. For `operator`, `authorizedSigner` **MUST** equal the challenged `operator`, and the server **MUST** verify `authentication` before broadcasting the open transaction. The public transaction proves authorization of the deposit and operator settlement key; it does not authenticate later metered requests.

Example open credential:

```
{
  "action": "open",
  "channelId": "C4HnVjA7WMUtSQzAv4G6T3qBjLwK5jM7PvE2nQ5sZ3kP",
  "payer":     "9xQeWvG816bUx9EPjHmaT23yvVM2ZWbrrpZb9PusVFin",
  "payee":     "FNvFqYn4yV7HsoZyHRsbsj1Vd2HFcUe2NMRJq3rJxg7c",
  "mint":      "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
  "authorizedSigner":
               "9xQeWvG816bUx9EPjHmaT23yvVM2ZWbrrpZb9PusVFin",
  "salt": "42",
  "depositAmount": "10000000",
  "gracePeriodSeconds": 900,
  "idleTimeoutSeconds": 86400,
  "openSlot": "352114093",
  "transaction": "AQAB...base64..."
}
```

For `voucherSigner="operator"`, the relevant decoded `open` fields include the operator key and bearer proof:

```
{
  "action": "open",
  "channelId": "C4HnVjA7WMUtSQzAv4G6T3qBjLwK5jM7PvE2nQ5sZ3kP",
  "payer": "9xQeWvG816bUx9EPjHmaT23yvVM2ZWbrrpZb9PusVFin",
  "authorizedSigner": "5fKb5cF22cFybZB1H4hLDydFhwoQy9JzKzRWaSbMkB6h",
  "authentication": {
    "type": "proof",
    "challengeId": "c_8d0e3b5a9f2c1d4e",
    "payer": "9xQeWvG816bUx9EPjHmaT23yvVM2ZWbrrpZb9PusVFin",
    "signature": "4vJ9...base58 Ed25519 signature...Qd"
  },
  "transaction": "AQAB...base64..."
}
```

Subsequent requests reuse the same `authentication` field:

```
{
  "action": "use",
  "channelId": "C4HnVjA7WMUtSQzAv4G6T3qBjLwK5jM7PvE2nQ5sZ3kP",
  "authentication": {
    "type": "proof",
    "challengeId": "c_8d0e3b5a9f2c1d4e",
    "payer": "9xQeWvG816bUx9EPjHmaT23yvVM2ZWbrrpZb9PusVFin",
    "signature": "4vJ9...base58 Ed25519 signature...Qd"
  }
}
```

### 8.3. Action: "voucher"

Submits a new voucher authorizing additional spend.

| Field | Type | Required | Description |
|---|---|---|---|
| `action` | string | **REQUIRED** | `"voucher"` |
| `channelId` | string | **REQUIRED** | Existing channel identifier |
| `voucher` | object | **REQUIRED** | Signed voucher (see Section 9) |

*Table 12*

This action is entirely off-chain. No transaction is broadcast. A `voucher` action directly authorizes service only for `client`. In `operator` mode, vouchers are operator-generated settlement artifacts and **MUST NOT** authenticate the caller.

### 8.4. Action: "use"

Authenticates a metered request under an operator-signed channel.

| Field | Type | Required | Description |
|---|---|---|---|
| `action` | string | **REQUIRED** | `"use"` |
| `channelId` | string | **REQUIRED** | Existing operator-signed channel identifier |
| `authentication` | object | **REQUIRED** | Reusable proof bound at open, as defined in Section 8.1 |

*Table 13*

This action is valid only when `voucherSigner` is `operator`. After authenticating the request, the operator meters the delivered service and signs the corresponding cumulative voucher.

### 8.5. Action: "topUp"

Adds funds to an existing channel.

| Field | Type | Required | Description |
|---|---|---|---|
| `action` | string | **REQUIRED** | `"topUp"` |
| `channelId` | string | **REQUIRED** | Existing channel identifier |
| `additionalAmount` | string | **REQUIRED** | Amount to add in base units |
| `transaction` | string | **REQUIRED** | Base64-encoded signed topUp transaction |

*Table 14*

### 8.6. Action: "close"

Requests cooperative close.

| Field | Type | Required | Description |
|---|---|---|---|
| `action` | string | **REQUIRED** | `"close"` |
| `channelId` | string | **REQUIRED** | Existing channel identifier |
| `voucher` | object | Conditionally **REQUIRED** | **REQUIRED** for `client` and **MUST** be absent for `operator`; authenticates close and **MAY** advance settlement (see Section 9) |
| `authentication` | object | Conditionally **REQUIRED** | Reusable proof; **REQUIRED** for `operator` and **MUST** be absent for `client` |

*Table 15*

`Action: "close"` is a request for the server to broadcast `settleAndSeal` (optionally bundled with `distribute` in the same transaction). Unlike `Action: "open"` and `Action: "topUp"`, the close credential does NOT carry a pre-signed transaction: cooperative close requires the payee signature, which the server controls, and the server constructs and broadcasts the transaction itself.

For `operator`, the server **MUST** verify `authentication` against the channel before initiating cooperative close. For `client`, the server **MUST** verify the voucher signature and channel binding. A voucher with `cumulativeAmount > settled` advances settlement. A voucher at or below `settled` **MUST** equal the server's `acceptedCumulative`, authenticates close only, and **MUST NOT** be submitted as a settlement update. This permits authenticated close when no additional payment is due.

See Section 14.5 for the full settlement procedure, including how `settleAndSeal` and `distribute` are bundled.

# 9. Voucher Format

## 9.1. Voucher Data

| Field | Type | Required | Description |
|---|---|---|---|
| `channelId` | string | **REQUIRED** | Channel this voucher authorizes |
| `cumulativeAmount` | string | **REQUIRED** | Total authorized spend (base units) |
| `expiresAt` | integer | **OPTIONAL** | Voucher expiration as a Unix timestamp in seconds (i64); `0` or omitted means no expiration. Encoded verbatim into the signed Borsh payload (see Section 9.3); no string/timezone conversion is performed at sign or verify time. |

*Table 16*

All other channel context (payer, recipient, token, program, and signer policy) is established by the on-chain channel state and the deterministic PDA derivation defined above. The voucher only needs to identify the channel — `channelId` — and authorize a cumulative amount, because `channelId` is already bound to that context and, since `openSlot` is a PDA seed, the address itself pins the incarnation (see Section 6.2.10); no separate epoch field is carried. Implementations **MUST NOT** accept vouchers for channels whose identity cannot be recomputed from the program ID and channel open parameters.

## 9.2. Signed Voucher

| Field | Type | Required | Description |
|---|---|---|---|
| `voucher` | object | **REQUIRED** | Voucher data (above) |
| `signer` | string | **REQUIRED** | Base58 public key of the voucher signer |
| `signature` | string | **REQUIRED** | Base58-encoded Ed25519 signature |
| `signatureType` | string | **REQUIRED** | `"ed25519"` |

*Table 17*

## 9.3. Voucher Signing

The signed voucher payload is 50 bytes in fixed Borsh layout:

| Offset | Length | Field | Encoding |
|---|---|---|---|
| 0 | 2 | `magic` | The tag byte `0x56` (ASCII `V`) followed by the format-version byte `0x01` |
| 2 | 32 | `channelId` | Raw Solana address bytes |
| 34 | 8 | `cumulativeAmount` | u64 little-endian |
| 42 | 8 | `expiresAt` | i64 little-endian; `0` = no expiration |

*Table 18*

The `magic` prefix is a domain-separation tag plus a payload format version (`0x01`): it separates voucher bytes from anything else the signing key might sign and pins the format version inside the signed bytes. The separation strength comes from the exact 50-byte message-length pin plus the `channelId` PDA binding, not from tag entropy: the tag byte only needs to differ from the first byte of other Ed25519-signable payloads (legacy transaction messages start with a small signature count, versioned transactions with `0x80`, offchain messages with `0xff`). There is no epoch field: `openSlot` is a PDA seed, so the `channelId` bytes already bind the voucher to one incarnation of the channel address (see Section 6.2.10).

Signing:

1. Serialize the voucher data into the layout above.
2. Sign with Ed25519 using `authorizedSigner`'s key.
3. Encode the signature as base58 for the HTTP `signature` field.

The Borsh bytes are authoritative for signature verification. The HTTP JSON shape is a transport view; clients and servers **MUST NOT** influence what bytes are signed via the JSON. The same 50-byte layout is the precompile message the channel program reads back on-chain for `settle` and for `settleAndSeal` when a voucher is applied.

## 9.4. Voucher Verification

The server **MUST** verify each voucher:

1. Deserialize the voucher data and serialize it into the 50-byte layout of Section 9.3, including the fixed `magic` prefix. A payload whose `magic` does not match exactly **MUST** be rejected.
2. Verify the Ed25519 signature over the Borsh voucher payload against the `signer` public key.
3. Verify the `signer` matches the channel's `authorizedSigner`.
4. Verify `voucher.channelId` matches the active channel PDA, re-derived from the decoded channel open parameters — including `openSlot` — and the channel program ID, never taken from the JSON envelope alone. Because `openSlot` is a PDA seed, this address binding also pins the channel incarnation; no separate epoch-equality check exists.

5. Verify `cumulativeAmount > acceptedCumulative` using the server's durable watermark, even when on-chain `settled` lags. Equal or lower amounts **MUST** be rejected for metered voucher acceptance unless they are exact idempotent replays handled per "Concurrency and Idempotency". The accepted increment `cumulativeAmount - acceptedCumulative` **MUST** correspond to the resource cost charged for the accompanying request, not merely be a positive advance.
6. Verify the channel account still exists, is owned by the channel program, and carries the `Channel` discriminator. A fully drained channel is `Distributed` and then deallocated (see Section 6.2.10); its address never hosts another channel, because a new incarnation carries a new `openSlot` seed and therefore a new address.
7. Verify `status == Open` (i.e., `closureStartedAt == 0` and the channel has not yet been sealed). Servers **MUST** reject new voucher acceptance on channels with a pending forced close unless the voucher is being used only to drive `settleAndSeal`.
8. Verify `cumulativeAmount <= escrowedAmount` (does not exceed deposit).
9. If `expiresAt` is present and non-zero, verify `now < expiresAt` (with configurable clock skew tolerance).
10. Persist the new `acceptedCumulative` amount AND the full `SignedVoucher` to durable storage BEFORE serving the resource. The numeric watermark alone is insufficient: on-chain `settle` / `settleAndSeal` require the stored signed payload.

## 9.5. On-Chain Voucher Verification

When the channel program executes `settle` or `settleAndSeal` (with a voucher), the voucher signature **MUST** be verified on-chain. On Solana, this can be done by:

- Including an `ed25519` program instruction in the same transaction that verifies the signature immediately before the channel instruction executes.
- Or implementing Ed25519 verification directly in the channel program (higher compute cost).

The first approach is preferred as it uses Solana's native signature verification at minimal compute cost. The precompile instruction **MUST** immediately precede the channel instruction in the same transaction.

When using instruction introspection to consume a native signature-verification instruction, channel programs **MUST**:

- validate the Instructions sysvar account address;
- use checked instruction-loading helpers provided by the Solana SDK;
- decode the on-chain voucher payload directly from the verified message bytes recorded by the precompile in the same transaction (see Section 9.3); the `magic` prefix **MUST** match exactly (reference error `VoucherBadMagic`, code 238), the voucher `channelId` **MUST** equal the channel PDA address (reference error `VoucherChannelMismatch`, code 232) — an address binding that subsumes the epoch check, since `openSlot` is a PDA seed — and the precompile-recorded signer **MUST** equal `authorizedSigner`;

- reject signature-verification instructions that are replayed, unrelated, or positioned such that the channel program cannot unambiguously determine which verified message they authorize.

For the single-signature case, the canonical `ed25519` precompile instruction totals 162 bytes: a 112-byte prefix (header, public key, and signature) followed by the 50-byte voucher message. Its `message_data_size` **MUST** be exactly 50.

# 10. Distribution Splits

Channels **MAY** commit a multi-recipient split of the merchant-side pool at `open`. The split is a list of `(recipient, shareBps)` entries; the payee receives the implicit-remainder share `10000 − Σ shareBps` and is NOT listed explicitly.

## 10.1. Canonical Preimage

The byte layout hashed at `open` and re-hashed at `distribute`:

```
count (u32 LE) || [ recipient (32 bytes) || shareBps (u16 LE) ] × count
```

- `count == 0` is legal; the payee receives 100% of the pool.
- Every active entry **MUST** have `shareBps > 0`.
- `0 ≤ Σ shareBps ≤ 10000`.
- Recipients **MUST** be unique and **MUST NOT** equal the channel PDA itself.
- The list size is bounded by an implementation-defined `MAX_DISTRIBUTION_RECIPIENTS` (**RECOMMENDED**: 32).

## 10.2. Hash Algorithm

Implementations **MUST** use a collision-resistant hash with a 32-byte digest. The chosen algorithm **MUST** be fixed at deployment and documented for clients so they can reproduce it. SHA-256 is **RECOMMENDED**; the specific hash implementation (e.g., the `sol_sha256` syscall versus a bundled library) is an implementation detail that does not affect wire compatibility.

## 10.3. Distribution Math

`distribute` pays each beneficiary the cumulative floor delta between `payoutWatermark` and `settled`:

- recipient `i`: `floor(settled * shareBps[i] / 10000) − floor(payoutWatermark * shareBps[i] / 10000)`;
- payee: `floor(settled * (10000 − Σ shareBps) / 10000) − floor(payoutWatermark * (10000 − Σ shareBps) / 10000)`.

During `status == Open`, flooring-residual dust remains in the escrow ATA while `payoutWatermark` advances to `settled`; because later distributions compute deltas from that watermark, previously residual value stays claimable once a share's cumulative entitlement crosses the next whole unit. At the `Sealed` branch of `distribute`, the final cumulative delta runs once, then the irreducible residual dust is swept to the protocol treasury ATA before the escrow ATA is closed. The treasury account is a deployment-level address documented out of band by the channel program.

## 11. Authorized Signer

By default, the payer signs vouchers directly. This matches the default channel model: the funding key is also the voucher-signing key, and the deposit is the hard cap enforced by the channel.

Whether the voucher signer is the payer or a delegated key, it **MUST** be a valid Ed25519 public-key point. `open` **MUST** reject an `authorizedSigner` that is not a curve point, since a non-curve value could never produce a verifiable voucher signature.

Implementations **MAY** support delegated signing where the payer authorizes a separate keypair (for example, a session key) to sign vouchers on their behalf. The `authorizedSigner` field in the channel state records the delegated public key. The server verifies vouchers against this key instead of the payer's.

This enables use cases like browser sessions where an ephemeral key signs vouchers without repeated wallet confirmations.

That client-controlled delegated voucher key still uses a `voucherSigner` value of `client`. With `voucherSigner` set to `operator`, `authorizedSigner` is the operator and the client authenticates requests with the reusable payer proof.

Implementations **MAY** additionally support delegated signers on other curves that Solana can verify through native programs, such as `secp256r1` for passkeys. Such extensions **MUST** define:

- a distinct `signatureType` value;
- the exact signed message format;
- the exact Solana verification program used on-chain; and
- how the delegated signer is bound into the channel's PDA derivation and open transaction.

## 12. Fee Sponsorship

When `feePayer` is `true` in the challenge:

- **Open**: The client builds the open transaction with the server's `feePayerKey` as fee payer, partially signs (deposit transfer authority only), and sends via `transaction` in the open credential. The server co-signs and broadcasts. The server's `feePayerKey` is also the `open` instruction's `rentPayer` (account index 1) and is recorded into `Channel.rentPayer`: the

same operator signature covers the fee-payer and `rentPayer` signer roles, so the operator funds both the transaction fee and the channel PDA + escrow ATA rent. The operator recovers that SOL rent after close — through the terminal `distribute`'s fast path or its periodic `reclaim` sweep (see Section 6.2.10); the epoch window floats only this rent, never the merchant payout or the payer refund.

- **TopUp**: Same pattern — client partially signs, server co-signs.
- **Settle/Close**: The server initiates these operations and always pays the fee.

This ensures clients never need SOL — neither for transaction fees nor for channel rent — during the entire session lifecycle; the client transacts in stablecoin only.

# 13. Server State Management

## 13.1. Per-Channel State

The server **MUST** maintain the following state for each open channel:

| Field | Description |
|---|---|
| `channelId` | Channel account address |
| `openSlot` | On-chain `Channel.openSlot` of the channel being metered; a PDA seed, needed to re-derive and validate `channelId` and to anticipate the reclaim gate |
| `status` | `"open"` or `"closed"` |
| `payer` | Payer public key |
| `voucherSigner` | `client` or `operator` |
| `authorizationPolicy` | Voucher signer policy |
| `authentication` | For `operator`, secure state sufficient to verify the proof, its opening session challenge, payer, and channel |
| `escrowedAmount` | Total deposited (from on-chain `Channel.deposit`) |
| `acceptedCumulative` | Highest voucher amount accepted |
| `highestVoucher` | Full highest accepted `SignedVoucher`, retained for on-chain settlement |
| `spentAmount` | Cumulative amount charged for delivered service |
| `settledOnChain` | Highest cumulative amount already settled on-chain |

| Field | Description |
|---|---|
| `closureStartedAt` | Pending forced-close timestamp, if any |
| `lastActivityAt` | Server timestamp of the most recent channel activity defined in Section 14.6 |
| `idleTimeoutSeconds` | Effective negotiated inactivity threshold |

*Table 19*

Server-side channel state — in particular `acceptedCumulative` and the stored highest `SignedVoucher` — **MUST** be keyed by `channelId`, not by challenge id or HTTP session id. Because `openSlot` is a PDA seed, the address is already per-incarnation: reopening a closed relationship is legal by design (see Section 6.2.10) and produces a new channel at a new address with its own ledger, so `channelId` alone cannot conflate incarnations and no `(channelId, openSlot)` composite key is needed.

The channel program does not bind vouchers to a cluster, so operators **MUST** pin each server and channel to a single cluster and RPC endpoint and **MUST NOT** share one metering ledger across clusters. A server **SHOULD** verify the resolved channel matches the challenge's `methodDetails.network` before metering.

For `client`, the available off-chain balance is:

```
available = acceptedCumulative - spentAmount
```

For `operator`, the operator creates authorization as service is metered, so the available capacity is:

```
available = escrowedAmount - acceptedCumulative
```

The on-chain settlement watermark is distinct:

```
unsettled = spentAmount - settledOnChain
```

## 13.2. Mint Allow-List

Servers **MUST** restrict a channel's `mint` to an explicit, server-controlled allow-list of vetted mints, curated out of band and never derived from client-supplied data. The server **MUST** set the 402 challenge `currency` only to an allow-listed mint and **MUST** reject any `open` whose decoded `mint` is not on the list. Because the open-validation binding in Section 14.1 ties the decoded `open` mint to the challenged `currency`, no off-list mint can enter a new channel.

The server **SHOULD** refuse to resume or `topUp` a channel whose `mint` has since been delisted.

This requirement exists because the channel program does not inspect a mint's freeze or mint authority (see Section 17.3); the server is the only gate that keeps unvetted mints out of channels.

## 13.3. Debit Processing

For each request on an open channel:

1. Authenticate `client` mode with a valid voucher, or `operator` mode with a valid `use` proof.
2. Compute `cost` from the challenged `amount`, `unitType`, and any implementation-specific metering policy.
3. Compute `available` using the formula for the channel's mode.
4. If `available < cost`, return 402 requesting a new voucher or `topUp`, as applicable.
5. For `client`, persist `spentAmount += cost`. For `operator`, sign a voucher for `acceptedCumulative + cost` and atomically persist that voucher, the new `acceptedCumulative`, and `spentAmount += cost`.
6. Persist the applicable transition atomically before releasing the corresponding response bytes.
7. Serve the resource with a receipt.

When cost is known only after generation begins, an operator-mode server **MUST** reserve sufficient channel capacity before delivery and persist the operator-signed voucher before releasing the response bytes whose cost it authorizes. Failed delivery **MUST** release unused reservations. Amounts already covered by a persisted voucher and released response bytes remain committed; unused reserved capacity **MUST NOT** advance the voucher.

## 13.4. Partial Settlement

The server **MAY** call the channel program's settle instruction at any time to claim accumulated funds without closing the channel. This is useful for:

- Reducing counterparty risk on long-running sessions
- Freeing up server working capital
- Periodic reconciliation

After settlement, the channel account's `settled` field on-chain reflects the claimed amount. The server **MUST** update `settledOnChain` after confirmation and continues accepting vouchers for amounts above the new settled baseline.

## 13.5. Crash Safety

Servers **MUST** persist metering state increments BEFORE delivering the response. Servers **SHOULD** support idempotency keys for exactly-once delivery. More precisely, servers **MUST** persist both:

- `acceptedCumulative` BEFORE relying on new voucher balance; and
- `spentAmount` BEFORE or atomically with delivering the metered service.

Servers **SHOULD** use transactional storage or write-ahead logging to ensure recovery after process or machine crashes.

### 13.6. Concurrency and Idempotency

Servers **MUST** serialize proof verification, voucher creation or acceptance, and debit processing per channel (`channelId`; the address is per-incarnation by construction). Voucher updates arriving on different HTTP connections or multiplexed streams **MUST** be processed atomically with respect to:

- `acceptedCumulative`;
- `spentAmount`; and
- `closureStartedAt`.

Servers **MUST** treat metered requests idempotently:

- Replaying an already processed request **MAY** return the cached receipt and **MUST NOT** change channel state or deliver additional service.
- Voucher submissions with `cumulativeAmount <= acceptedCumulative` and no matching cached idempotent response **MUST** be rejected and **MUST NOT** reduce channel state.
- Clients **MAY** safely retry voucher submissions after network failures using the same idempotency key.

Clients **SHOULD** include an `Idempotency-Key` header on metered HTTP requests. Servers **SHOULD** cache `(challengeId, idempotencyKey)` pairs and **MUST NOT** increment `spentAmount` twice for a duplicate idempotent request.

Repeated presentation of a session bearer proof is expected and is not, by itself, a duplicate request. Idempotency keys and atomic channel accounting provide request-level replay protection independently of the bearer proof.

## 14. Settlement Procedure

### 14.1. Open

1. Decode the open transaction before signing, paying fees, or broadcasting. Verify it contains the expected channel program instruction and that the instruction uses the `open` discriminator (the reference implementation composes channel-PDA creation, escrow ATA creation, deposit transfer, and the `distributionHash` commitment in a single instruction).
2. Verify the instruction targets the challenged channel program and encodes the challenged `payer`, `payee`, `mint`, `authorizedSigner`, `salt`, `deposit`, `grace_period`, `open_slot`, and canonical `distributionSplits` preimage. The decoded `authorizedSigner` **MUST** equal the credential's `authorizedSigner` and **MUST** be a valid Ed25519 public-key point; reject non-curve values.

3. Recompute the expected PDA from the decoded payer, payee, mint, authorized signer, salt, and `open_slot` plus the channel program ID. Verify it equals both the decoded channel account and the declared `channelId`.
4. Verify the decoded escrow account is the associated token account for `(channelId, mint, tokenProgram)`. If the challenge supplied `tokenProgram`, the decoded token program **MUST** match it; otherwise it **MUST** be a supported token program for the mint.
5. Verify the credential's `gracePeriodSeconds` equals the challenge policy and is greater than zero. Decode the open instruction and verify its `grace_period` equals the same value.
6. Resolve the effective idle timeout as defined in Section 14.6. If the credential contains `idleTimeoutSeconds`, verify that the challenge advertises `idleTimeoutOptionsSeconds` and that the selected value exactly matches one of its values. Reject an unsupported selection before signing, paying fees, or broadcasting. The server **MUST NOT** silently clamp or replace the client's value. If the credential omits the field, select an offered value when options were advertised, or select any value from 1 through 2592000 seconds at the server's discretion when they were not.
7. Verify the credential's `openSlot` equals the decoded `open_slot` and satisfies the open-slot window relative to both the challenged `recentSlot` (`openSlot <= recentSlot` and `recentSlot - openSlot <= OPEN_SLOT_WINDOW`) and the window against the server's current view of the cluster slot (`openSlot <= slot` and `slot - openSlot <= OPEN_SLOT_WINDOW`); the program enforces the same rule at execution (see Section 6.2.10).
8. Verify the transaction's fee payer matches the challenge policy:
   - if `feePayer` is `true`, the fee payer **MUST** equal `feePayerKey`;
   - otherwise the payer funds the transaction.

   Verifiers that read the open accounts by fixed index **MUST** account for the `rentPayer` account at index 1 (`payer=0`, `rentPayer=1`, `payee=2`, `mint=3`, `authorizedSigner=4`, `channel=5`, with every account after `payer` shifted by `+1`) and **MUST** verify that `accounts[1]` equals the operator / fee-payer key. `rentPayer` is derived from the existing operator / fee payer and carries no separate wire field; a single operator signature satisfies both the fee-payer and `rentPayer` signer roles.
9. Validate the complete compiled message — resolving any version-0 address-lookup-table entries — not just the channel instruction. Verify the transaction does use the challenged `recentBlockhash` and does not include unrelated writable accounts or instructions that could redirect funds or mutate channel parameters, and that the server fee payer is never used as an authority, source, or writable account by any instruction. The server **SHOULD** reject transactions that route value through unexpected external programs.
10. Verify the decoded `deposit` equals `depositAmount`, satisfies `methodDetails.minimumDeposit` (when set), and that the resulting `distributionHash` matches the digest of the canonical preimage of the splits proposed in the 402 challenge.
11. Reject any disagreement between the challenge, credential payload, decoded transaction, derived PDA, escrow ATA, or token program.

12. Verify `voucherSigner`. For `operator`, verify the reusable proof as defined in Section 8.1 and verify that decoded `authorizedSigner` equals the challenged `operator`. For `client`, reject an `authentication` field.
13. If fee payer mode: co-sign and broadcast. Otherwise: broadcast as-is.
14. Verify channel state on-chain after confirmation: - payer matches transaction signer; - payee matches the challenged recipient; - mint matches the challenge currency; - deposit matches the requested amount; - `gracePeriod` is non-zero and matches the challenge policy; - `openSlot` equals the credential's `openSlot`; - authorized signer matches the open parameters; - `distributionHash` matches the proposed splits; - `rentPayer` equals the operator / fee-payer key that funded the channel rent; - channel is not sealed; and - `closureStartedAt` is `0`.
15. Create server-side channel state keyed by `channelId` (per-incarnation by construction, since `openSlot` is a PDA seed). Persist the effective idle timeout, voucher signer and any session proof verifier. Initialize `lastActivityAt` as part of the same state transition.
16. Return 200 with receipt.

## 14.2. Resume

When a challenge resumes an existing channel (`methodDetails.channelId`), the server **MUST** re-authenticate the on-chain account before metering against it — decoding the account bytes is not sufficient. The server **MUST** verify the account is owned by the channel program and that its discriminator, `version`, `status == Open`, PDA derivation (re-derived over the stored open parameters, including `openSlot`), `mint` (still allow-listed), `payee`, `authorizedSigner`, `openSlot`, and `distributionHash` all match the active challenge and session. The challenged `idleTimeoutSeconds` **MUST** equal the effective value stored for the channel. Resume only ever applies to a live channel: because `openSlot` is a PDA seed, a deallocated address is never reoccupied — reopening a closed relationship creates a new channel at a new `channelId` with a fresh ledger. A resumed channel shares one cumulative ledger across challenges, keyed by `channelId`, so a single cumulative voucher cannot be reused to buy multiple responses.

An operator-signed channel **MUST NOT** resume if its durable proof binding is missing or inconsistent with its payer, original session challenge, or channel ID. Public transaction or account data **MUST NOT** be used to reconstruct that binding.

## 14.3. Voucher Update (No Settlement)

This procedure directly authorizes service only for `client`.

1. Verify voucher signature and monotonicity.
2. Verify the channel is open and has no pending forced close.
3. Persist `acceptedCumulative`.
4. Debit `cost` from available balance by persisting `spentAmount`.
5. Return 200 with receipt.

### 14.4. TopUp

1. If fee payer mode: co-sign and broadcast. Otherwise: broadcast as-is.
2. Verify the top-up transaction targets the expected channel PDA and channel program and only increases deposit for that channel.
3. Verify the on-chain deposit increase after confirmation.
4. Increase `escrowedAmount` in server-side state.
5. Return 200 with receipt.

`topUp` is callable only while `status == Open` and **MUST NOT** clear `closureStartedAt`. Once forced close is requested, the paths forward are `settleAndSeal` (within grace) or `seal` (after grace).

### 14.5. Close (Cooperative)

1. Authenticate close according to `voucherSigner`. For `client`, verify the required voucher's channel, signer, signature, freshness, and `cumulativeAmount <= deposit`. For `operator`, verify the bound session proof.
2. Select the settlement voucher. In `client` mode, use the supplied voucher only when `cumulativeAmount > settled`; otherwise treat it only as close authentication after verifying it equals `acceptedCumulative`. In `operator` mode, use the stored highest voucher when it advances `settled`. If no voucher advances settlement, seal at the current on-chain watermark.
3. Build and immediately broadcast `settleAndSeal` bundled with `distribute` in the same transaction, so the merchant-side payout, payer refund, treasury sweep, and escrow-ATA closure all land atomically. The bundle is never slot-gated and **MUST NOT** be deferred waiting for the epoch window. When `clock.slot > openSlot + OPEN_SLOT_WINDOW` already holds, the same `distribute` also deallocates the channel PDA in place; otherwise the channel is left `Distributed` and the operator's periodic `reclaim` sweep recovers the PDA rent after the window (see Section 6.2.10). A bundle whose `distribute` carries many recipients may require a version-0 transaction with an address lookup table.
4. Mark the channel as `"closed"` in server-side state.
5. Persist final `settledOnChain` and terminal accounting state after confirmation.
6. Return 200 with receipt containing `txHash` and (if `distribute` ran) the refunded amount.

For deployments whose `payee` is a PDA, the server **MUST** provide a working CPI signer-seed adapter for `settleAndSeal` before opening channels, or else refuse the channel before metering begins. A PDA payee with no cooperative-close path can leave delivered service uncollectible: the permissionless `settle` crank cannot apply a new voucher once `requestClose` has moved the channel to `Closing`.

### 14.6. Server-Initiated Idle Close

For a new channel, the server advertises the supported values in `methodDetails.idleTimeoutOptionsSeconds`. If this field is absent, the server does not offer client selection and chooses the effective timeout at its discretion within the range from 1 through 2592000 seconds, inclusive.

For example, a server can offer 30 seconds, 10 minutes, and 1 day:

```
{
  "idleTimeoutOptionsSeconds": [30, 600, 86400]
}
```

The client selects an offered value by including `idleTimeoutSeconds` in its open credential. The client **MUST NOT** include this field when the challenge omits `idleTimeoutOptionsSeconds`. If the client omits its selection, the server selects an offered value when options were advertised, or selects any value in the permitted range at its discretion when they were not.

A client-selected value **MUST** exactly match an offered value. The server **MUST** reject an unsupported selection and **MUST NOT** silently clamp, replace, or otherwise reinterpret it. A successful open commits the server to the effective timeout for the lifetime of that channel.

A challenge that resumes a channel **MUST** include the effective value as `methodDetails.idleTimeoutSeconds`, so the client can recover the negotiated policy without retaining the original open exchange. A resumed channel does not renegotiate its idle timeout.

For this policy, channel activity is any of the following events that completes successfully:

- opening or resuming the channel;
- accepting a new voucher;
- confirming a top-up; or
- delivering a billable unit of service.

Rejected credentials, failed transactions, and replayed requests that return a cached response are not channel activity. Servers **MUST** update `lastActivityAt` atomically with the state change or service delivery that constitutes channel activity.

When no channel activity has occurred for the effective idle timeout, the server **SHOULD** initiate cooperative close promptly. It **MUST NOT** close a channel solely for inactivity before that interval has elapsed. Other terminal conditions, including operator shutdown or security intervention, **MAY** close a channel earlier; clients therefore **MUST NOT** treat the idle timeout as a guarantee that the channel remains available until the threshold.

Before closing an idle channel, the server **MUST** atomically stop voucher acceptance, top-up processing, debit processing, and paid-service delivery for that channel. It **MUST** serialize the close against those operations and then follow Section 14.5, using the stored highest accepted

voucher when its cumulative amount exceeds the on-chain settlement watermark. The mechanism that gates work while close is in progress is implementation-specific and does not define another server-side or on-chain channel status. A later client request for the closed channel receives the normal terminal-session error and a fresh challenge.

## 14.7. Forced Close (Client-Initiated)

If the server becomes unresponsive, the client can force-close the channel:

1. The payer authorizes `requestClose` and submits it directly to RPC. Because the operator funds channel rent and the client transacts in stablecoin only, a SOL-free client cannot pay the transaction fee for this escape route on its own; such a client **MUST** obtain SOL (or a fee-paying submitter) to drive `requestClose`, while `seal` and `distribute` are permissionless and **MAY** be cranked by any party.
2. Grace period begins (per-channel `gracePeriod`).
3. During the grace period, the server **MAY** still call `settleAndSeal` with the latest voucher.
4. After the grace period, any party submits `seal` (permissionless) to transition the channel to `Sealed`.
5. The payer **MAY** submit `withdrawPayer` to recover `deposit - settled` immediately. Independently, any party **MAY** submit `distribute` with the splits preimage, also immediately — no token movement is slot-gated: the merchant side is paid, any pending payer refund is also paid, residual is swept to treasury, and the escrow ATA is closed. The channel PDA is deallocated in the same instruction when the epoch window has already elapsed, or left `Distributed` for a later permissionless `reclaim`; in both cases all freed SOL goes to `rentPayer` (see Section 6.2.10).

## 14.8. One-Shot Session Example

A metered API prices one request at 16 base units. A client expecting to make a single call can still use a session channel with no permanent on-chain storage cost:

1. At build time the cluster slot is `S`. The client opens with `openSlot = S − 1400`, leaving a 100-slot landing window (`OPEN_SLOT_WINDOW = 1500`) while shortening the operator's post-close rent float, and `depositAmount = "50000"`. Because `openSlot` is a PDA seed, the client derives the channel address from its chosen value up front and pre-signs the voucher `{channelId, cumulativeAmount: "16"}` before the open transaction confirms.
2. The client sends the metered request with the voucher; the server verifies it per Section 9.4 and serves the resource.
3. The client sends `Action: "close"` with the same voucher as the final voucher; the server immediately broadcasts `settleAndSeal` bundled with `distribute`. In that one transaction, seconds after the open, the merchant side receives 16 and the payer is refunded 49984. The window has not yet elapsed, so the drained channel PDA is left `Distributed`.

4. The reclaim gate unlocks at slot `openSlot + 1501 = S + 101`, roughly 100 slots (~40 seconds) after the open landed. The operator's next periodic `reclaim` sweep deallocates the PDA and returns 100% of the channel rent to `rentPayer`. Nobody's payout or refund waited on the window — it floated only the operator's rent.

## 15. Receipt Format

Receipts are returned in the `Payment-Receipt` header.

| Field | Type | Required | Description |
|---|---|---|---|
| `method` | string | **REQUIRED** | `"solana"` |
| `intent` | string | **REQUIRED** | `"session"` |
| `reference` | string | **REQUIRED** | Channel identifier |
| `status` | string | **REQUIRED** | `"success"` |
| `timestamp` | string | **REQUIRED** | RFC 3339 timestamp |
| `challengeId` | string | **OPTIONAL** | Challenge identifier for audit correlation |
| `acceptedCumulative` | string | **REQUIRED** | Highest voucher amount accepted |
| `spent` | string | **REQUIRED** | Total amount charged so far |
| `idleTimeoutSeconds` | integer | **REQUIRED** | Effective negotiated inactivity threshold for the channel |

*Table 20*

For close actions, the receipt **MAY** additionally include:

| Field | Type | Description |
|---|---|---|
| `txHash` | string | Settlement transaction signature |
| `spent` | string | Total amount settled |
| `refunded` | string | Amount refunded to client |

*Table 21*

For streaming responses, servers **SHOULD** include the receipt in the initial response headers and **SHOULD** emit a final receipt when the stream completes. When balance is exhausted mid-stream, servers **SHOULD** pause delivery and request a higher voucher or top-up rather than serving beyond the authorized balance.

### 15.1. Voucher Submission Transport

Voucher updates and top-up requests **SHOULD** be submitted to the same resource URI that requires payment. This allows session payment to compose with arbitrary protected endpoints without a dedicated payment control plane route.

Operator-mode clients submit `use` credentials to the same URI with `method="solana"` and `intent="session"`.

Clients **MAY** use `HEAD` for voucher-only or top-up-only requests when no response body is required. Servers **SHOULD** support such requests where practical.

## 16. Error Responses

Servers **MUST** use the standard problem types defined in [I-D.ryan-httpauth-payment-01]: `malformed-credential`, `invalid-challenge`, and `verification-failed`. The `detail` field **SHOULD** describe the specific failure (e.g., "Amount exceeds deposit", "Channel not found").

All error responses **MUST** include a fresh challenge in `WWW-Authenticate`.

When a server rejects a client-selected `idleTimeoutSeconds` that was not offered in the challenge, it **MUST** use the `verification-failed` problem type. Its `detail` **SHOULD** state that the requested idle timeout was not one of the advertised options. The fresh challenge conveys the currently supported options.

## 17. Security Considerations

### 17.1. Transport Security

All communication **MUST** use TLS 1.2 or higher.

### 17.2. Session Bearer Proofs

The channel-open transaction and account state are public. They prove that the payer authorized the deposit and settlement signer, but not that a later HTTP caller is the payer. Servers **MUST NOT** authenticate a metered request using a transaction signature, serialized open transaction, open credential, channel ID, or another value recoverable from the ledger.

The session proof is not placed on-chain, but it is a bearer credential: anyone who obtains it can request service against the channel's remaining capacity. Clients and servers **MUST** keep it confidential, transmit it only over TLS, and redact the proof, signature, and Authorization header from logs and traces. Servers **MUST** reject it once close begins or the channel becomes terminal. Temporary insufficient capacity does not invalidate the proof; a successful `topUp` can restore available capacity.

The proof is bound to one channel and opening session challenge, but not to an individual HTTP method, target, body, or idempotency key. Servers **MUST** enforce idempotency and atomic capacity reservation independently. Applications requiring sender-constrained or per-request integrity **MAY** define an extension that signs request context with a client-held key.

## 17.3. Escrow Safety

Funds are held by the channel program, not the server. The server can only claim funds by presenting valid voucher signatures to the program. The client can always recover unspent funds via forced close after the grace period.

The channel program intentionally does NOT inspect a mint's freeze authority or mint authority. Hard-rejecting any mint with a live freeze authority would exclude most real-world stablecoins (USDC, USDT, PYUSD, EURC), all of which retain an issuer-controlled freeze authority. The cost of allowing them is that a live freeze authority can freeze the escrow ATA at any point in the channel lifecycle; once frozen, every value-moving instruction (`topUp`, `distribute`, `withdrawPayer`) rejects, wedging both the merchant payout leg and the payer refund leg with no permissionless crank to unwind it until the authority thaws. The trust decision is therefore pushed off-chain: a merchant accepting payments in mint `M` implicitly accepts that `M`'s freeze authority can wedge any channel denominated in `M`, and **SHOULD** allow-list (see Section 13.2) only mints whose freeze and mint authorities it considers acceptably governed. This mint-issuer trust model is distinct from the Token-2022 *extension* allow-list in Section 17.13.

## 17.4. Payout Forfeiture

`distribute` never blocks on an unusable beneficiary account. A nonzero share whose canonical ATA is missing, frozen, closed, malformed, carries an unsupported Token-2022 account extension, or has a reassigned authority is redirected to the treasury ATA and `payoutWatermark` advances regardless, so the beneficiary permanently forfeits that share — later repair cannot reclaim it. The same applies to the payer refund ATA at `Sealed`. This removes a griefing vector (a single poisoned ATA cannot stall payouts to the rest of the channel) at the cost of forfeitable funds. Operators **SHOULD** ensure recipient, payee, and payer ATAs exist and are healthy (initialized, unfrozen, canonical, extension-clean) — or withdraw the payer headroom via `withdrawPayer` beforehand — before cranking `distribute`.

## 17.5. Voucher Replay Protection

Vouchers are bound to a specific channel incarnation via `channelId` — the address is per-incarnation, because `openSlot` is a PDA seed — and ordered by `cumulativeAmount`; there is no per-voucher nonce. A voucher from one channel cannot be replayed in another, and a voucher from a closed channel cannot be replayed against a reopened relationship, whose channel lives at a different address (see Section 17.6).

This replay protection depends on deterministic PDA derivation. The channel address **MUST** be bound to the channel program ID and channel open parameters so that vouchers cannot be replayed across different channel program deployments.

Vouchers are not bound to a cluster; the same program and seeds derive an identically-addressed channel on another cluster, so a voucher could in principle be replayed there. This residual cross-cluster replay is an accepted operational risk, mitigated off-chain by pinning each server and channel to a single cluster.

## 17.6. Reincarnation Replay

Terminal closure ends with full deallocation of the channel PDA — by `distribute`'s fast path or by `reclaim` (see Section 6.2.10) — and the same participant relationship can legally be reopened. Replay protection across incarnations is address binding by construction: `openSlot` is a PDA seed, so every incarnation derives its own address, every voucher signs over `channelId`, and the program only accepts vouchers whose `channelId` equals the live channel's address.

The channel address never repeats. An address encodes one fixed `openSlot` in its seeds, so open can only ever derive it while `clock.slot - openSlot <= OPEN_SLOT_WINDOW`. Incarnation N's address stays occupied — live, then `Distributed` — until it is deallocated (by `distribute`'s fast path or by `reclaim`) at some slot `C > openSlot_N + OPEN_SLOT_WINDOW`; from `C` onward the open window has permanently closed over `openSlot_N`, so no second channel can ever exist at that address, for any client behavior inside the window — including adversarial choices of `openSlot`. An old voucher can never match a later incarnation: the later incarnation lives at a different address, so a stale voucher fails the address binding (wrong `channelId`) or targets a deallocated account.

Two rules keep this argument sound:

- `OPEN_SLOT_WINDOW` is consensus-critical and **MAY** only ever be decreased in future program versions. Increasing it would re-arm the addresses of channels deallocated under the smaller window: an address whose `openSlot` was already too stale to re-derive could become derivable again, allowing a second channel — and the old vouchers — to land at it.
- Future `openSlot` values **MUST** be strictly rejected at `open`. Beyond breaking the inequality above, a far-future `openSlot` would push the reclaim gate arbitrarily far out, permanently stranding the operator's PDA rent.

## 17.7. Open Transaction Binding

Servers that sponsor or submit open transactions **MUST** treat the decoded transaction contents as the committed request. A malicious client can otherwise present a benign HTTP envelope while embedding a different payee, distribution split, deposit, signer, channel PDA, or grace period. Such a mismatch can make the server sponsor or meter a channel it did not challenge.

## 17.8. Cumulative Amount Safety

Vouchers authorize cumulative totals (not deltas). A compromised voucher only authorizes up to its stated amount. The channel program enforces that settlements never exceed the deposit.

## 17.9. Grace Period Security

The grace period prevents a race condition where the payer withdraws before the server can settle. Without it, a malicious payer could use the service, then immediately withdraw. The server has the grace period to submit any outstanding vouchers.

Servers **MUST** verify that a new channel uses the challenged `gracePeriodSeconds`. If the transaction sets a zero, shorter, or envelope-disagreeing `grace_period`, the payer could request close and recover funds before the server has time to settle accepted vouchers.

Because `topUp` **MUST NOT** clear `closureStartedAt`, servers **MUST** guard the equivalent grief vector at the HTTP layer by rate-limiting `requestClose` retries and refusing to extend service after a forced-close broadcast.

Servers **MUST** stop accepting new service vouchers once `closureStartedAt` is set. During the grace period, the server **MAY** use the latest previously accepted voucher to drive `settleAndSeal` (and, optionally, `distribute`). Servers **MUST NOT** resume metered service after `closureStartedAt` is set.

## 17.10. Delegated Signer Risks

If client-controlled delegated voucher signing is used, a compromised delegated key can authorize spend up to the delegation's limit. The `authorizedSigner` is bound into the PDA seed set at open time and cannot be changed without closing and reopening the channel. If a delegated signing key is compromised, the payer's only recourse is to call `requestClose`, but the attacker retains the ability to sign vouchers up to the full deposit cap throughout the entire grace period before funds can be recovered. Implementations **MUST** treat delegated keys as short-lived, single-session credentials with TTLs on the order of minutes to bound exposure in the event of a key compromise.

With `voucherSigner` set to `operator`, compromise of `operator` permits voucher creation up to the deposit cap. Theft of the reusable proof does not permit direct voucher creation, but lets the thief request service that the operator can charge to the channel. Replacing a compromised proof requires closing and reopening the channel.

## 17.11. Channel Program Trust

Clients **MUST** verify the `methodDetails.channelProgram` in the challenge matches a known, audited program before depositing funds. A malicious server could specify a program that steals deposits.

## 17.12. CPI and Program-ID Validation

Channel programs frequently rely on external Solana programs, including the System Program, SPL Token or Token-2022, Associated Token Program, and native signature-verification programs. Implementations **MUST** validate every external program account used in CPI against the expected canonical program ID before invocation. Implementations **MUST NOT** allow user-controlled program accounts to influence escrow, settlement, refund, or signature-verification CPIs.

If multiple token-program variants are supported, implementations **MUST** bind the chosen token-program variant into channel creation and subsequent account validation. A channel opened for one token-program variant **MUST NOT** be settled or refunded through a different token-program account.

## 17.13. Token-2022 Extension Policy

Implementations **MUST** enforce a closed allow-list of permitted Token-2022 extensions at `open` and re-validate it on every token-touching instruction. Extension presence alone is disqualifying; unlisted, unknown, or malformed extensions **MUST** be rejected before any token movement.

The **RECOMMENDED** mint allow-list:

- `MetadataPointer`
- `TokenMetadata`
- `GroupPointer`
- `TokenGroup`
- `GroupMemberPointer`
- `TokenGroupMember`

The **RECOMMENDED** token-account allow-list:

- `ImmutableOwner`

All other extensions **MUST** be rejected:

| Extension | Reason |
|---|---|
| `NonTransferable` | No transfer from escrow can succeed |
| `PermanentDelegate` | Delegate can move escrow arbitrarily |
| `DefaultAccountState` | Destination ATAs may be born non-`Initialized` |
| `ConfidentialTransferMint` | Channel program does not produce confidential-transfer proofs |
| `TransferFeeConfig` | Withheld fees desync `deposit` / `settled` from escrow |

| Extension | Reason |
|---|---|
| `TransferHook` | Hook program can revert any transfer |
| `InterestBearing` | Visible amount changes over time |
| `ScaledUiAmountConfig` | Display-vs-raw divergence breaks exact distribution |
| `Pausable` | Mint-level pause can block escrow release |
| `CpiGuard` / `MemoTransfer` (account) | Distribution CPIs use neither delegate flow nor memos |
| `MintCloseAuthority` | Mint identity can be recreated while channels reference it |

*Table 22*

Implementations **MUST NOT** resolve transfer-hook extra accounts, route through fee withholding, or honor pause flags.

## 17.14. Account Ownership Validation

Before deserializing or mutating any account, implementations **MUST** validate the expected owner for:

- the channel PDA account;
- any escrow SOL or token-holding account;
- any mint account referenced by the channel; and
- any payer or payee token account used for settlement or refund.

Servers performing off-chain verification **SHOULD** also verify account ownership and program ownership against RPC state before accepting an open, top-up, settle, or close flow as valid.

## 17.15. Channel Exhaustion

A malicious client could open many channels with small deposits. The on-chain storage cost is transient: closure closes the escrow ATA immediately and deallocates the channel PDA once the epoch window has passed (fast path or `reclaim`), and the operator (`rentPayer`) recovers all of the rent it fronted (see Section 6.2.10), so channel spam does not strand rent permanently. It does, however, tie up operator SOL while channels stay open — plus a per-channel PDA-rent float of roughly 2.7 million lamports for up to the window after close — and consume server resources.

Servers **SHOULD** therefore still enforce a minimum economically useful deposit to avoid channel spam with balances too small to justify signature verification and settlement overhead, and **SHOULD** apply the advertised idle timeout to recycle the rent float. Operators **MAY** use shorter advertised timeouts for low-value channels, but **MUST** keep the effective timeout stable after opening a channel as required by Section 14.6.

### 17.16. Denial of Service

To mitigate voucher flooding and channel griefing:

- servers **SHOULD** rate-limit voucher submissions per channel;
- servers **SHOULD** perform cheap format and monotonicity checks before expensive signature verification;
- servers **MAY** enforce a minimum voucher delta; and
- servers **SHOULD** refuse channels with prolonged inactivity or uneconomic deposit sizes.

### 17.17. Clock Skew

Voucher expiration depends on timestamp comparison. Servers **MUST** allow configurable clock skew tolerance (**RECOMMENDED**: 30 seconds).

### 17.18. Solana Verification Programs

This specification uses Solana-native verification primitives where possible. The base interoperable path is Ed25519, using either:

- an `ed25519` verification instruction in the same transaction as `settle` or `settleAndSeal`, with the channel program reading the Instructions sysvar to confirm success; or
- direct in-program verification if compute budget and implementation constraints permit.

Implementations that support delegated `secp256r1` passkey signers **SHOULD** use Solana's native `Secp256r1SigVerify1111111111111111111111111` verification program and **MUST** define a distinct `signatureType` and wire format for that extension.

## 18. IANA Considerations

### 18.1. Payment Intent Registration

The `session` intent is registered by [I-D.payment-intent-session]. This document does not register a new payment intent; it defines how the `solana` payment method implements the registered `session` intent.

## 19. References

### 19.1. Normative References

**[RFC2119]** Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

**[RFC3339]** Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

**[RFC4648]** Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

**[RFC8174]** Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

**[RFC8259]** Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

**[RFC8785]** Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/info/rfc8785>.

**[RFC9457]** Nottingham, M., Wilde, E., and S. Dalal, "Problem Details for HTTP APIs", RFC 9457, DOI 10.17487/RFC9457, July 2023, <https://www.rfc-editor.org/info/rfc9457>.

**[I-D.ryan-httpauth-payment-01]** Ryan, B. and J. Moxey, "The 'Payment' HTTP Authentication Scheme", January 2026, <https://datatracker.ietf.org/doc/draft-ryan-httpauth-payment/01/>.

**[I-D.payment-intent-session]** Ryan, B., Moxey, J., and T. Meagher, "Session Intent for HTTP Payment Authentication", June 2026, <https://datatracker.ietf.org/doc/draft-payment-intent-session/>.

### 19.2. Informative References

**[SOLANA-DOCS]** Solana Foundation, "Solana Documentation", 2026, <https://solana.com/docs>.

**[SPL-TOKEN]** Solana Foundation, "SPL Token Program", 2026, <https://solana.com/docs/tokens>.

**[BASE58]** Sporny, M., "Base58 Encoding Scheme", 2021, <https://datatracker.ietf.org/doc/html/draft-msporny-base58-03>.

## Appendix A. Acknowledgements

The authors thank the Tempo team for their input on this specification.

# Authors' Addresses

**Ludo Galabru**
Solana Foundation
Email: ludo.galabru@solana.org

**Jo**
Solana Foundation
Email: jo.desormeaux@solana.org

**Michael Assaf**
Moonsong Labs
Email: michael@moonsonglabs.com