Workgroup:	Network Working Group		
Internet-Draft:	draft-solana-session-00		
Published:	30 June 2026		
Intended Status:	Informational		
Expires:	1 January 2027		
Authors:	L. Galabru	Desormeaux	M. Assaf
	Solana Foundation	*Solana Foundation*	*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, authorizes incremental spend via signed vouchers, and 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 1 January 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 `settleAndFinalize` and `distribute` so the merchant payout, payer refund, treasury sweep, and PDA tombstone all land atomically.
- **Fee payer separation**: The server can sponsor the cooperative on-chain operations it submits (open, topUp, settle, settleAndFinalize, distribute) so the client never needs SOL for transaction fees during the normal session lifecycle. Escape-route instructions (requestClose, finalize, withdrawPayer) are client-submitted and self-funded.
- **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.

Step 11 typically bundles `settleAndFinalize` and `distribute` in the same transaction so the merchant payout, payer refund, treasury sweep, and PDA tombstone all land atomically.

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.

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.

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, and a salt) with the following logical fields. Field names use camelCase; tag and enum-variant values (`Channel`, `ClosedChannel`, `Open`, `Closing`, `Finalized`) 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 fields without colliding with `ClosedChannel`
`bump`	u8	Account state	Canonical PDA bump
`status`	u8	Account state	`Open` / `Closing` / `Finalized` enum value

Field	Type	Storage	Description
`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 `Finalized`)
`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 `finalize`. 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

Field	Type	Storage	Description
`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 `distribute` can reclaim the freed SOL rent to this account at `finalize` without an off-chain input. Distinct from `payer`: the client (`payer`) only moves stablecoin and never needs SOL

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; and
- the authorized signer public key (or payer if no delegation is used).

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.

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
`distributionSplits`	`(Pubkey, u16)[]`	Splits preimage; canonical encoding hashed into `distributionHash` (see Section 10)

Table 2

`open` **MUST** reject the instruction when the target PDA already exists with the `ClosedChannel` discriminator; reopening a previously finalized channel PDA is forbidden regardless of seed inputs. `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 17.11) **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 refund to `rentPayer` at tombstone, and surplus escrow tokens are swept to the treasury at `finalize`.

Servers **MUST** use a `salt` unique per channel. Reusing the `(payer, payee, mint, authorizedSigner, salt)` tuple of an already-open channel reverts `open` (the PDA already holds an initialized `Channel`); resume that channel instead of reopening it.

`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 (`channelId`, `cumulativeAmount`, `expiresAt`) from them (see Section 9.3), 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`.

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 3

`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. finalize

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

6.2.6. settleAndFinalize

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

Parameter	Type	Description
`hasVoucher`	u8	`0` finalizes 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 finalizing

Table 4

The payee **MUST** be a signer. Callable from `Open` and from `Closing` while `now < closureStartedAt + gracePeriod`; after the grace deadline use `finalize` 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 `settleAndFinalize` via CPI with signer seeds. The permissionless `settle`, `finalize`, 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 5

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 `Finalized`, `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, closes the escrow ATA, and tombstones the channel PDA (see Section 6.2.9). The freed SOL rent from closing the escrow ATA and tombstoning the PDA is reclaimed to `Channel.rentPayer` (the operator), not the payer; the token refund still goes to the payer. `distribute` **MUST NOT** be callable from `Closing`.

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 `Finalized` (finalization tombstones the channel in the same instruction, so there is no later crank to reclaim). Malformed token-account data and wrong (non-canonical) accounts hard-fail rather than redirecting.

6.2.8. withdrawPayer

One-shot payer refund in `Finalized` that does NOT tombstone the PDA. The program requires `status == Finalized` and `payerWithdrawnAt == 0`, transfers `deposit - settled` to the payer, and stamps `payerWithdrawnAt`. The payer **MUST** be a signer.

6.2.9. Tombstoning

The `Finalized` branch of `distribute` performs tombstoning. The program **MUST NOT** fully deallocate the channel account; it **MUST** realloc the account data to 1 byte and write the `ClosedChannel` discriminator at offset 0. The rent difference between the pre-tombstone balance and the 1-byte tombstone rent-exempt minimum **MUST** be returned to the channel's `rentPayer` (the operator that funded the rent at `open`), not the payer. `withdrawPayer` **MUST NOT** tombstone.

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 `settleAndFinalize` (or to bundle a voucher into `settleAndFinalize` directly).

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

6.4. Access Control

Instruction	Caller	Gating
open	Payer	Payer signs the deposit transfer

Instruction	Caller	Gating
settle	Anyone (permissionless crank)	Precompile-verified Ed25519 voucher from `authorizedSigner`
topUp	Payer	Payer signs the additional transfer; rejected when `status != Open`
requestClose	Payer	Payer signer equals channel `payer`
finalize	Anyone (permissionless crank)	`status == Closing` and elapsed grace period
settleAndFinalize	Payee	Payee signer equals channel `payee`
distribute	Anyone (permissionless crank)	On-chain hash commitment to splits preimage
withdrawPayer	Payer	Payer signer equals channel `payer` and `status == Finalized`

Table 6

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 17.3). 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 above the rent cost of the channel account plus a minimum economically useful balance to avoid spam.

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-beta", "devnet", "testnet", 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.

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.

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

ttlSeconds **OPTIONAL**. Suggested session duration in seconds.

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. Four 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`
Top-Up	`topUp`
Close	`close`

Table 7

8.1. Action: "open"

Opens a new payment channel.

Field	Type	Required	Description
`action`	string	**REQUIRED**	`"open"`

Field	Type	Required	Description
`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)
`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`
`distributionSplits`	array	**OPTIONAL**	Splits preimage (see the challenge's `methodDetails.distributionSplits`); **MUST** byte-match the splits proposed in the 402 challenge
`authorizationPolicy`	object	**OPTIONAL**	Voucher signer policy. When present, **MUST** be consistent with `authorizedSigner`
`transaction`	string	**REQUIRED**	Base64-encoded (standard alphabet, padded) signed or partially signed transaction
`capabilities`	object	**OPTIONAL**	Implementation-specific extensions

Table 8

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.

`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.

Example open credential:

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

8.2. 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 9

This action is entirely off-chain. No transaction is broadcast.

8.3. 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 10

8.4. Action: "close"

Requests cooperative close.

Field	Type	Required	Description
`action`	string	**REQUIRED**	`"close"`
`channelId`	string	**REQUIRED**	Existing channel identifier
`voucher`	object	**OPTIONAL**	Final signed voucher (see Section 9)

Table 11

`Action: "close"` is a request for the server to broadcast `settleAndFinalize` (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.

When `voucher` is present, it **MUST** strictly advance the on-chain watermark (`settled < voucher.cumulativeAmount`). A supplied voucher at or below the current on-chain `settled` is invalid and **MUST** cause `settleAndFinalize` to reject; clients **SHOULD** omit `voucher` instead when no additional settlement is needed. When `voucher` is omitted, the server finalizes at the current on-chain `settled` watermark.

See Section 14.5 for the full settlement procedure, including how `settleAndFinalize` 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 12

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 and authorize a cumulative amount because `channelId` is already bound to that context. 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 13

9.3. Voucher Signing

The signed voucher payload is 48 bytes in fixed Borsh layout:

Offset	Length	Field	Encoding
0	32	`channelId`	Raw Solana address bytes

Offset	Length	Field	Encoding
32	8	`cumulativeAmount`	u64 little-endian
40	8	`expiresAt`	i64 little-endian; `0` = no expiration

Table 14

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 layout is the on-chain argument for `settle` and (without the `hasVoucher` byte) for `settleAndFinalize`.

9.4. Voucher Verification

The server **MUST** verify each voucher:

1. Deserialize and canonicalize the voucher data.
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.
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 discriminator is not `ClosedChannel` (i.e., the channel has not been tombstoned by `distribute`).
7. Verify `status == Open` (i.e., `closureStartedAt == 0` and the channel has not yet been finalized). Servers **MUST** reject new voucher acceptance on channels with a pending forced close unless the voucher is being used only to drive `settleAndFinalize`.
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` / `settleAndFinalize` require the stored signed payload.

9.5. On-Chain Voucher Verification

When the channel program executes `settle` or `settleAndFinalize` (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 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.

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 `Finalized` 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.

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.
- **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
`status`	`"open"` or `"closed"`
`payer`	Payer public key
`authorizationPolicy`	Voucher signer policy
`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

Field	Description
`settledOnChain`	Highest cumulative amount already settled on-chain
`closureStartedAt`	Pending forced-close timestamp, if any

Table 15

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.

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.

The available off-chain balance is computed as:

```
available = acceptedCumulative - spentAmount
```

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 17.2); 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. Compute `cost` from the challenged `amount`, `unitType`, and any implementation-specific metering policy.
2. Compute `available = acceptedCumulative - spentAmount`.
3. If `available < cost`: return 402 requesting a new voucher or topUp.
4. Persist `spentAmount += cost` BEFORE serving.

5. Serve the resource with a receipt.

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 voucher acceptance and debit processing per `channelId`. 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.

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`, 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, and salt 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. 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.
7. Validate the complete compiled message — resolving any version-0 address-lookup-table entries — not just the channel instruction. Verify the transaction 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.
8. 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.
9. Reject any disagreement between the challenge, credential payload, decoded transaction, derived PDA, escrow ATA, or token program.
10. If fee payer mode: co-sign and broadcast. Otherwise: broadcast as-is.
11. 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; -

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 finalized; and - `closureStartedAt` is `0`.

12. Create server-side channel state.
13. 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, `mint` (still allow-listed), `payee`, `authorizedSigner`, and `distributionHash` all match the active challenge and session. A resumed channel shares one cumulative ledger across challenges, keyed by `channelId`, so a single cumulative voucher cannot be reused to buy multiple responses.

14.3. Voucher Update (No Settlement)

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 `settleAndFinalize` (within grace) or `finalize` (after grace).

14.5. Close (Cooperative)

1. If a final voucher is provided, verify the `SignedVoucher` against the active channel: `voucher.channelId` equals the payload `channelId`, `signer` equals the channel `authorizedSigner`, the Ed25519 signature verifies over the Borsh payload, freshness checks pass, and `settled < cumulativeAmount <= deposit`.

2. Build and broadcast `settleAndFinalize`. The server **SHOULD** bundle `distribute` in the same transaction so the merchant-side payout, payer refund, treasury sweep, and PDA tombstone all land atomically. A bundle whose `distribute` carries many recipients may require a version-0 transaction with an address lookup table.
3. Mark the channel as `"closed"` in server-side state.
4. Persist final `settledOnChain` and terminal accounting state after confirmation.
5. 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 `settleAndFinalize` 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. Forced Close (Client-Initiated)

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

1. Client submits `requestClose` directly to RPC.
2. Grace period begins (per-channel `gracePeriod`).
3. During the grace period, the server **MAY** still call `settleAndFinalize` with the latest voucher.
4. After the grace period, any party submits `finalize` (permissionless) to transition the channel to `Finalized`.
5. The payer **MAY** submit `withdrawPayer` to recover `deposit - settled` immediately. Independently, any party **MAY** submit `distribute` with the splits preimage; the merchant side is paid, any pending payer refund is also paid, residual is swept to treasury, and the PDA is tombstoned.

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

Field	Type	Required	Description
`acceptedCumulative`	string	**REQUIRED**	Highest voucher amount accepted
`spent`	string	**REQUIRED**	Total amount charged so far

Table 16

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 17

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.

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`.

17. Security Considerations

17.1. Transport Security

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

17.2. 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 17.11.

17.3. 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 `Finalized`. 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`.

17.4. Voucher Replay Protection

Vouchers are bound to a specific channel via `channelId` and ordered by `cumulativeAmount`. A voucher from one channel cannot be replayed in another.

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.

17.5. 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.

17.6. 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.

17.7. 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 `settleAndFinalize` (and, optionally, `distribute`). Servers **MUST NOT** resume metered service after `closureStartedAt` is set.

17.8. Delegated Signer Risks

If delegated 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.

17.9. 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.

17.10. 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.

17.11. 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 18

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

17.12. 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.

17.13. Channel Exhaustion

A malicious client could open many channels with small deposits, consuming on-chain storage. Channel programs **SHOULD** require a minimum deposit that covers the rent cost of the channel account.

Servers **SHOULD** also enforce a minimum economically useful deposit to avoid channel spam with balances too small to justify signature verification, storage, and settlement overhead.

17.14. 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.

17.15. Clock Skew

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

17.16. 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 `settleAndFinalize`, 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