Workgroup:	Network Working Group
Internet-Draft:	draft-evm-session-00
Published:	23 July 2026
Intended Status:	Informational
Expires:	24 January 2027
Authors:	X. Tian	E. Wang	M. Wong	A. Zhou
---	---	---	---	---
	OKG	*OKG*	*OKG*	*OKG*

EVM Session Intent for HTTP Payment Authentication

Abstract

This document defines the "evm" payment method implementation of the "session" intent for the Payment HTTP Authentication Scheme. It specifies unidirectional streaming payment channels for incremental, voucher-based payments on any EVM-compatible blockchain, suitable for metered services such as LLM inference.

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 24 January 2027.

Copyright Notice

Table of Contents

1. Introduction

This document is published as Informational but contains normative requirements using BCP 14 keywords [RFC2119] [RFC8174] to ensure interoperability between implementations.

This document defines the "evm" payment method implementation of the "session" intent registered by [I-D.payment-intent-session].

The `session` intent establishes a unidirectional streaming payment channel using on-chain escrow and off-chain [EIP-712] vouchers. This enables high-frequency, low-cost payments by batching many off-chain voucher signatures into periodic on-chain settlements.

Unlike the `charge` intent which requires the full payment amount upfront, the `session` intent allows clients to pay incrementally as they consume services, paying exactly for resources received.

This specification adapts the streaming payment channel mechanism defined in [I-D.tempo-session]: on-chain escrow holds deposited funds; the client signs cumulative EIP-712 vouchers authorizing increasing payment amounts off-chain; the server settles periodically or at session close. This document extends the mechanism for any EVM-compatible chain, with EVM-specific transaction formats, gas models, and domain separators.

1.1. Use Case: LLM Token Streaming

Consider an LLM inference API that charges per output token:

1. Client requests a streaming completion (SSE response)
2. Server returns 402 with a `session` challenge
3. Client opens a payment channel on-chain, depositing funds
4. Server begins streaming response
5. As response streams, or over incremental requests, client signs vouchers with increasing amounts
6. Server settles periodically or at stream completion

The client pays exactly for tokens received, with no worst-case reservation.

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

Streaming Payment Channel A unidirectional off-chain payment mechanism where the payer deposits funds into an escrow contract and signs cumulative vouchers authorizing increasing amounts.

Voucher An [EIP-712] signed message authorizing a cumulative payment amount for a specific channel. Vouchers are monotonically increasing in amount.

Channel A payment relationship between a payer and payee, identified by a unique `channelId`. The channel holds deposited funds and tracks cumulative settlements.

Settlement The on-chain [ERC-20] transfer that converts off-chain voucher authorizations into actual token movement.

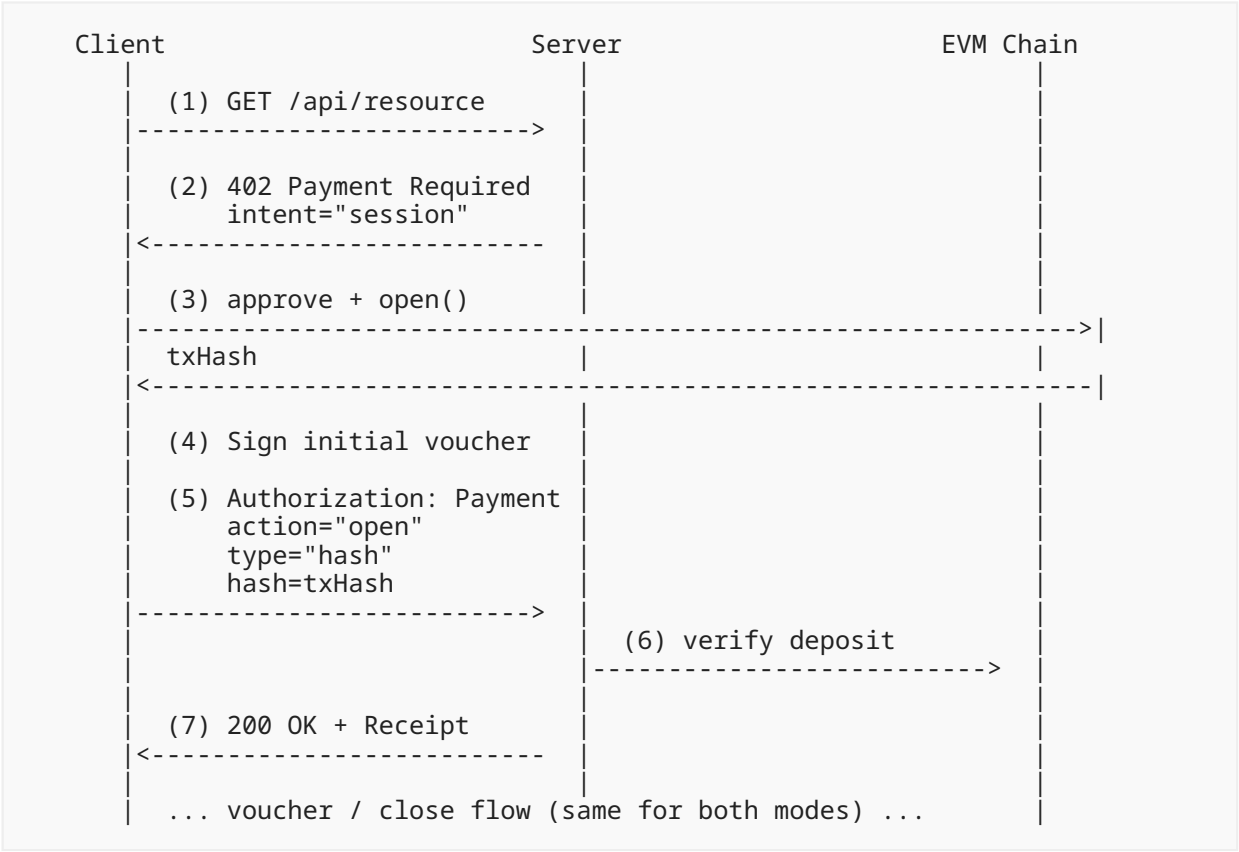
Authorized Signer An address delegated to sign vouchers on behalf of the payer. Defaults to the payer if not specified. In this specification, voucher signatures are verified either as ECDSA secp256k1 signatures (for EOA signers) or via ERC-1271 `isValidSignature` (for smart-contract wallets such as Safe or ERC-4337 accounts). The `authorizedSigner` field **MAY** be an EOA or an ERC-1271-compliant contract wallet.

Base Units The smallest indivisible unit of an ERC-20 token, determined by the token's decimal precision. For example, USDC (6 decimals) uses 1,000,000 base units per 1 USDC.

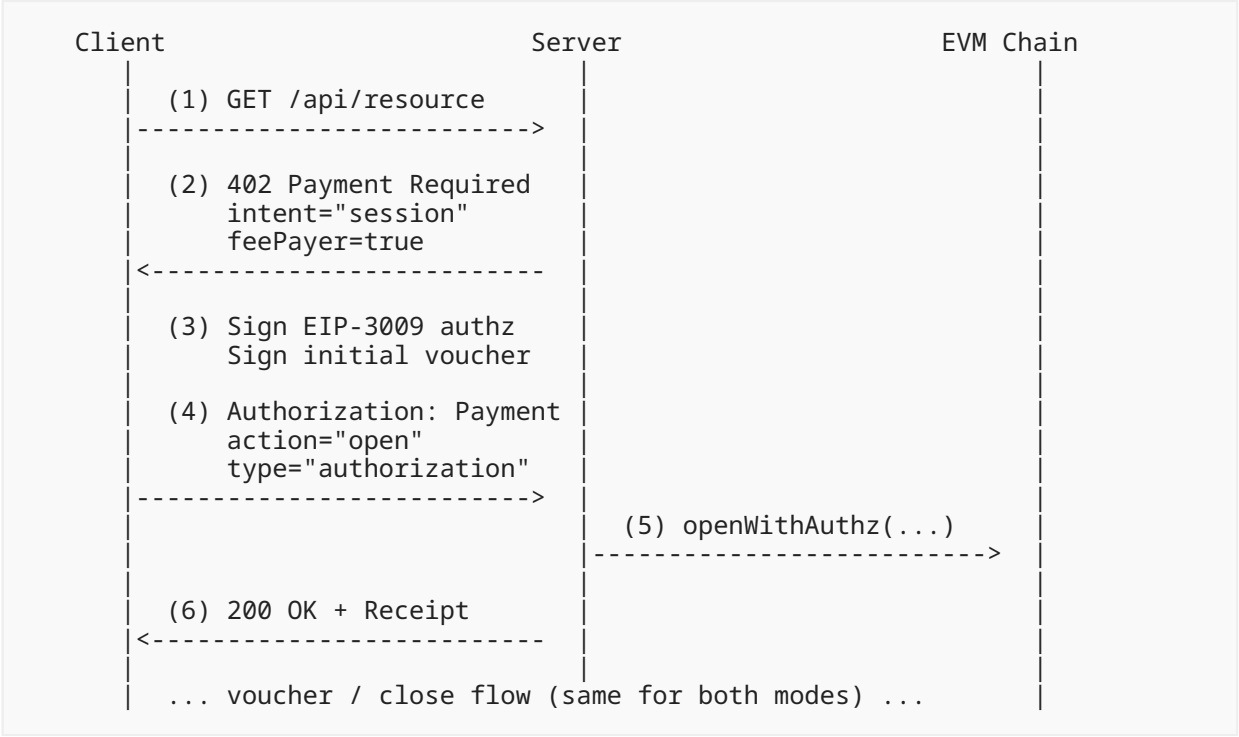
4. Session Flow

The following diagrams illustrate the two open modes.

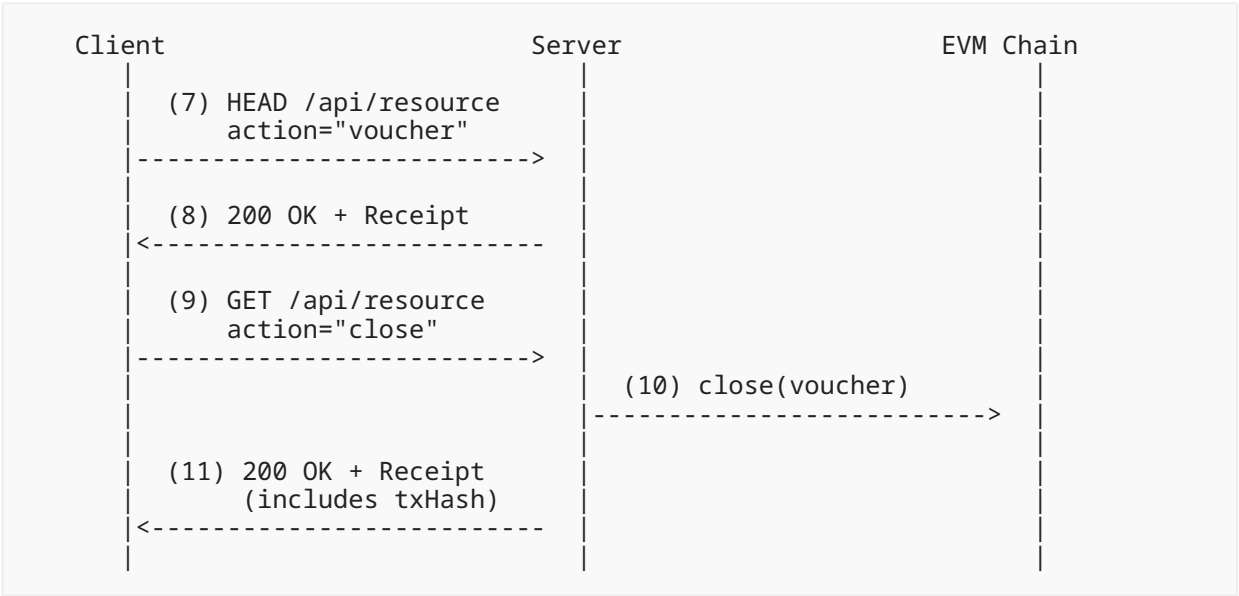
Client-broadcast open (feePayer: false):



Server-submitted open (feePayer: true):



Voucher and close flow (common to both modes):



Voucher updates and close requests are submitted to the **same resource URI** that requires payment. Servers **SHOULD** support voucher updates via any HTTP method [RFC9110]; clients **MAY** use `HEAD` for pure voucher top-ups when no response body is needed.

5. Concurrency Model

A channel supports one active session at a time. The cumulative voucher semantics ensure correctness — each voucher advances a single monotonic counter. The channel is the unit of concurrency; no additional session locking is required.

When a client sends a new streaming request on a channel that already has an active session, servers **SHOULD** terminate the previous session and start a new one. Voucher updates **MAY** arrive on separate HTTP connections (including HTTP/2 streams) and **MUST** be processed atomically with respect to balance updates.

Servers **MUST** ensure that voucher acceptance and balance deduction are serialized per channel to prevent race conditions.

6. Encoding Conventions

This section defines normative encoding rules for interoperability.

6.1. Hexadecimal Values

All byte arrays (addresses, hashes, signatures, channelId) use:

- Lowercase hexadecimal encoding
- `0x` prefix
- No padding or truncation

Type	Length	Example
address	42 chars (0x + 40 hex)	`0x742d35cc6634c0532925a3b844bc9e7595f8fe00`
bytes32	66 chars (0x + 64 hex)	`0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f`
signature (bytes)	132 chars (0x + 130 hex)	65-byte ECDSA signature

Table 1

All signatures in this specification are 65 bytes, encoded as `r (32 bytes) || s (32 bytes) || v (1 byte)` and passed as a single `bytes` parameter. Implementations **MUST NOT** produce EIP-2098 compact 64-byte signatures [EIP-2098]. Implementations **MAY** accept them (e.g., when using a standard signature-verification library such as OpenZeppelin SignatureChecker that transparently handles both formats), but **MUST NOT** require clients to produce them.

Implementations **MUST** use lowercase hex for channelId, signatures, and hashes. Address fields in the request schema (currency, recipient, escrowContract) **SHOULD** use [EIP-55] mixed-case encoding for display but **MUST** be compared by decoded 20-byte value, not string form.

6.2. Numeric Values

Integer values (amounts, timestamps) are encoded as decimal strings in JSON to avoid precision loss with large numbers:

Field	Encoding	Example	Rationale
`cumulativeAmount`	Decimal string	`"250000"`	May exceed Number.MAX_SAFE_INTEGER
`validAfter`, `validBefore`	Decimal string	`"1743523500"`	uint256 on-chain; string for consistency
`chainId`	JSON number	`196`	Small values; no precision risk

Table 2

The `chainId` uses JSON number encoding because EVM chain IDs are small enough to avoid precision issues. All other large integers use decimal strings. In EIP-712 typed data, `chainId` is a `uint256` — implementations **MUST** convert the JSON number to `uint256` when constructing the domain separator.

6.3. Timestamp Format

HTTP headers and receipt fields use [RFC3339] formatted timestamps. Timestamps in EIP-712 signed data use Unix seconds as decimal strings.

7. Channel Escrow Contract

Streaming payment channels require an on-chain escrow contract that holds user deposits and enforces voucher-based withdrawals.

The escrow's deposit accounting assumes the `currency` token transfers exactly the requested amount. Implementations **MUST** restrict the escrow to well-behaved ERC-20 tokens and **MUST NOT** use it with fee-on-transfer or rebasing tokens: those would make `channel.deposit` over-record the balance actually held, letting a payee settle more than was escrowed.

7.1. Channel State

Each channel is identified by a unique `channelId` and stores:

Field	Type	Description
`payer`	address	User who deposited funds
`payee`	address	Server authorized to withdraw
`token`	address	[ERC-20] token address
`authorizedSigner`	address	Authorized signer (0 = payer)
`deposit`	uint128	Total amount deposited
`settled`	uint128	Cumulative amount already withdrawn by payee
`closeRequestedAt`	uint64	Timestamp when close was requested (0 if not)
`finalized`	bool	Whether channel is closed. Sticky: set once and never cleared; the record is never deleted (see Section 15.2)

Table 3

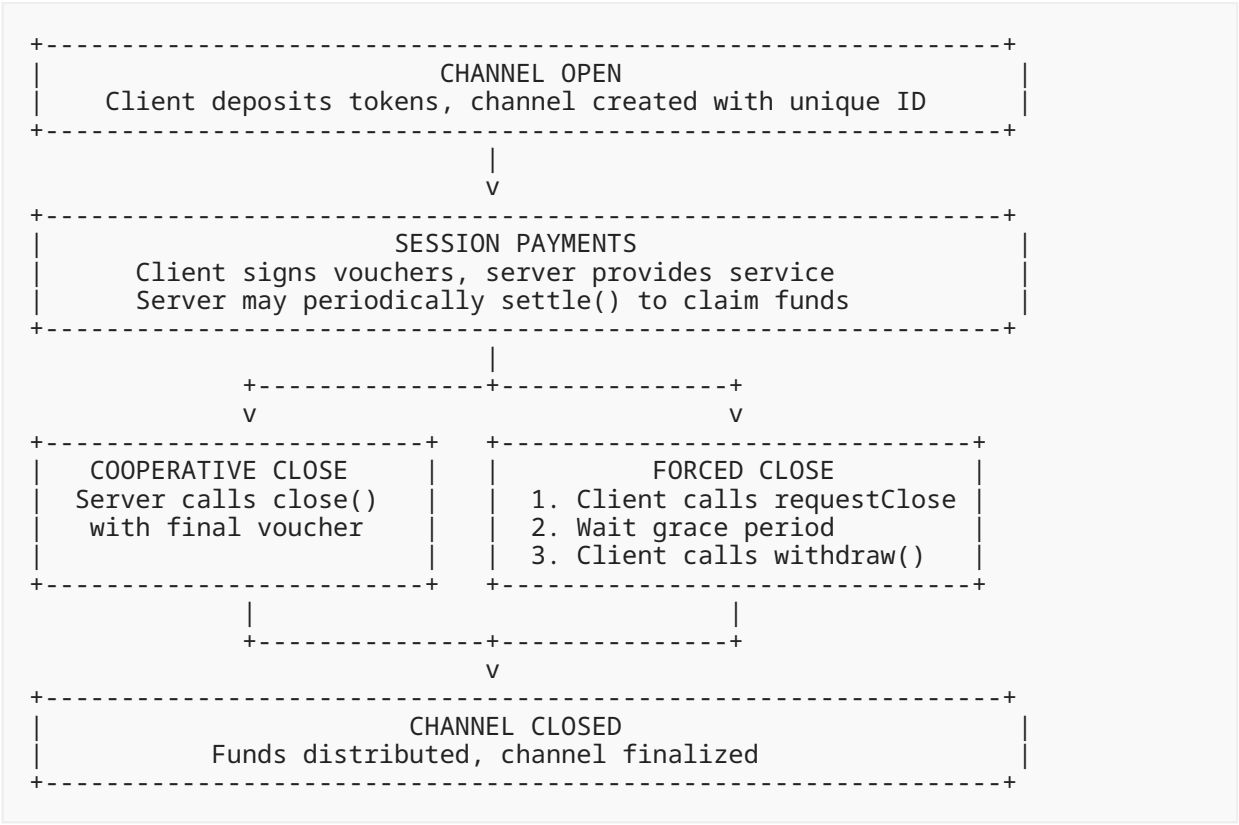
The `channelId` **MUST** be computed deterministically using the escrow contract's `computeChannelId()` function or equivalent logic:

```
channelId = keccak256(abi.encode(
    payer,
    payee,
    token,
    salt,
    authorizedSigner,
    address(this),
    block.chainid
))
```

Note: The `channelId` includes `address(this)` (the escrow contract address) and `block.chainid`, explicitly binding the channel to a specific contract deployment and chain. This computation is identical to the Tempo escrow specification. For the relayed open functions (`openWithAuthorization`, `openWithPermit2`), `payer` in this computation is the `from` argument (the depositor), not the relayer that submits the transaction.

7.2. Channel Lifecycle

Channels have no expiry — they remain open until explicitly closed.



7.3. Contract Functions

The escrow surface is split into a **mandatory core** that every compliant contract **MUST** implement, and an **optional Relayed / Gasless Operations profile** (Section 9.4) that an implementation **MAY** omit in whole.

Mandatory core:

- `open`, `settle`, `topUp`, `close`, `requestClose`, `withdraw`

Optional Relayed / Gasless Operations profile (Section 9.4):

- Payer-funded via EIP-3009, relayer-submitted: `openWithAuthorization`, `topUpWithAuthorization`
- Payer-funded via Permit2, relayer-submitted: `openWithPermit2`, `topUpWithPermit2`
- Payee-initiated via EIP-712 payee authorization, relayer-submitted: `settleWithAuthorization`, `closeWithAuthorization`

The two payer-funded paths are alternatives, not both required: EIP-3009 suits tokens that ship `receiveWithAuthorization` (e.g. USDC), while Permit2 covers any ERC-20 the payer has approved to the Permit2 contract. An implementation **MAY** offer either or both.

All six relayed functions share one shape: an off-chain authorization (EIP-3009 or Permit2 from the payer, or an EIP-712 payee authorization from the seller) plus submission by any relayer that pays gas. An implementation **MAY** support any subset, but a server **MUST NOT** advertise a capability (`feePayer: true`, or relayed settle/close) whose underlying function its escrow does not implement. Each function below that belongs to the profile is tagged accordingly; all others are mandatory.

7.3.1. open

Opens a new channel with escrowed funds. The caller becomes the payer. Requires prior `approve(escrow, deposit)` on the ERC-20 token; the contract pulls funds via `transferFrom`. The contract **MUST** revert if a channel with the computed `channelId` already exists.

Channel records **MUST** be retained permanently: the `finalized` flag is sticky (set once at close/withdraw and never cleared) and the channel record **MUST NOT** be deleted. Because `channelId` is derived only from stable inputs (`payer, payee, token, salt, authorizedSigner`, contract, chain) with no epoch component, deleting a finalized record would let the same inputs re-derive an identical `channelId` and EIP-712 voucher digest, replaying old-epoch vouchers against a freshly funded channel. Retaining the record makes the "already exists" check above reject any re-open. See Section 15.2.

Parameter	Type	Description
`payee`	address	Server's address authorized to withdraw
`token`	address	ERC-20 token contract address
`deposit`	uint128	Amount to deposit in base units
`salt`	bytes32	Random value for channelId computation
`authorizedSigner`	address	Delegated signer; `address(0)` = payer

Table 4

```
function open(
    address payee,
    address token,
    uint128 deposit,
    bytes32 salt,
    address authorizedSigner
) external returns (bytes32 channelId);
```

7.3.2. openWithAuthorization

Part of the optional Relayed / Gasless Operations profile (Section 9.4).

Opens a channel using EIP-3009 [EIP-3009] authorization. The server (or any relayer) submits the transaction, pulling funds from the payer via `receiveWithAuthorization` inside the contract.

> **Note:** The escrow contract **MUST** use `receiveWithAuthorization` (not `transferWithAuthorization`) to pull funds from the token. `receiveWithAuthorization` enforces `msg.sender == to`, preventing front-running attacks where an attacker extracts the EIP-3009 signature from the mempool and calls the token directly (see Section 15.13). This specification targets tokens that support the USDC v2.2 `bytes signature` overload of `receiveWithAuthorization` and calls it directly with the packed 65-byte signature. Tokens that only expose the canonical `(uint8 v, bytes32 r, bytes32 s)` interface are NOT supported by this escrow design.

Parameter	Type	Description
`payee`	address	Server's address
`token`	address	ERC-20 token contract
`deposit`	uint128	Amount to deposit
`salt`	bytes32	Random value
`authorizedSigner`	address	Delegated signer; `address(0)` = payer
`from`	address	Payer address (EIP-3009 `from`)
`validAfter`	uint256	EIP-3009 validity start
`validBefore`	uint256	EIP-3009 validity end
`nonce`	bytes32	EIP-3009 nonce; **MUST** equal the value derived per Section 15.13
`signature`	bytes	Packed EIP-3009 authorization signature (65 bytes)

Table 5

```
function openWithAuthorization(
    address payee,
    address token,
    uint128 deposit,
    bytes32 salt,
    address authorizedSigner,
    address from,
    uint256 validAfter,
    uint256 validBefore,
    bytes32 nonce,
    bytes calldata signature
) external returns (bytes32 channelId);
```

The `nonce` parameter is supplied by the caller for transparency, but the contract **MUST** recompute the expected nonce as `keccak256(abi.encode(from, payee, token, salt, authorizedSigner))` and revert if the supplied value does not match. Compliant implementations **SHOULD** revert with a dedicated error such as `NonceMismatch()` so callers can distinguish this failure mode. See Section 15.13 for the threat model.

7.3.3. openWithPermit2

Part of the optional Relayed / Gasless Operations profile (Section 9.4).

Opens a channel using [Permit2] `SignatureTransfer` with a witness. The server (or any relayer) submits the transaction, pulling funds from the payer via the canonical Permit2 contract. This path supports any ERC-20 token that the payer has previously approved for the Permit2 contract (typically a one-time, unlimited approval).

> **Note:** The escrow contract **MUST** use `permitWitnessTransferFrom` (not `permitTransferFrom`) to bind the channel intent (`payee`, `salt`, `authorizedSigner`) into the EIP-712 signature as a named witness struct. This serves two purposes: (1) wallets that render EIP-712 typed data display the channel parameters as labeled fields at signing time, instead of leaving them to be hashed opaquely into the nonce; (2) the contract enforces channel-parameter integrity via the Permit2 signature itself, so an attacker cannot front-run with a different `payee`. The Permit2 `spender` is fixed to `msg.sender` by the contract, so only the escrow can spend the signature.

Parameter	Type	Description
`payee`	address	Server's address
`token`	address	ERC-20 token contract
`deposit`	uint128	Amount to deposit
`salt`	bytes32	Random value
`authorizedSigner`	address	Delegated signer; `address(0)` = payer
`from`	address	Payer address (Permit2 `owner`)
`nonce`	uint256	Permit2 nonce (any unused value; bitmap-based replay protection)
`deadline`	uint256	Permit2 signature deadline (Unix seconds)
`signature`	bytes	Permit2 EIP-712 signature (65 bytes)

Table 6

```
function openWithPermit2(
    address payee,
    address token,
    uint128 deposit,
    bytes32 salt,
    address authorizedSigner,
    address from,
    uint256 nonce,
    uint256 deadline,
    bytes calldata signature
) external returns (bytes32 channelId);
```

The escrow contract **MUST** construct the Permit2 `PermitTransferFrom` struct, `SignatureTransferDetails`, and the `ChannelOpenWitness` witness struct from these parameters with `permitted.token = token`, `permitted.amount = deposit`, `transferDetails.to = address(this)`, `transferDetails.requestedAmount = deposit`, and witness fields `(payee, salt, authorizedSigner)`. It then computes `witnessHash = keccak256(abi.encode(CHANNEL_OPEN_WITNESS_TYPEHASH, payee, salt, authorizedSigner))` and calls `IPermit2(PERMIT2).permitWitnessTransferFrom(permit, transferDetails, from, witnessHash, WITNESS_TYPE_STRING, signature)` on the canonical Permit2 deployment. If the payer signed a different `payee`, `salt`, or `authorizedSigner` than the function arguments, the Permit2 signature verification reverts.

7.3.4. settle

Server withdraws funds using a signed voucher without closing the channel. The contract **MUST** revert if `msg.sender != channel.payee`.

Parameter	Type	Description
`channelId`	bytes32	Unique channel identifier
`cumulativeAmount`	uint128	Cumulative total authorized
`signature`	bytes	EIP-712 signature from authorized signer

Table 7

The contract **MUST** revert (e.g., with `AmountNotIncreasing()`) if `cumulativeAmount <= channel.settled`, and (e.g., with `AmountExceedsDeposit()`) if `cumulativeAmount > channel.deposit`; only a strictly increasing cumulative amount within the deposited balance advances settlement. Otherwise the contract computes `delta = cumulativeAmount - channel.settled`, sets `channel.settled = cumulativeAmount`, and transfers `delta` to the payee. This on-chain check is the last line of defense for the rollback prevention described in Section 15.5; a server-side check alone does not constrain a payee calling the contract directly.

```
function settle(
    bytes32 channelId,
    uint128 cumulativeAmount,
    bytes calldata signature
) external;
```

7.3.5. topUp

User adds more funds to an existing channel. Requires prior `approve(escrow, additionalDeposit)`. The contract **MUST** revert if `msg.sender != channel.payer`. If a close request is pending (`channel.closeRequestedAt != 0`), calling `topUp()` **MUST** reset `closeRequestedAt` to `0`, cancelling the pending close.

Parameter	Type	Description
`channelId`	bytes32	Existing channel identifier
`additionalDeposit`	uint128	Additional amount in base units

Table 8

```
function topUp(
    bytes32 channelId,
    uint128 additionalDeposit
) external;
```

7.3.6. topUpWithAuthorization

Part of the optional Relayed / Gasless Operations profile (Section 9.4).

Adds funds using EIP-3009 authorization. The server calls this on behalf of the payer.

Parameter	Type	Description
`channelId`	bytes32	Existing channel identifier
`additionalDeposit`	uint128	Additional amount
`from`	address	Payer address
`topUpSalt`	bytes32	Random value for nonce derivation
`validAfter`	uint256	EIP-3009 validity start
`validBefore`	uint256	EIP-3009 validity end
`nonce`	bytes32	EIP-3009 nonce; **MUST** equal the value derived per Section 15.13

Parameter	Type	Description
`signature`	bytes	Packed EIP-3009 authorization signature (65 bytes)

Table 9

```
function topUpWithAuthorization(
    bytes32 channelId,
    uint128 additionalDeposit,
    address from,
    bytes32 topUpSalt,
    uint256 validAfter,
    uint256 validBefore,
    bytes32 nonce,
    bytes calldata signature
) external;
```

As with `openWithAuthorization`, the contract **MUST** recompute the expected nonce as `keccak256(abi.encode(channelId, additionalDeposit, from, topUpSalt))` and revert (e.g., with `NonceMismatch()`) if the supplied `nonce` does not match. Clients **MUST** use the same derivation when signing.

7.3.7. topUpWithPermit2

Part of the optional Relayed / Gasless Operations profile (Section 9.4).

Adds funds using [Permit2] `SignatureTransfer` with a witness. Mirrors `openWithPermit2` for an existing channel. The escrow **MUST** call `permitWitnessTransferFrom` with a `ChannelTopUpWitness` binding `channelId`, so the payer's wallet shows the target channel and the contract enforces that the signature cannot be redirected to a different channel. Unlike the EIP-3009 path, no `topUpSalt` is required: Permit2's unordered nonce already provides replay protection for repeated top-ups, and `channelId` alone binds the deposit to the channel. Because this function takes no `token` argument, the escrow sets `permitted.token = channel.token` from the existing channel record when reconstructing the Permit2 permit.

Parameter	Type	Description
`channelId`	bytes32	Existing channel identifier
`additionalDeposit`	uint128	Additional amount
`from`	address	Payer address
`nonce`	uint256	Permit2 nonce (any unused value; bitmap-based replay protection)
`deadline`	uint256	Permit2 signature deadline (Unix seconds)

Parameter	Type	Description
`signature`	bytes	Permit2 EIP-712 signature (65 bytes)

Table 10

```
function topUpWithPermit2(
    bytes32 channelId,
    uint128 additionalDeposit,
    address from,
    uint256 nonce,
    uint256 deadline,
    bytes calldata signature
) external;
```

7.3.8. close

Server closes the channel, settling outstanding voucher and refunding remainder to payer. The contract **MUST** revert if `msg.sender != channel.payee`.

If `cumulativeAmount <= channel.settled`, the payee is forfeiting any uncollected amount (e.g., to cleanly close an exhausted or abandoned channel). In this case the contract **MAY** skip voucher signature verification and `signature` **MAY** be empty.

Parameter	Type	Description
`channelId`	bytes32	Channel to close
`cumulativeAmount`	uint128	Final cumulative amount
`signature`	bytes	EIP-712 voucher signature; **MAY** be empty when `cumulativeAmount <= channel.settled` (forfeit path)

Table 11

```
function close(
    bytes32 channelId,
    uint128 cumulativeAmount,
    bytes calldata signature
) external;
```

7.3.9. requestClose

User requests channel closure, starting a grace period. The contract **MUST** revert if `msg.sender != channel.payer`, if no channel exists for `channelId`, or if the channel is already finalized.

Parameter	Type	Description
channelId	bytes32	Channel to request closure

Table 12

```
function requestClose(bytes32 channelId) external;
```

7.3.10. withdraw

User withdraws the unsettled remainder after the forced-close grace period expires. The contract **MUST** revert if `msg.sender != channel.payer`, if the channel is already finalized, if no close has been requested (`channel.closeRequestedAt == 0`), or if the grace period has not elapsed (`block.timestamp < channel.closeRequestedAt + CLOSE_GRACE_PERIOD`). On success it transfers `channel.deposit - channel.settled` to the payer and sets `finalized` atomically with the payout.

The `closeRequestedAt == 0` check is essential: without it, `block.timestamp >= 0 + CLOSE_GRACE_PERIOD` is trivially true, so a payer could call `withdraw` without ever calling `requestClose`, draining the channel before the payee settles outstanding vouchers and bypassing the grace period entirely. The `finalized` check prevents a second payout (a cooperative `close` followed by `requestClose` + `withdraw`) from drawing on the contract's pooled balance.

Parameter	Type	Description
channelId	bytes32	Channel to withdraw from

Table 13

```
function withdraw(bytes32 channelId) external;
```

7.4. Access Control

Function	Caller	Description
open	Anyone	Creates channel; caller becomes payer
openWithAuthorization	Anyone (typically server)	Creates channel via EIP-3009; `from` becomes payer
openWithPermit2	Anyone (typically server)	Creates channel via Permit2 SignatureTransfer; `from` becomes payer

Function	Caller	Description
`settle`	Payee only	Withdraws funds using voucher
`topUp`	Payer only	Adds funds (approve + pull)
`topUpWithAuthorization`	Anyone (typically server)	Adds funds via EIP-3009; no caller restriction because the EIP-3009 signature provides authorization
`topUpWithPermit2`	Anyone (typically server)	Adds funds via Permit2; no caller restriction because the Permit2 signature provides authorization
`close`	Payee only	Closes with final voucher
`settleWithAuthorization`	Anyone (typically relayer)	Settles via payee EIP-712 authorization; payee signature, not `msg.sender`, authorizes
`closeWithAuthorization`	Anyone (typically relayer)	Closes via payee EIP-712 authorization; payee signature, not `msg.sender`, authorizes
`requestClose`	Payer only	Initiates forced close
`withdraw`	Payer only	Withdraws after grace period

Table 14

7.5. Signature Verification

The escrow contract **MUST** perform the following verification for all functions that accept voucher signatures (`settle`, `close`):

1. **Canonical signatures**: The contract **MUST** reject ECDSA signatures with non-canonical (high-s) values. Signatures **MUST** have `s <= secp256k1_order / 2` where the half-order is `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0`. See Section 15.9 for rationale.
2. **Authorized signer verification**: The contract **MUST** recover the signer address from the EIP-712 signature and verify it matches:
   - `channel.authorizedSigner` if non-zero
   - Otherwise `channel.payer` The signer may be an EOA (verified via ECDSA) or an ERC-1271 contract wallet (verified via `isValidSignature`).
3. **Domain binding**: The contract **MUST** use its own address as `verifyingContract` in the EIP-712 domain separator, ensuring vouchers cannot be replayed across different escrow deployments.

Failure to enforce these requirements on-chain would allow attackers to bypass server-side validation by submitting transactions directly to the contract.

7.6. Contract Errors

This subsection is informative. The normative requirement is that the escrow reverts under each condition below, as stated in the relevant function and security sections; the error names are a **RECOMMENDED** common vocabulary for implementers, tooling, and diagnostics. Implementations **MAY** use different names or revert representations. On-the-wire, a reverted transaction is reported to clients as the `transaction-reverted` problem type (Section 12.7), not as a raw Solidity selector.

Suggested error	Revert condition	Functions
`ChannelAlreadyExists`	A channel with the computed `channelId` already exists, including a finalized one (Section 15.2)	`open`, `openWithAuthorization`, `openWithPermit2`
`ChannelNotFound`	No channel for the given `channelId`	`settle`, `topUp*`, `close*`, `requestClose`, `withdraw`
`ChannelFinalized`	Channel is already finalized	`settle`, `topUp*`, `close*`, `requestClose`, `withdraw`
`NotPayee`	`msg.sender != channel.payee`	`settle`, `close`
`NotPayer`	`msg.sender != channel.payer`	`topUp`, `requestClose`, `withdraw`
`AmountNotIncreasing`	`cumulativeAmount <= channel.settled` on a non-forfeit path (Section 15.5)	`settle*`, `close*`
`AmountExceedsDeposit`	`cumulativeAmount > channel.deposit`	`settle*`, `close*`
`InvalidSignature`	Voucher or payee signature fails recovery, signer mismatch, or non-canonical high-s (Section 7.5)	`settle*`, `close*`
`NonceMismatch`	Supplied EIP-3009 nonce ≠ value derived from channel parameters (Section 15.13)	`openWithAuthorization`, `topUpWithAuthorization`

Suggested error	Revert condition	Functions
`NonceAlreadyUsed`	`(channel.payee, channelId, nonce)` already consumed (Section 9.4.2)	`settleWithAuthorization`, `closeWithAuthorization`
`AuthorizationExpired`	`block.timestamp > deadline` on a payee authorization (Section 9.4.2)	`settleWithAuthorization`, `closeWithAuthorization`
`CloseNotReady`	`withdraw` called with no pending close (`closeRequestedAt == 0`) or before the close grace period elapsed	`withdraw`
`ZeroDeposit`	Deposit amount is `0`	`open*`, `topUp*`
`DepositOverflow`	Deposit would exceed the `uint128` bound	`open*`, `topUp*`

Table 15

In the table, a trailing `*` denotes the base function plus its `WithAuthorization` and `WithPermit2` variants where they exist.

8. Request Schema

The `request` parameter in the `WWW-Authenticate` challenge contains a JSON [RFC8259] object, serialized using JCS [RFC8785] and then base64url-encoded [RFC4648].

8.1. Fields

Field	Type	Required	Description
`amount`	string	**REQUIRED**	Price per unit of service in base units (not total charge)
`unitType`	string	**OPTIONAL**	Unit being priced (e.g., `"llm_token"`, `"byte"`, `"request"`)
`suggestedDeposit`	string	**OPTIONAL**	Suggested channel deposit amount in base units
`currency`	string	**REQUIRED**	ERC-20 token contract address (EIP-55 checksummed)
`recipient`	string	**REQUIRED**	Payee address (server's withdrawal address)

Field	Type	Required	Description
`description`	string	**OPTIONAL**	Human-readable payment description
`externalId`	string	**OPTIONAL**	Merchant's reference (order ID, invoice number, etc.)

Table 16

For the `session` intent, `amount` specifies the price per unit of service in base units, not a total charge. The total cost depends on consumption: `total = amount * units_consumed`.

8.2. Method Details

Field	Type	Required	Description
`methodDetails.chainId`	number	**REQUIRED**	EVM chain ID
`methodDetails.escrowContract`	string	**REQUIRED**	Address of the channel escrow contract
`methodDetails.channelId`	string	**OPTIONAL**	Channel ID if resuming an existing channel
`methodDetails.minVoucherDelta`	string	**OPTIONAL**	Minimum amount increase between vouchers (base units). Default: `"0"` (any positive increment accepted). See Section 15.8
`methodDetails.feePayer`	boolean	**OPTIONAL**	If `true`, server pays gas for open/topUp (default: `false`)
`methodDetails.credentialTypes`	array	**OPTIONAL**	Credential formats the server accepts, as an ordered list of top-level `payload.type` values. EVM session uses the shared EVM values `"permit2"`, `"authorization"`, and `"hash"`; it omits the charge-only full signed transaction path. Order expresses server preference

Field	Type	Required	Description
`methodDetails.permit2Contract`	string	**OPTIONAL**	Permit2 contract address used as the EIP-712 `verifyingContract` on the `permit2` authorization path. Defaults to the canonical deterministic Permit2 deployment; **REQUIRED** when the target chain's Permit2 is not at the canonical address. The client **MUST** use this value (or the canonical default when omitted) as `verifyingContract` when signing, and it **MUST** match the escrow's configured Permit2 address

Table 17

Servers **MAY** advertise `credentialTypes` listing every credential format they accept for this challenge. The list uses the same ordered preference semantics as the EVM charge intent: clients select the first listed type they can produce unless local policy chooses otherwise, and **MUST NOT** submit a format absent from the list. If `credentialTypes` is omitted, it defaults to `["hash"]`.

When `feePayer` is `true`, servers that want clients to use a server-submitted open/topUp format **MUST** include at least one such type backed by the escrow (`openWithAuthorization` deployed ⇒ include `"authorization"`; `openWithPermit2` deployed ⇒ include `"permit2"`). They **MAY** also include `"hash"` as a client-broadcast fallback. This makes the supported paths discoverable in-band rather than relying on out-of-band documentation. A contract **MAY** additionally expose its relayed-path support on-chain (e.g. via an introspection view) for clients that verify the escrow directly, but the challenge field is the authoritative signal for the session flow.

Channel reuse is **OPTIONAL**. Servers **MAY** include `channelId` to suggest resuming an existing channel:

- **New channel** (no `channelId`): Client generates a random salt, computes `channelId` using the formula in Section 7.1, opens the channel on-chain, and returns the `channelId` in the credential.
- **Existing channel** (`channelId` provided): Client **MUST** verify `channel.deposit - channel.settled >= amount` before resuming.

Example (new channel):

```
{
  "amount": "100",
  "unitType": "llm_token",
  "suggestedDeposit": "5000000",
  "currency": "0x74b7F16337b8972027F6196A17a631ac6dE26d22",
  "recipient": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
  "methodDetails": {
    "escrowContract": "0x1234567890abcdef1234567890abcdef12345678",
    "chainId": 196,
    "minVoucherDelta": "10000"
  }
}
```

This requests a price of 0.0001 USDC per LLM token on X Layer, with a suggested deposit of 5.00 USDC (approximately 50,000 tokens). The minVoucherDelta of 10,000 base units (0.01 USDC) means vouchers cover at least 100 tokens each.

Example (existing channel):

```
{
  "amount": "25",
  "unitType": "llm_token",
  "currency": "0xA8CE8aee21bC2A48a5EF670afCc9274C7bbbC035",
  "recipient": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
  "methodDetails": {
    "escrowContract": "0x1234567890abcdef1234567890abcdef12345678",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "chainId": 196
  }
}
```

9. Fee Payment

The `feePayer` field affects only the client-originated channel funding transactions (`open` and `topUp`). Settlement and close are always server-initiated and server-funded.

9.1. Server-Paid Fees (feePayer: true)

When `feePayer: true`, the client submits a token-pull authorization signature instead of broadcasting an on-chain transaction. The server submits the on-chain transaction and pays gas from its own balance.

This specification supports two authorization formats, distinguished by the credential's top-level `payload.type`:

- **EIP-3009** ([EIP-3009], `type="authorization"`): the token itself implements `receiveWithAuthorization`. Suitable for stablecoins such as USDC and EURC that ship EIP-3009. No prior approval is required from the payer.

- **Permit2** ([Permit2], `type="permit2"`): the canonical Permit2 contract (deployed at the same deterministic address on most major EVM chains) brokers the transfer via `permitWitnessTransferFrom`, with the channel parameters carried as a named EIP-712 witness so they appear as labeled fields in the payer's wallet at signing time. Suitable for any ERC-20 token, including tokens without native EIP-3009 support. The payer **MUST** have previously approved the Permit2 contract for the token (typically a one-time, unlimited approval).

Selection rules:

1. The server advertises accepted credential formats in the challenge's `methodDetails.credentialTypes`, or defaults to `"hash"` when that field is omitted. For `feePayer: true`, the client selects one advertised server-submitted format (`"authorization"` or `"permit2"`) that it can produce; the credential's top-level `payload.type` is the on-the-wire discriminator for the choice.
2. **EIP-3009 path**: The client signs the EIP-712 typed data for `receiveWithAuthorization`. The server calls `openWithAuthorization()` or `topUpWithAuthorization()`. The escrow contract internally calls `receiveWithAuthorization` on the token.
3. **Permit2 path**: The client signs the EIP-712 typed data for Permit2 `PermitWitnessTransferFrom` with a channel-parameter witness (`ChannelOpenWitness` for `open`, `ChannelTopUpWitness` for `topUp`). The server calls `openWithPermit2()` or `topUpWithPermit2()`. The escrow contract internally calls `permitWitnessTransferFrom` on the canonical Permit2 contract, which verifies the witness against the function arguments and pulls tokens via the prior Permit2 approval.

When `feePayer` is `true`, the `currency` token **MUST** support at least one of the two server-submitted paths advertised by the server. Servers **MUST NOT** advertise `feePayer: true` for tokens whose authorization paths they cannot service.

9.2. Client-Paid Fees (feePayer: false)

When `feePayer: false` or omitted:

- **EOA clients**: Client calls `approve(escrow, deposit)` and then `open()` on the escrow contract, paying gas from their own balance.
- **Smart Wallet clients**: Client batches `approve + open` in a UserOperation (ERC-4337 [ERC-4337]). A Paymaster **MAY** sponsor gas for this client-submitted transaction path.

Servers that accept this client-broadcast path either omit `methodDetails.credentialTypes` (defaulting to `"hash"`) or include `"hash"` in the list; clients **MUST NOT** submit `type="hash"` when `credentialTypes` is present and omits `"hash"`.

9.3. Server-Initiated Operations

`settle` and `close` are server-originated on-chain transactions. The server pays gas for these regardless of the `feePayer` setting.

By default the payee (merchant) holds the native token of the target chain to pay gas for `settle()` and `close()`. Merchants that prefer not to maintain a native-token gas balance **MAY** instead use the relayed payee-side functions defined in the Relayed / Gasless Operations profile (Section 9.4): `settleWithAuthorization` and `closeWithAuthorization`. These let the payee sign an EIP-712 authorization off-chain and have any relayer submit and pay gas.

9.4. Relayed / Gasless Operations Profile

This profile is **OPTIONAL**. An implementation **MAY** omit it entirely, or implement any subset of its functions. The mandatory core (Section 7.3) is sufficient to operate a channel when the party initiating each transaction pays its own gas. The profile exists only to let a relayer submit and fund a transaction on behalf of a party that authorized it off-chain.

A server **MUST NOT** advertise a capability whose backing function its escrow does not implement: `feePayer: true` requires at least one of the payer-funded functions for the offered top-level `type`, and relayed settlement requires the corresponding payee-side function.

9.4.1. Payer-funded functions

`openWithAuthorization`, `openWithPermit2`, `topUpWithAuthorization`, and `topUpWithPermit2` are specified in Section 7.3. They are callable by anyone; the payer's funds move only under the payer's own EIP-3009 or Permit2 signature, and the channel parameters the contract trusts are bound into that signature as required by Section 15.13 (deterministic nonce for EIP-3009, named witness for Permit2). On the top-up paths the funding `from` need not equal `channel.payer`: the contract credits the existing channel, and any refund on close or withdraw is still paid to `channel.payer`, so a third-party top-up can only add funds, never redirect them.

9.4.2. Payee-initiated functions

`settleWithAuthorization` and `closeWithAuthorization` let the payee authorize a settlement or close off-chain and have any relayer submit it. Each requires **two** signatures: the payer/`authorizedSigner` `Voucher` (Section 11) that authorizes the amount, and an EIP-712 authorization from `channel.payee` that authorizes this specific relayed call.

Parameter	Type	Description
`channelId`	bytes32	Channel identifier
`cumulativeAmount`	uint128	Cumulative amount to settle/finalize
`nonce`	uint256	Payee-chosen, unused value in the `(payee, channelId)` scope
`deadline`	uint256	Unix seconds; contract **MUST** revert after this
`payeeSignature`	bytes	EIP-712 payee authorization (65 bytes)

Parameter	Type	Description
`voucherSignature`	bytes	EIP-712 `Voucher` signature; **MAY** be empty on the `close` forfeit path (`cumulativeAmount <= channel.settled`)

Table 18

```
function settleWithAuthorization(
    bytes32 channelId,
    uint128 cumulativeAmount,
    uint256 nonce,
    uint256 deadline,
    bytes calldata payeeSignature,
    bytes calldata voucherSignature
) external;

function closeWithAuthorization(
    bytes32 channelId,
    uint128 cumulativeAmount,
    uint256 nonce,
    uint256 deadline,
    bytes calldata payeeSignature,
    bytes calldata voucherSignature
) external;
```

The payee authorization uses these EIP-712 types under the same domain separator as the `Voucher` (Section 11):

```
SettleAuthorization(bytes32 channelId,uint128 cumulativeAmount,uint256
nonce,uint256 deadline)
CloseAuthorization(bytes32 channelId,uint128 cumulativeAmount,uint256
nonce,uint256 deadline)
```

Requirements for these functions:

1. The contract **MUST** recover the `payeeSignature` signer and verify it equals `channel.payee` (EOA via ECDSA, contract wallet via ERC-1271).
2. The contract **MUST** verify the `Voucher` signature exactly as `settle` / `close` do (Section 7.5), including the strictly increasing rule (Section 15.5); the `close` forfeit path (`cumulativeAmount <= channel.settled`) **MAY** omit `voucherSignature`.
3. Replay protection: the contract **MUST** maintain a used-set keyed by `(channel.payee, channelId, nonce)` and revert (e.g., `NonceAlreadyUsed()`) if the nonce was already consumed; `settle` and `close` authorizations **MUST** share the same used-set so a nonce cannot be reused across the two. Any caller **MAY** submit; the payee signature — not `msg.sender` — provides authorization.
4. The contract **MUST** revert (e.g., `AuthorizationExpired()`) when `block.timestamp > deadline`.

The nonce is an arbitrary unused `uint256` (no ordering requirement), mirroring the Permit2 unordered-nonce model rather than a sequential counter. Payees **SHOULD** set a short `deadline` to bound the lifetime of an unsubmitted authorization.

10. Credential Schema

The credential in the `Authorization` header contains a base64url-encoded JSON object per [I-D.httpauth-payment].

10.1. Credential Structure

Field	Type	Required	Description
`challenge`	object	**REQUIRED**	Echo of the challenge parameters
`payload`	object	**REQUIRED**	Session-specific payload object
`source`	string	CONDITIONAL	Payer identifier as a DID. **REQUIRED** when payload `type="hash"`; NOT **REQUIRED** when `type="authorization"` or `type="permit2"`

Table 19

The `source` field **SHOULD** use the `did:pkh` method [DID-PKH] with the chain ID from the challenge and the payer's Ethereum address (e.g., `did:pkh:eip155:196:0xConsumer...`). When `type="authorization"` or `type="permit2"`, the payer is identified via `authorization.from`.

10.2. Payload Actions

The `payload` object uses an `action` discriminator:

Field	Type	Required	Description
`action`	string	**REQUIRED**	One of `"open"`, `"topUp"`, `"voucher"`, `"close"`

Table 20

Action	Description
`open`	Confirms channel open; begins streaming
`topUp`	Adds funds to an existing channel
`voucher`	Submits an updated cumulative voucher
`close`	Requests server to close the channel

Table 21

10.2.1. Open Payload (feePayer: false)

When `feePayer` is `false`, the client broadcasts the `open()` or `approve + open()` transaction themselves and submits the txHash. For smart wallets, this **MAY** be an ERC-4337 UserOperation whose outer transaction targets an EntryPoint while the inner execution opens the escrow channel.

Field	Type	Required	Description
`action`	string	**REQUIRED**	`"open"`
`type`	string	**REQUIRED**	`"hash"`
`channelId`	string	**REQUIRED**	Channel identifier (hex bytes32)
`hash`	string	**REQUIRED**	Tx hash of the on-chain open, direct or via ERC-4337 EntryPoint
`cumulativeAmount`	string	**REQUIRED**	Initial cumulative amount (typically `"0"`)
`signature`	string	**REQUIRED**	EIP-712 voucher signature for the initial amount
`authorizedSigner`	string	**OPTIONAL**	Address delegated to sign vouchers (defaults to payer if omitted)
`salt`	string	**REQUIRED**	Random bytes32 hex for channelId computation

Table 22

The initial voucher (with `cumulativeAmount` typically `"0"`) is **REQUIRED** so that the server holds a signed voucher from the start of the session. This ensures the server can call `settle()` or `close()` at any time, even if the client disconnects immediately after opening.

When `type="hash"`, the `source` field in the credential structure is **REQUIRED**. The server needs the payer identity to verify the on-chain deposit. This is consistent with the `charge` intent's requirement for hash credentials.

Example:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "evm",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "source": "did:pkh:eip155:196:0xaabbccddee1122334455667788990aabbccddee",
  "payload": {
    "action": "open",
    "type": "hash",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "hash":
"0x9f8e7d6c5b4a39281700abcdef1234567890abcdef1234567890abcdef123456",
    "cumulativeAmount": "0",
    "signature": "0xabcdef1234567890...",
    "authorizedSigner": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
    "salt":
"0xaaaa1234bbbb5678cccc9012dddd3456eeee7890ffff1234aaaa5678bbbb9012"
  }
}
```

10.2.2. Open Payload (feePayer: true)

When `feePayer` is `true`, the client submits a token-pull authorization for the server to call the corresponding `openWith...()` function on the escrow contract. The top-level `type` field selects the format.

Field	Type	Required	Description
`action`	string	**REQUIRED**	`"open"`
`type`	string	**REQUIRED**	`"authorization"` for EIP-3009, or `"permit2"` for Permit2
`channelId`	string	**REQUIRED**	Channel identifier (hex bytes32)
`authorization`	object	**REQUIRED**	Token-pull authorization parameters; shape determined by the top-level `type`
`signature`	string	**REQUIRED**	Authorization signature (65 bytes hex). EIP-3009 signature if `type="authorization"`; Permit2 EIP-712 signature if `type="permit2"`
`cumulativeAmount`	string	**REQUIRED**	Initial cumulative amount (typically `"0"`)
`voucherSignature`	string	**REQUIRED**	EIP-712 voucher signature for the initial amount

Field	Type	Required	Description
`authorizedSigner`	string	**OPTIONAL**	Address delegated to sign vouchers (defaults to payer if omitted)
`salt`	string	**REQUIRED**	Random bytes32 hex for channelId computation

Table 23

The `authorization` object takes one of two shapes.

EIP-3009 shape (`type="authorization"`):

Field	Type	Required	Description
`from`	string	**REQUIRED**	Payer address
`to`	string	**REQUIRED**	Escrow contract address (= `methodDetails.escrowContract`)
`value`	string	**REQUIRED**	Deposit amount in base units
`validAfter`	string	**REQUIRED**	Unix timestamp, valid from. `"0"` = immediately
`validBefore`	string	**REQUIRED**	Unix timestamp, expires
`nonce`	string	**REQUIRED**	`bytes32` hex. EIP-3009 nonce; **MUST** be derived from channel parameters per Section 15.13

Table 24

The `nonce` **MUST** be derived from the surrounding payload's channel parameters: for `action="open"`, `nonce = keccak256(abi.encode(from, payee, token, salt, authorizedSigner))`; for `action="topUp"`, `nonce = keccak256(abi.encode(channelId, additionalDeposit, from, topUpSalt))`. The client **MUST** sign the EIP-3009 typed data with this derived nonce and **MUST** transmit it in the credential. The escrow contract recomputes the expected nonce from its own function arguments and reverts if the caller-supplied value does not match.

Permit2 shape (`type="permit2"`):

Field	Type	Required	Description
`from`	string	**REQUIRED**	Payer address (Permit2 `owner`); the Permit2 signature is verified against this address

Field	Type	Required	Description
`permitted`	object	**REQUIRED**	Permit2 `TokenPermissions`: `{ "token": <ERC-20 address = request.currency>, "amount": <deposit in base units> }`
`nonce`	string	**REQUIRED**	Decimal string. uint256 Permit2 unordered nonce (any unused value)
`deadline`	string	**REQUIRED**	Decimal string. Unix timestamp after which the signature is invalid
`witness`	object	**REQUIRED**	Channel-parameter witness. For `open`: `{ "payee", "salt", "authorizedSigner" }`. For `topUp`: `{ "channelId" }`

Table 25

The `authorization` object carries every field the Permit2 signature covers, so the signed digest is reconstructable from the object alone (plus the domain below). This mirrors the EIP-3009 shape, whose fields are likewise exactly what that scheme signs.

The `spender` is the one signed field deliberately omitted: Permit2 fixes `spender = msg.sender` inside `permitWitnessTransferFrom`, so it is always the escrow contract. Clients **MUST** set `spender = methodDetails.escrowContract` when constructing the EIP-712 hash; the escrow supplies it implicitly on-chain.

The `witness` object **MUST** carry the channel parameters:

- For `open`: `payee`, `salt`, `authorizedSigner`. `payee` **MUST** equal the challenge `request.recipient`; `salt` and `authorizedSigner` **MUST** equal the corresponding open-payload fields (an omitted `authorizedSigner` is the zero address).
- For `topUp`: `channelId`, which **MUST** equal the payload `channelId`.

The channel parameters the server passes to the escrow are authoritative: the escrow reconstructs the witness from them and the Permit2 verification reverts on any mismatch. A server **MUST** reject the credential if `authorization.witness` disagrees with those authoritative values rather than forwarding a signature that will revert on-chain.

The Permit2 EIP-712 domain and struct types are:

```
EIP712Domain(string name,uint256 chainId,address verifyingContract)
  name              = "Permit2"
  chainId           = methodDetails.chainId
  verifyingContract = canonical Permit2 contract address

// For open (action="open"):
PermitWitnessTransferFrom(TokenPermissions permitted,address spender,uint256
nonce,uint256 deadline,ChannelOpenWitness witness)
ChannelOpenWitness(address payee,bytes32 salt,address authorizedSigner)
TokenPermissions(address token,uint256 amount)

// For topUp (action="topUp"):
PermitWitnessTransferFrom(TokenPermissions permitted,address spender,uint256
nonce,uint256 deadline,ChannelTopUpWitness witness)
ChannelTopUpWitness(bytes32 channelId)
TokenPermissions(address token,uint256 amount)
```

The `witnessTypeString` passed to `permitWitnessTransferFrom` is the suffix beginning at `ChannelOpenWitness witness)` (or `ChannelTopUpWitness witness)`) followed by the witness struct definition and the `TokenPermissions` definition, per the Permit2 encoding rules.

Note that the Permit2 domain omits the `version` field. The canonical Permit2 contract is deployed at the same deterministic address on most major EVM chains. The client uses `methodDetails.permit2Contract` when present, and otherwise the canonical deterministic address, as `verifyingContract`. The escrow's configured Permit2 address **MUST** be the same one the client signed against; a mismatch makes the Permit2 signature fail verification on-chain. Servers **MUST** advertise `methodDetails.permit2Contract` on any chain whose Permit2 is not at the canonical address.

Example (EIP-3009):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "evm",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "action": "open",
    "type": "authorization",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "authorization": {
      "from": "0xaabbccddee11223344556677889900aabbccddee",
      "to": "0x1234567890abcdef1234567890abcdef12345678",
      "value": "10000000",
      "validAfter": "0",
      "validBefore": "1743523500",
      "nonce": "0xaaaa...aaaa"
    },
    "signature": "0xabcdef...eip3009sig",
    "cumulativeAmount": "0",
    "voucherSignature": "0x123456...vouchersig",
    "authorizedSigner": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
    "salt":
"0xaaaa1234bbbb5678cccc9012dddd3456eeee7890ffff1234aaaa5678bbbb9012"
  }
}
```

Example (Permit2):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "evm",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "action": "open",
    "type": "permit2",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "authorization": {
      "from": "0xaabbccddee11223344556677889900aabbccddee",
      "permitted": {
        "token": "0x74b7F16337b8972027F6196A17a631ac6dE26d22",
        "amount": "10000000"
      },
      "nonce": "1",
      "deadline": "1743523500",
      "witness": {
        "payee": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
        "salt":
"0xaaaa1234bbbb5678cccc9012dddd3456eeee7890ffff1234aaaa5678bbbb9012",
        "authorizedSigner": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00"
      }
    },
    "signature": "0xfedcba...permit2sig",
    "cumulativeAmount": "0",
    "voucherSignature": "0x123456...vouchersig",
    "authorizedSigner": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
    "salt":
"0xaaaa1234bbbb5678cccc9012dddd3456eeee7890ffff1234aaaa5678bbbb9012"
  }
}
```

The `authorization.witness.payee` equals the challenge `request.recipient`, and `witness.salt` / `witness.authorizedSigner` equal the top-level `salt` / `authorizedSigner` (they appear in both places because the witness is a faithful copy of what was signed, while the top-level fields are what the server passes to the escrow). The server treats the top-level channel parameters as authoritative when calling `openWithPermit2`; the Permit2 signature reverts on-chain if the witness it reconstructs disagrees. `spender` is not shown because Permit2 fixes it to the escrow (`msg.sender`).

10.2.3. TopUp Payload

The `topUp` action adds funds to an existing channel. It resets any pending close timer.

When feePayer: false (client broadcasts):

Field	Type	Required	Description
`action`	string	**REQUIRED**	`"topUp"`
`type`	string	**REQUIRED**	`"hash"`
`channelId`	string	**REQUIRED**	Channel ID
`hash`	string	**REQUIRED**	Tx hash of the on-chain topUp, direct or via ERC-4337 EntryPoint
`additionalDeposit`	string	**REQUIRED**	Additional amount deposited

Table 26

When `type="hash"`, the credential-level `source` field is **REQUIRED**, as described in the Credential Structure section.

When feePayer: true (server submits via EIP-3009 or Permit2):

Field	Type	Required	Description
`action`	string	**REQUIRED**	`"topUp"`
`type`	string	**REQUIRED**	`"authorization"` for EIP-3009, or `"permit2"` for Permit2
`channelId`	string	**REQUIRED**	Channel ID
`salt`	string	CONDITIONAL	Random bytes32 hex; passed as `topUpSalt` for EIP-3009 nonce derivation. **REQUIRED** when `type="authorization"`; omitted for `"permit2"` (the Permit2 path uses no `topUpSalt`)
`authorization`	object	**REQUIRED**	Token-pull authorization parameters; shape determined by the top-level `type`, using the same shapes defined in Section 10.2.2
`signature`	string	**REQUIRED**	Authorization signature

Field	Type	Required	Description
`additionalDeposit`	string	**REQUIRED**	Additional amount to deposit. **MUST** equal the authorization amount (`authorization.value` for `"authorization"`, `authorization.permitted.amount` for `"permit2"`)

Table 27

The top-level `additionalDeposit` and the authorization amount **MUST** be equal: the server passes this value as the `additionalDeposit` argument to `topUpWithAuthorization` / `topUpWithPermit2`, and the escrow binds it into the token-pull signature (EIP-3009 `value`, or Permit2 `permitted.amount`). A server **MUST** reject the credential if the two disagree.

10.2.4. Voucher Payload

The `voucher` action submits an updated cumulative voucher. For `action="voucher"` and `action="close"`, the `source` field is **OPTIONAL**; the server identifies the payer from the established channel state.

Field	Type	Required	Description
`action`	string	**REQUIRED**	`"voucher"`
`channelId`	string	**REQUIRED**	Channel identifier
`cumulativeAmount`	string	**REQUIRED**	Cumulative amount authorized
`signature`	string	**REQUIRED**	EIP-712 voucher signature

Table 28

Vouchers **MAY** carry an optional `deposit` field to merge a deposit authorization with the voucher update in a single round-trip. This allows the server to process both the funding and the new voucher atomically.

Field	Type	Required	Description
`deposit`	object	**OPTIONAL**	Deposit extension
`deposit.action`	string	**REQUIRED**	`"open"` or `"topUp"`
`deposit.type`	string	**REQUIRED**	`"authorization"` for EIP-3009, or `"permit2"` for Permit2

Field	Type	Required	Description
`deposit.authorization`	object	**REQUIRED**	Token-pull authorization parameters; shape determined by `deposit.type`, using the same shapes defined in Section 10.2.2
`deposit.signature`	string	**REQUIRED**	Authorization signature (65 bytes, hex-encoded)
`deposit.salt`	string	CONDITIONAL	Random bytes32 hex. Used for channelId computation when `deposit.action` is `"open"` (**REQUIRED**). When `deposit.action` is `"topUp"`, passed as `topUpSalt` for EIP-3009 nonce derivation (**REQUIRED** for `type="authorization"`; unused by `"permit2"`)
`deposit.authorizedSigner`	string	**OPTIONAL**	Address delegated to sign vouchers. Omitted ⇒ the zero address, which the contract treats as the payer; clients **MUST** use the zero address (not `authorization.from`) when deriving the EIP-3009 nonce or constructing the Permit2 witness. Only applicable when `deposit.action` is `"open"`

Table 29

When `deposit` is present, the server processes the deposit first by calling the matching escrow function (`openWithAuthorization` / `topUpWithAuthorization` for `deposit.type="authorization"`, or `openWithPermit2` / `topUpWithPermit2` for `deposit.type="permit2"`), then validates and accepts the voucher. If the deposit fails, the server **MUST** reject the entire credential.

`deposit.action: "open"` is an optimization pattern that allows the client to pre-compute the `channelId` deterministically (per the formula in the Channel State section) and bundle channel creation with the initial voucher in a single round-trip. Despite using `action="voucher"` in the payload, the server creates the channel as part of processing. The server **MUST** process the deposit first (calling `openWithAuthorization` or `openWithPermit2`), then validate the voucher against the newly created channel. For already-existing channels, `deposit.action` **MUST** be `"topUp"`. The escrow contract will revert if `open` is called on an existing `channelId`.

Example (voucher only):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "evm",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "action": "voucher",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "cumulativeAmount": "250000",
    "signature": "0xabcdef1234567890..."
  }
}
```

Example (voucher + deposit merge):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "evm",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "action": "voucher",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "cumulativeAmount": "15000000",
    "signature": "0xabcdef...vouchersig",
    "deposit": {
      "action": "topUp",
      "type": "authorization",
      "salt":
"0xcccc5678dddd9012eeee3456ffff7890aaaa1234bbbb5678cccc9012dddd3456",
      "authorization": {
        "from": "0xaabbccddee11223344556677889900aabbccddee",
        "to": "0x1234567890abcdef1234567890abcdef12345678",
        "value": "5000000",
        "validAfter": "0",
        "validBefore": "1743523500",
        "nonce": "0xbbbb...bbbb"
      },
      "signature": "0x789abc...eip3009sig"
    }
  }
}
```

10.2.5. Close Payload

The `close` action requests the server to close the channel and settle on-chain.

Field	Type	Required	Description
`action`	string	**REQUIRED**	`"close"`
`channelId`	string	**REQUIRED**	Channel identifier
`cumulativeAmount`	string	**REQUIRED**	Final cumulative amount
`signature`	string	**REQUIRED**	EIP-712 voucher signature

Table 30

The server calls `close(channelId, cumulativeAmount, signature)` on the escrow contract.

Example:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "evm",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "action": "close",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "cumulativeAmount": "500000",
    "signature": "0xabcdef1234567890..."
  }
}
```

11. Voucher Signing Format

Vouchers use [EIP-712] typed structured data signing.

11.1. Type Definitions

```
{
  "Voucher": [
    { "name": "channelId", "type": "bytes32" },
    { "name": "cumulativeAmount", "type": "uint128" }
  ]
}
```

11.2. Domain Separator

Field	Type	Value
`name`	string	`"EVM Payment Channel"`
`version`	string	`"1"`
`chainId`	uint256	EVM chain ID (e.g., 196)
`verifyingContract`	string	Escrow contract address

Table 31

Note: The domain `name` differs from Tempo's `"Tempo Stream Channel"`. This is the only semantic difference in the voucher signing scheme.

11.3. Signing Procedure

1. Construct the domain separator hash:

```
domainSeparator = keccak256(
  abi.encode(
    keccak256("EIP712Domain(string name,string version,uint256
chainId,address verifyingContract)"),
    keccak256(bytes("EVM Payment Channel")),
    keccak256(bytes("1")),
    chainId,
    verifyingContract
  )
)
```

2. Construct the struct hash:

```
structHash = keccak256(
  abi.encode(
    keccak256("Voucher(bytes32 channelId,uint128 cumulativeAmount)"),
    channelId,
    cumulativeAmount
  )
)
```

3. Compute the signing hash:

```
signingHash = keccak256(
  "\x19\x01" || domainSeparator || structHash
)
```

4. Sign with ECDSA using secp256k1 curve
5. Encode as a 65-byte `bytes` value: `r (32) || s (32) || v (1)`, where `v` is 27 or 28

11.4. Cumulative Semantics

Vouchers specify cumulative totals, not incremental deltas:

- Voucher #1: `cumulativeAmount = 100` (authorizes 100 total)
- Voucher #2: `cumulativeAmount = 250` (authorizes 250 total)
- Voucher #3: `cumulativeAmount = 400` (authorizes 400 total)

When settling, the contract computes: `delta = cumulativeAmount - settled`

Server **MUST** verify `cumulativeAmount <= 2^128 - 1` (uint128 upper bound). Vouchers exceeding this **MUST** be rejected.

12. Verification Procedure

12.1. Common Verification

For all actions, servers **MUST** perform the following steps before action-specific verification:

1. Decode the base64url credential and parse the JSON object
2. Verify `payload.action` is a recognized action
3. Look up the stored challenge using `credential.challenge.id`
4. Verify all fields in `credential.challenge` exactly match the stored challenge parameters
5. Verify the challenge has not expired

12.2. Transaction Outcome Checks

Whenever a server relies on an on-chain transaction — verifying a client-submitted `txHash` (`type="hash"`) or submitting one itself (`type="authorization"`, `type="permit2"`, `settle`, `close`) — it **MUST** read the receipt's `status` field and **MUST NOT** treat the transaction as effective on `status` alone being present.

- `status == 0x1` (success): the call's on-chain effects occurred; proceed to verify channel state.
- `status == 0x0` (reverted): the transaction was mined but **no state changed** (e.g. `NonceMismatch`, `AmountNotIncreasing`, `ChannelAlreadyExists`, or a failed token pull). The server **MUST** fail fast with a typed error and **MUST NOT** keep polling for a state change that will never come.
- No receipt yet (not mined): distinct from a revert; the server **MAY** await or retry up to a bounded timeout.

A reverted `status == 0x0` is a definitive negative outcome, not a pending one; conflating the two is what causes close/settlement paths to hang on a spinner.

12.3. Open Verification

On `action="open"`, servers **MUST**:

When `type="hash"`:

1. Verify the txHash via `eth_getTransactionReceipt`, checking `receipt.status` per Section 12.2 (a reverted `0x0` **MUST** be rejected immediately, not awaited)
2. Verify the transaction successfully caused `open()` to be executed on the expected escrow, either directly or through an ERC-4337 EntryPoint-mediated UserOperation
3. Verify that this execution created or initialized the specific `payload.channelId`
4. Query the escrow contract to verify channel state:
   - Channel exists with the provided `channelId`
   - `channel.payee` matches server's address
   - `channel.token` matches `request.currency`
   - `channel.deposit - channel.settled >= amount`
   - Channel is not finalized
   - `channel.closeRequestedAt == 0` (no pending close)
5. Verify the initial voucher signature (see Section 12.5)
6. Initialize server-side accounting state

When `type="authorization"` or `type="permit2"`:

1. Verify the `authorization` parameters according to the top-level `type`:
   - `"authorization"`: validate EIP-3009 fields and signature
   - `"permit2"`: validate Permit2 fields and signature
2. Call the matching escrow function:
   - `"authorization"`: `openWithAuthorization()`
   - `"permit2"`: `openWithPermit2()`
3. Verify channel state as above
4. Verify the initial voucher signature
5. Initialize server-side accounting state

12.4. TopUp Verification

On `action="topUp"`, servers **MUST**:

When `type="hash"`:

1. Verify the txHash shows a successful `topUp()` execution on the expected escrow (check `receipt.status` per Section 12.2; reject a reverted `0x0` immediately), either directly or through an ERC-4337 EntryPoint-mediated UserOperation
2. Verify that this execution affected the specific `payload.channelId`
3. Query updated channel state
4. Verify the channel deposit increased by exactly `payload.additionalDeposit`
5. Update server-side balance

When `type="authorization"` or `type="permit2"`:

1. Verify the `authorization` parameters according to the top-level `type`:
   - `"authorization"`: validate EIP-3009 fields and signature
   - `"permit2"`: validate Permit2 fields and signature
2. Call the matching escrow function:
   - `"authorization"`: `topUpWithAuthorization()`
   - `"permit2"`: `topUpWithPermit2()`
3. Verify updated channel state
4. Update server-side balance

12.5. Voucher Verification

On `action="voucher"`, servers **MUST**:

1. Verify `channel.closeRequestedAt == 0` (no pending close). Reject vouchers on channels with a pending forced close.
2. If `deposit` field is present, process deposit first:
   - For `deposit.type="authorization"`, call `openWithAuthorization` or `topUpWithAuthorization`
   - For `deposit.type="permit2"`, call `openWithPermit2` or `topUpWithPermit2`
   - Verify updated channel state
3. Verify voucher signature using EIP-712 recovery
4. Verify signature uses canonical low-s values
5. Recover signer and verify it matches the expected signer from on-chain state (`channel.authorizedSigner` if non-zero, otherwise `channel.payer`). For ERC-1271 contract wallets, verify via `isValidSignature`.
6. If `cumulativeAmount <= highestVoucherAmount`, return `200 OK` without changing state (idempotent replay).
7. Verify monotonicity:
   - `(cumulativeAmount - highestVoucherAmount) >= minVoucherDelta`
8. Verify `cumulativeAmount <= channel.deposit` (ensures the settlement delta `cumulativeAmount - channel.settled` does not exceed available funds `channel.deposit - channel.settled`)
9. Persist voucher to durable storage before providing service
10. Update `highestVoucherAmount = cumulativeAmount`

Signature and signer verification (steps 3-5) **MUST** be performed before the idempotency short-circuit in step 6. Because `voucher` and `close` credentials **MAY** omit the `source` field and identify the payer from channel state, returning `200 OK` for a stale `cumulativeAmount` before verifying the signature would let any party that knows a `channelId` trigger successful-looking responses, and — when the voucher is submitted alongside a service request — consume already-authorized balance on the payer's behalf.

Implementations **MAY** cache successfully-verified voucher signatures keyed by `(channelId, cumulativeAmount, signature)` and short-circuit on an exact bit-for-bit replay before re-running `ecrecover`. This optimization is safe because the cache hit itself proves the signature was previously verified.

12.6. Idempotency

Servers **MUST** treat voucher submissions idempotently **only after the voucher signature and signer have been verified** per steps 3-5 of Section 12.5:

- If signature verification fails, the server **MUST** return an error response per Section 12.7, regardless of how `cumulativeAmount` compares to `highestVoucherAmount`.
- After successful signature and signer verification:
  - If `cumulativeAmount == highestVoucherAmount`, the server **MUST** return `200 OK` without changing state.
  - If `cumulativeAmount < highestVoucherAmount`, the server **MUST** return `200 OK` without changing state.
  - Only vouchers with `cumulativeAmount > highestVoucherAmount` proceed to the monotonicity and balance checks in Section 12.5.

12.7. Error Responses

Status	When
400 Bad Request	Malformed payload or missing fields
402 Payment Required	Invalid signature or signer mismatch
409 Conflict	A submitted/verified on-chain transaction reverted (`receipt.status == 0x0`, see Section 12.2)
410 Gone	Channel finalized or not found

Table 32

Error responses use Problem Details [RFC9457]. Problem type URIs:

Type URI	Description
`https://paymentauth.org/problems/session/invalid-signature`	Voucher signature invalid
`https://paymentauth.org/problems/session/signer-mismatch`	Signer not authorized
`https://paymentauth.org/problems/session/amount-exceeds-deposit`	Exceeds channel deposit
`https://paymentauth.org/problems/session/delta-too-small`	Below `minVoucherDelta`

Type URI	Description
`https://paymentauth.org/problems/session/channel-not-found`	No channel with this ID
`https://paymentauth.org/problems/session/channel-finalized`	Channel closed
`https://paymentauth.org/problems/session/challenge-not-found`	Challenge unknown or expired
`https://paymentauth.org/problems/session/insufficient-balance`	Insufficient authorized balance
`https://paymentauth.org/problems/session/transaction-reverted`	An on-chain open/topUp/settle/close transaction reverted

Table 33

These problem types are the **interoperable error surface**: they are what a client consumes, and they **MUST** be uniform across deployments regardless of how each escrow names its internal reverts (Section 7.6). Accordingly, when a server-submitted transaction reverts (Section 12.2) — a `feePayer: true` open/topUp, a relayed settle/close — or when an action fails server-side validation, the server **MUST** report the failure using the most specific applicable problem type, falling back to `transaction-reverted` when none is more specific. Servers **MUST NOT** surface a raw on-chain revert selector or an implementation-specific error string as the problem `type`.

Recommended mapping from on-chain revert condition (Section 7.6) to problem type:

Revert condition	Problem type
`AmountExceedsDeposit`	`amount-exceeds-deposit`
`ChannelNotFound`	`channel-not-found`
`ChannelFinalized`	`channel-finalized`
`InvalidSignature`	`invalid-signature`
Insufficient on-chain balance / token pull failed	`insufficient-balance`
`NonceMismatch`, `NonceAlreadyUsed`, `AuthorizationExpired`, `AmountNotIncreasing`, `ChannelAlreadyExists`, or any other	`transaction-reverted`

Table 34

For `feePayer: false`, where the client broadcasts its own `open` / `topUp`, the client determines the outcome by querying channel state (Open/TopUp Verification, Section 12.2) rather than decoding the revert, so it likewise does not depend on the escrow's internal error encoding.

Example error response:

```
{
  "type": "https://paymentauth.org/problems/session/invalid-signature",
  "title": "Invalid Signature",
  "status": 402,
  "detail": "Voucher signature could not be verified",
  "channelId": "0x6d0f4fdf..."
}
```

13. Server-Side Accounting

Servers **MUST** maintain per-session accounting state:

Field	Type	Description
`acceptedCumulative`	uint128	Highest valid voucher amount accepted (monotonically increasing). Also referred to as `highestVoucherAmount` in the verification procedure
`spent`	uint128	Cumulative amount charged for delivered service (monotonically increasing)
`settledOnChain`	uint128	Last cumulative amount settled on-chain

Table 35

Available balance: `available = acceptedCumulative - spent`

13.1. Per-Request Processing

1. **Voucher acceptance**: Verify and persist new `acceptedCumulative`
2. **Balance check**: If `available < cost`, return 402
3. **Charge and deliver**: Persist `spent := spent + cost` BEFORE delivering service
4. **Receipt generation**: Include balance state

13.2. Crash Safety

- Persist `spent` increments BEFORE delivering service
- Persist `acceptedCumulative` BEFORE relying on new balance
- Use transactional storage or write-ahead logging

13.3. Insufficient Balance During Streaming

When balance is exhausted during a streaming response:

1. Server **MUST** stop delivering additional metered content
2. Server **MUST** emit a `payment-need-voucher` [SSE] event:

```
event: payment-need-voucher
data: {"channelId":"0x6d0f...",
  "requiredCumulative":"250025",
  "acceptedCumulative":"250000",
  "deposit":"500000"}
```

The `payment-need-voucher` event data:

Field	Type	Required	Description
`channelId`	string	**REQUIRED**	Channel identifier
`requiredCumulative`	string	**REQUIRED**	Minimum next voucher amount
`acceptedCumulative`	string	**REQUIRED**	Current highest accepted
`deposit`	string	**REQUIRED**	Current on-chain deposit

Table 36

When `requiredCumulative > deposit`, the client **MUST** submit a `topUp` before sending a new voucher.

Note: The SSE event types `payment-need-voucher` and `payment-receipt` are defined by this specification. They are not registered in any external event type registry.

13.4. Request Idempotency

To prevent double-charging on retries:

- Clients **SHOULD** include an `Idempotency-Key` header on paid requests
- Servers **SHOULD** track `(challengeId, idempotencyKey)` pairs and return cached responses for duplicates
- Servers **MUST NOT** increment `spent` for duplicate idempotent requests

13.5. Cost Calculation

Servers **MUST** support at least one of:

- **Fixed cost**: A predetermined amount per request

- **Usage-based**: Proportional to resource consumption (e.g., tokens generated, bytes transferred)

For streaming responses (SSE), servers **SHOULD**:

1. Reserve an estimated cost before starting delivery
2. Adjust `spent` as actual consumption is measured
3. Pause delivery if `available` is exhausted

14. Settlement Procedure

14.1. Settlement Timing

Servers **MAY** settle at any time:

- Periodically (every N seconds or M base units)
- When `action="close"` is received
- When unsettled amount exceeds a threshold
- Based on gas cost optimization

For every server-submitted `settle()` or `close()`, the server **MUST** check `receipt.status` per Section 12.2 and treat a reverted `0x0` as a definitive failure to settle (surface a typed error and retry as appropriate), never as a pending result to be polled to timeout.

14.2. Cooperative Close

When the client sends `action="close"`:

1. Server **MUST** verify `cumulativeAmount >= spent` (the client's final voucher covers all delivered service). If the client's voucher is insufficient, the server **SHOULD** settle using the highest previously accepted voucher instead of the close voucher
2. Server calls `close(channelId, cumulativeAmount, signature)`
3. Server **MUST** check the resulting `receipt.status` per Section 12.2. On a reverted `0x0`, the server **MUST NOT** report the channel as closed or block the client on a spinner; it **MUST** surface a typed error and **MAY** retry (e.g. re-submit with the highest accepted voucher, or after diagnosing the revert reason)
4. On `status == 0x1`, the contract has settled the delta and refunded the remainder to the payer; the server returns a receipt with the transaction hash

14.3. Forced Close

If the server does not respond:

1. Client calls `requestClose(channelId)` on-chain

2. Grace period begins (defined by the contract's `CLOSE_GRACE_PERIOD` constant; the reference value is 15 minutes. Compliant implementations **MUST NOT** use a grace period shorter than 10 minutes to ensure the server has reasonable time to settle outstanding vouchers. Servers **MUST** verify the contract's grace period meets this minimum before accepting channels on that contract)
3. Server can still `settle()` or `close()` during grace period
4. After grace period, client calls `withdraw(channelId)`
5. Client receives remaining (unsettled) funds

14.4. Sequential Sessions

A single channel supports sequential sessions. Each session uses the same cumulative voucher counter. The channel's `highestVoucherAmount` is the source of truth for the next voucher's minimum value.

14.5. Voucher Submission Transport

Vouchers are submitted via HTTP requests to the **same resource URI** that requires payment. There is no separate session endpoint. Clients **SHOULD** use HTTP/2 multiplexing or maintain separate connections for voucher updates and content streaming.

For voucher-only updates (no response body needed), clients **MAY** use `HEAD` requests.

14.6. Receipt Generation

Servers **MUST** return a `Payment-Receipt` header on **every successful paid request**. For streaming responses (SSE), servers **MUST** include the receipt in the initial response headers AND as a final SSE event:

```
event: payment-receipt
data: {"method":"evm","intent":"session","status":"success",...}
```

For chunked responses, the final receipt **MAY** be delivered as an HTTP trailer if the client advertises `TE: trailers`.

Field	Type	Required	Description
`method`	string	**REQUIRED**	`"evm"`
`intent`	string	**REQUIRED**	`"session"`
`status`	string	**REQUIRED**	`"success"`
`timestamp`	string	**REQUIRED**	[RFC3339] response time

Field	Type	Required	Description
`reference`	string	**REQUIRED**	Stable session reference; equal to `channelId`
`challengeId`	string	**REQUIRED**	Challenge identifier
`channelId`	string	**REQUIRED**	Channel identifier
`acceptedCumulative`	string	**REQUIRED**	Highest voucher accepted
`spent`	string	**REQUIRED**	Total amount charged
`chainId`	number	**REQUIRED**	EVM chain ID where settlement occurs
`units`	number	**OPTIONAL**	Units consumed this request
`txHash`	string	**OPTIONAL**	On-chain transaction hash (present on settlement/close)
`confirmations`	number	**OPTIONAL**	Block confirmations at receipt time

Table 37

The `reference` field is the core spec's stable receipt reference and **MUST** equal `channelId`. The `txHash` field is optional settlement evidence because not every response involves an on-chain settlement; voucher updates are off-chain. When present, `txHash` can also serve as a method-specific settlement reference.

Example receipt (per-request):

```
{
  "method": "evm",
  "intent": "session",
  "status": "success",
  "timestamp": "2026-04-01T12:08:30Z",
  "challengeId": "kM9xPqWvT2nJrHsY4aDfEb",
  "reference": "0x6d0f4fdf...",
  "channelId": "0x6d0f4fdf...",
  "chainId": 196,
  "acceptedCumulative": "250000",
  "spent": "237500",
  "units": 500
}
```

Example receipt (on close):

```
{
  "method": "evm",
  "intent": "session",
  "status": "success",
  "timestamp": "2026-04-01T12:10:00Z",
  "challengeId": "kM9xPqWvT2nJrHsY4aDfEb",
  "reference": "0x6d0f4fdf...",
  "channelId": "0x6d0f4fdf...",
  "chainId": 196,
  "acceptedCumulative": "250000",
  "spent": "250000",
  "txHash":
"0x1a2b3c4d5e6f7890abcdef1234567890abcdef1234567890abcdef1234567890"
}
```


15. Security Considerations

15.1. Replay Prevention

Vouchers are bound to a specific channel and contract via:

- `channelId` in the voucher message
- `verifyingContract` in EIP-712 domain
- `chainId` in EIP-712 domain
- Cumulative amount semantics (can only increase)

EIP-3009 nonces prevent replay of deposit authorizations at the contract level. Because the nonce is derived deterministically from the channel parameters (see Section 15.13), each unique `(payee, salt, authorizedSigner)` open or `(channelId, additionalDeposit, topUpSalt)` top-up produces a distinct nonce and the token contract rejects any reuse for the same `from`.

15.2. Channel Re-Use / Cross-Epoch Replay

`channelId` is derived from stable inputs only — `(payer, payee, token, salt, authorizedSigner, address(this), chainId)` — with no epoch or open-nonce. After a channel is closed, the same inputs (notably the same `salt`) re-derive a byte-identical `channelId`, and because the EIP-712 domain and `Voucher` struct are unchanged, an old voucher's digest is also byte-identical. If the contract permitted re-opening that `channelId`, the payee could replay a stale high-water voucher against the new deposit.

The escrow **MUST** prevent this by retaining finalized channel records permanently (sticky `finalized` flag, no struct deletion) so the open "already exists" check rejects every re-open of a used `channelId`. This matches the Tempo reference design, which likewise carries a persistent `finalized` flag and folds no epoch into `channelId`. Clients that want a fresh channel after close **MUST** choose a new `salt`.

15.3. Cross-Chain Replay

The EIP-712 domain separator includes `chainId`, making signatures invalid on other chains.

15.4. Voucher Tampering

EIP-712 signatures bind all voucher fields. Any modification invalidates the signature.

15.5. Rollback Prevention

Server **MUST** only accept strictly increasing `cumulativeAmount`, and the escrow contract **MUST** enforce the same on-chain in `settle` (reverting when `cumulativeAmount <= channel.settled`). Old vouchers are automatically superseded.

15.6. Overflow Protection

Server **MUST** verify `cumulativeAmount <= 2^128 - 1`. The escrow contract enforces the same constraint.

15.7. Deposit Cap

Server **MUST** verify `cumulativeAmount <= channel.deposit`. The escrow contract enforces this on-chain as well.

15.8. Denial of Service

- Rate limit voucher submissions (**SHOULD** limit to 10/second/session)
- Enforce `minVoucherDelta` to prevent tiny increments
- Enforce minimum deposit thresholds
- Perform format validation before signature recovery
- When `feePayer` is `true`, servers **SHOULD** enforce minimum deposit amounts to prevent gas griefing. A malicious client could sign many small EIP-3009 authorizations, forcing the server to spend gas on economically unprofitable `openWithAuthorization` calls.

15.9. Signature Malleability

ECDSA signatures have an inherent malleability: given a valid 65-byte signature containing components `(r, s, v)`, the value `(r, secp256k1_order - s, 55 - v)` is also valid for the same message. (Note: `55 - v` maps 27→28 and 28→27, which is the correct v-flip for EIP-712 signatures where v ∈ {27, 28}.) This could allow an attacker to submit a modified signature that passes `ecrecover` but references a different transaction hash.

The escrow contract **MUST** enforce canonical (low-s) signatures to prevent this. See the signature verification requirements in the Contract Functions section.

15.10. Reentrancy

Functions that transfer tokens out — `settle`, `close`, `withdraw`, and their relayed variants — **MUST** follow the checks-effects-interactions pattern: all state changes (`channel.settled`, `channel.finalized`, and the payee-relayed nonce used-set) **MUST** be committed before the external token transfer. The core functions are additionally protected by their `msg.sender` access checks (a re-entrant call from a malicious token carries `msg.sender == token` and fails the payer/payee check), but `settleWithAuthorization` and `closeWithAuthorization` are callable by any relayer, so implementations **MUST** apply a reentrancy guard or rely strictly on checks-effects-interactions for those paths. Restricting the escrow to well-behaved tokens without transfer callbacks (e.g. USDC) further reduces this surface.

15.11. No Voucher Expiry

Vouchers have no `validUntil` field. Channels have no expiry — they are closed explicitly. Vouchers remain valid until the channel closes. The close grace period protects against clients disappearing.

Operational guidance: Servers **SHOULD** settle and close channels that have been inactive for extended periods (e.g., 30+ days).

15.12. Chain Reorganization

If a chain reorganization removes a confirmed `open()` or `topUp()` transaction, the server loses its escrow guarantee. Mitigations:

- Servers **SHOULD** use sufficient confirmation depth before accepting open/topUp (e.g., 1 block for L2 rollups with fast finality, 12+ blocks for Ethereum mainnet)
- For L2 rollups, consider L1 settlement finality for high-value channels
- Voucher-based payments are not affected (off-chain)
- Settlement transactions should use appropriate gas pricing

15.13. Front-Running Protection

The escrow contract's `channelId` is deterministic. An attacker who observes a pending `open()` transaction could front-run it. However, the `channelId` includes the `payer` address (the `msg.sender` of `open()`), so a front-runner calling `open()` with identical parameters would produce a different `channelId` because their address differs. The `salt` parameter (chosen by the client) provides additional protection by making the `channelId` unpredictable before the transaction appears in the mempool.

When `feePayer` is `true`, the token-pull authorization signature is visible in the pending `openWith…` transaction. The same direct-call attack is mitigated at the token-transfer layer by both supported authorization formats:

- **EIP-3009**: The escrow contract **MUST** use `receiveWithAuthorization` (not `transferWithAuthorization`) when calling the token. This enforces `msg.sender == to`, so only the escrow contract can execute the transfer. The `authorization.to` field **MUST** be the escrow contract address.
- **Permit2**: The Permit2 contract fixes `spender = msg.sender` inside `permitWitnessTransferFrom`, so only the caller of `permitWitnessTransferFrom` (the escrow contract) can spend the signature. The `transferDetails.to` is also constrained to the escrow contract address. The channel parameters (`payee`, `salt`, `authorizedSigner` for `open`; `channelId` for `topUp`) are bound into the signature as a named EIP-712 witness, so an attacker who substitutes any of those values when calling the escrow causes the Permit2 signature verification to revert.

The two paths achieve channel-parameter integrity by different mechanisms:

- Permit2 signs `permitted.token`, `permitted.amount`, `spender`, `nonce`, `deadline`, **and** the witness struct, which carries the channel parameters explicitly. No nonce derivation is required: any unused Permit2 nonce is acceptable, and clients **MAY** use random or sequential nonces.
- EIP-3009 signs only `from`, `to`, `value`, `validAfter`, `validBefore`, `nonce`. It has no witness mechanism, so the channel parameters (`payee`, `salt`, `authorizedSigner` for `open`; `channelId`, `topUpSalt` for `topUp`) are not covered by the underlying signature. An attacker could front-run `openWithAuthorization` with a different `payee` (their own address), and the underlying transfer signature would remain valid.

To close this gap on the EIP-3009 path, compliant escrow contracts **MUST** derive the EIP-3009 nonce deterministically from the channel parameters and **MUST** validate the caller-supplied `nonce` argument against the derived value, reverting (e.g., with `NonceMismatch()`) on any mismatch:

```
// openWithAuthorization (EIP-3009 nonce, bytes32)
nonce = keccak256(abi.encode(from, payee, token, salt, authorizedSigner))

// topUpWithAuthorization (EIP-3009 nonce, bytes32)
nonce = keccak256(abi.encode(channelId, additionalDeposit, from, topUpSalt))
```

The contract passes the validated nonce to `receiveWithAuthorization`. If an attacker calls the escrow with a substituted `payee`, `salt`, `authorizedSigner`, `channelId`, `additionalDeposit`, or `topUpSalt`, the derived nonce differs from the one the payer signed; the contract **MUST** reject the call before invoking the token, and even if it did not, the underlying signature verification at the token contract would revert.

The `nonce` is exposed as an explicit function parameter (rather than derived silently) so that callers and indexers can observe the value the payer signed; the on-chain check is what makes the binding non-bypassable. Implementations **MUST NOT** skip this check, and **MUST NOT** fall back to using the caller-supplied value unchanged.

Clients **MUST** use the same derivation formula when signing the EIP-3009 authorization and **MUST** transmit the derived nonce in the credential. Including `from` ensures the nonce is bound to the depositor identity, even though the underlying signature already covers `from` directly.

Trade-off note: The deterministic-nonce approach for EIP-3009 trades signing-time UX (the nonce appears as an opaque hash in wallet displays) for protocol simplicity. The Permit2 witness approach trades a slightly longer EIP-712 type string for intent-visible UX — wallets that render typed data show `payee`, `salt`, and `authorizedSigner` as labeled fields the user can review.

15.14. ERC-20 Approval Front-Running

When `feePayer` is `false`, the client calls `approve(escrow, deposit)` followed by `open()`. The classic ERC-20 approval front-running attack (where a spender races to spend both the old and new allowance) does not apply here because the escrow contract is trusted code with deterministic behavior. However, clients **SHOULD** batch `approve` and `open` in a single transaction when possible (e.g., via ERC-4337 UserOperations or multicall) to minimize the window between approval and channel creation.

When `feePayer` is `true` and `type="permit2"`, the payer must have previously approved the canonical Permit2 contract for the token (typically a one-time, unlimited approval). The same reasoning applies: Permit2 is trusted code with deterministic behavior, and each Permit2 `SignatureTransfer` is gated by a single-use unordered nonce.

15.15. Contract Wallet Signer Mutability

When `authorizedSigner` is an ERC-1271 contract wallet (e.g., Safe, ERC-4337 account), voucher validity depends on the wallet's current signer state at verification time, not at signing time. If the wallet's owner set, signing key, or signature-validation policy changes after a voucher is signed, `isValidSignature` **MAY** return failure for previously-signed vouchers, rendering them unredeemable on-chain.

This creates an asymmetric risk:

- **Payee risk**: If the payer rotates keys on their contract wallet after signing vouchers but before the payee calls `settle()`, the payee loses the ability to redeem accumulated off-chain authorizations.
- **Payer mitigation**: Payers using contract wallets as `authorizedSigner` **SHOULD** avoid key rotation during active sessions, or coordinate rotation with settlement.
- **Payee mitigation**: Payees **SHOULD** settle more frequently when the `authorizedSigner` is a contract wallet, reducing the value at risk from signer-state changes. Payees **MAY** inspect the signer address to determine whether it is a contract (via `EXTCODESIZE`) and adjust settlement cadence accordingly.

EOA signers are not affected: ECDSA recovery is stateless and depends only on the signature and message.

Recommended pattern: When the payer is a contract wallet, the payer **SHOULD** delegate voucher signing to an ephemeral EOA session key by setting `authorizedSigner` to that EOA's address, rather than leaving `authorizedSigner` unset (which defaults to the payer contract wallet). This preserves the AA benefits for the escrowed funds — the contract wallet still controls `open()`, `topUp()`, and `close()` calls — while eliminating the signer-mutability risk for off-chain vouchers. The session key **SHOULD** be scoped to the lifetime of the channel and discarded after `close()`.

Contract-wallet `authorizedSigner` remains permitted for cases where EOA delegation is not acceptable (e.g., enterprise multi-sig policies that require every signed artifact to carry a quorum signature). In such cases, the mitigations above apply.

15.16. Escrow Guarantees

The escrow contract provides the following security properties:

- **Payer protection**: Funds can only be withdrawn with a valid voucher signature. Forced close + grace period ensures payer can always recover uncommitted funds.
- **Payee protection**: A valid voucher is an irrevocable on-chain claim. The payee can call `settle()` at any time.
- **Atomicity**: `close()` settles and refunds in a single transaction.

15.17. Disconnection Handling

Scenario	Handling
Client disappears	Server holds last voucher, can `settle()` unilaterally
Server crashes	Server persists vouchers, can `settle()` on restart
Session idle timeout	Server settles and closes after configured threshold

Table 38

16. IANA Considerations

16.1. Payment Method Registration

The `evm` payment method is registered by [I-D.evm-charge]. This document does not create a separate registration.

16.2. 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 `evm` payment method implements the registered `session` intent.

16.3. Problem Type Registration

This document registers the following problem types:

Type URI	Title	Status
`.../session/invalid-signature`	Invalid Signature	402
`.../session/signer-mismatch`	Signer Mismatch	402
`.../session/amount-exceeds-deposit`	Amount Exceeds Deposit	402
`.../session/delta-too-small`	Delta Too Small	402
`.../session/channel-not-found`	Channel Not Found	410
`.../session/channel-finalized`	Channel Finalized	410
`.../session/challenge-not-found`	Challenge Not Found	402
`.../session/insufficient-balance`	Insufficient Balance	402
`.../session/transaction-reverted`	Transaction Reverted	409

Table 39

Base URI: `https://paymentauth.org/problems`

17. References

17.1. 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>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[RFC9111] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <https://www.rfc-editor.org/info/rfc9111>.

[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>.

[EIP-712] Bloemen, R., Logvinov, L., and J. Evans, "Typed structured data hashing and signing", September 2017, <https://eips.ethereum.org/EIPS/eip-712>.

[EIP-3009] Kim, P. J., Britz, K., and D. Knott, "Transfer With Authorization", September 2020, <https://eips.ethereum.org/EIPS/eip-3009>.

[Permit2] Uniswap Labs, "Permit2: Token Approvals for the Next Generation of DeFi", December 2022, <https://github.com/Uniswap/permit2>.

[I-D.evm-charge] Wong, M., "EVM Charge Intent for HTTP Payment Authentication", 2026, <https://datatracker.ietf.org/doc/draft-evm-charge/>.

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

[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/>.

17.2. Informative References

[EIP-55] "Mixed-case checksum address encoding", n.d., <https://eips.ethereum.org/EIPS/eip-55>.

[EIP-2098] Moore, R. and N. Johnson, "Compact Signature Representation", March 2019, <https://eips.ethereum.org/EIPS/eip-2098>.

[ERC-20] "Token Standard", n.d., <https://eips.ethereum.org/EIPS/eip-20>.

[ERC-4337] "Account Abstraction Using Alt Mempool", n.d., <https://eips.ethereum.org/EIPS/eip-4337>.

[DID-PKH] W3C Credentials Community Group, "did:pkh Method Specification", 2022, <https://github.com/w3c-ccg/did-pkh/blob/main/did-pkh-method-draft.md>.

[I-D.tempo-session] Horne, L., Konstantopoulos, G., Robinson, D., Ryan, B., and J. Moxey, "Tempo Session Intent for HTTP Payment Authentication", 2026, <https://datatracker.ietf.org/doc/draft-tempo-session/>.

[SSE] WHATWG, "Server-Sent Events", n.d., <https://html.spec.whatwg.org/multipage/server-sent-events.html>.

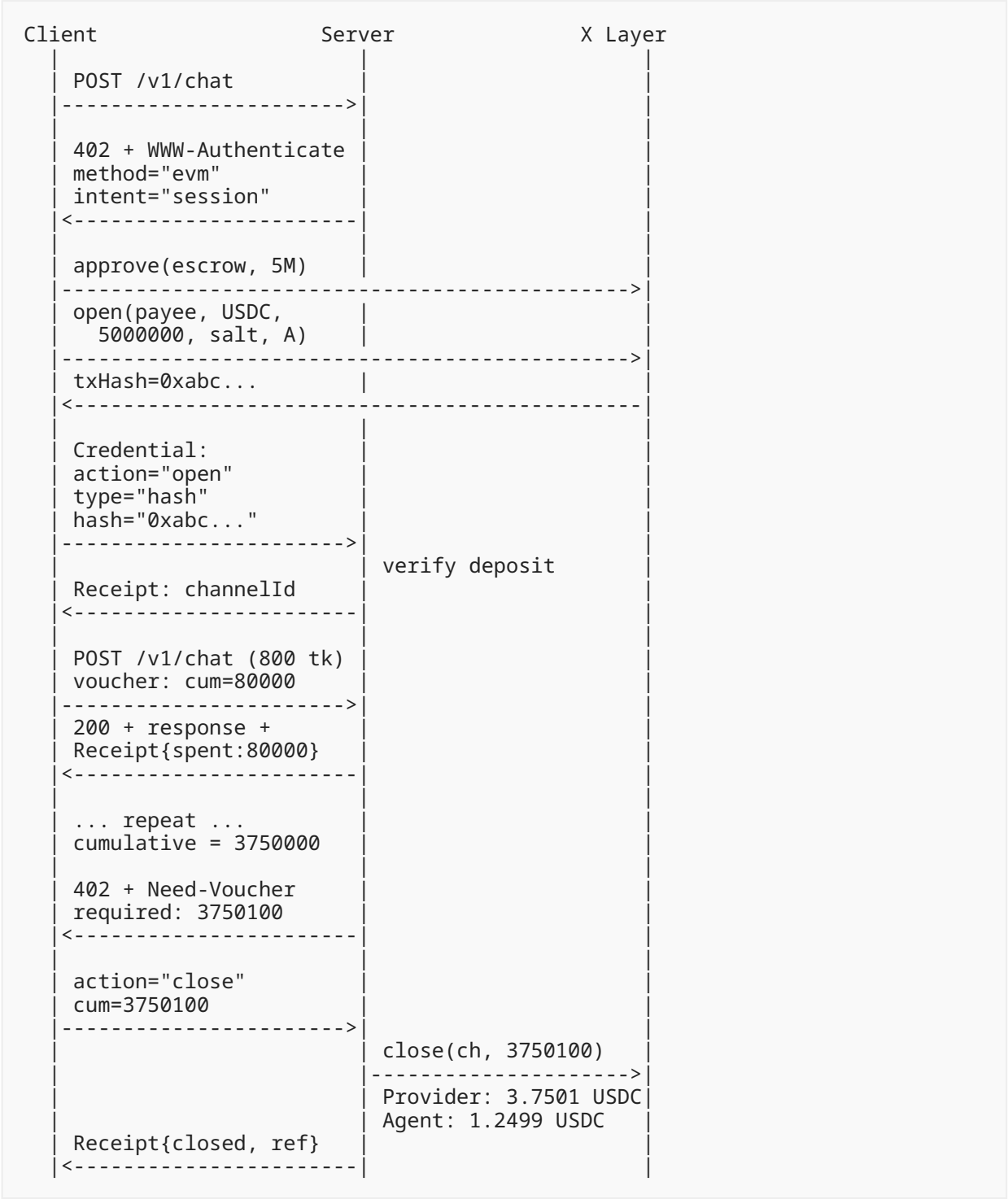
Appendix A. Scenario Walkthroughs

A.1. LLM Token Billing (Escrow + High-Frequency Voucher)

Agent A calls Provider P's LLM inference API, per-token billing on X Layer with USDC.

Parameters:

- Unit price: 100 base units per token = 0.0001 USDC (6 decimals)
- Suggested deposit: 5,000,000 = 5.0 USDC (~50,000 tokens)
- minVoucherDelta: 10,000 = 0.01 USDC (100 tokens per voucher)



Key numbers:

- Total consumed: 3,750,100 base units = 3.7501 USDC
- Refunded: 5,000,000 - 3,750,100 = 1,249,900 = 1.2499 USDC

- On-chain transactions: only 2 (open + close), all intermediate vouchers are off-chain

A.2. LLM Token Billing (Deposit Merge Mode)

Same setup as Scenario 1, but using feePayer: true + deposit merge mode. Consumer pays zero gas.

Key advantages:

- Consumer needs no native token for gas
- Voucher + deposit merged into single HTTP round-trip

- Server batches on-chain operations for gas optimization

Appendix B. Acknowledgements

The authors thank the Tempo Labs team for the foundational session payment channel design and the MPP community for their feedback.

Authors' Addresses

Xin Tian
OKG
Email: xin.tian@okg.com

Eason Wang
OKG
Email: wangyuxin@okg.com

Michael Wong
OKG
Email: michael.wong@okg.com

Aaron Zhou
OKG
Email: guoliang.zhou@okg.com