Workgroup:	Network Working Group	
Internet-Draft:	draft-hedera-session-00	
Published:	7 August 2026	
Intended Status:	Informational	
Expires:	8 February 2027	
Authors:	T. Rowbotham	L. Walker
		Hedera / Hashgraph

Hedera Session Intent for HTTP Payment Authentication

Abstract

This document defines the "session" intent for the "hedera" payment method in the Payment HTTP Authentication Scheme. It specifies unidirectional streaming payment channels for incremental, voucher-based payments suitable for low-cost metered services on the Hedera network.

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 8 February 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. Payment method specifications that reference this document inherit these requirements.

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.

The escrow contract (HederaStreamChannel.sol) is deployed on Hedera's EVM layer and uses standard ERC-20 token transfers. Hedera Token Service (HTS) tokens are exposed as ERC-20 via [HIP-218], enabling payment channels with native HTS tokens such as Circle USDC [CIRCLE-USDC-HEDERA].

1.1. Use Case: LLM Token Streaming

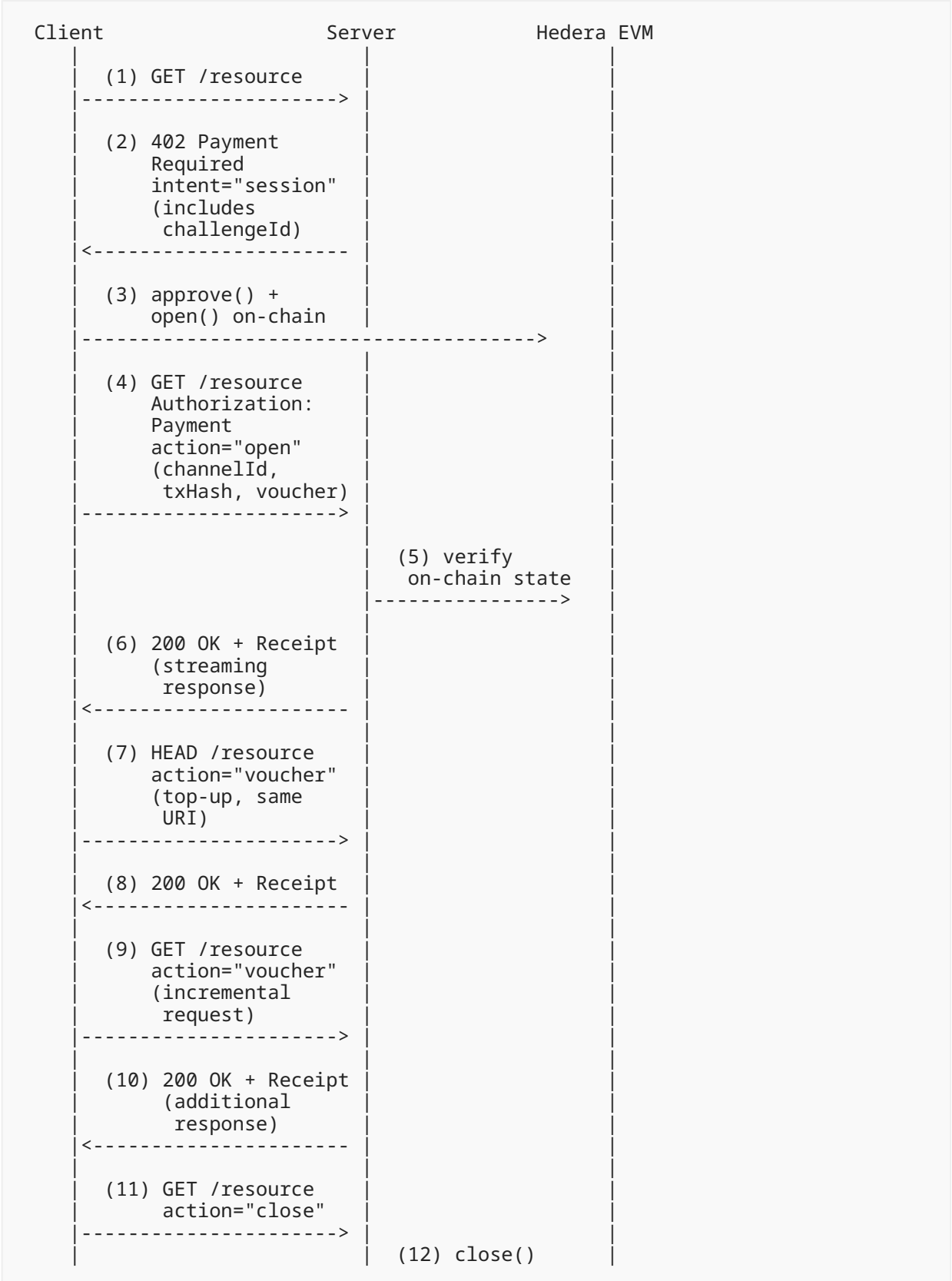
Consider an LLM inference API that charges per output token:

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 into the HederaStreamChannel escrow
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.

1.2. Session Flow

The following diagram illustrates the Hedera session flow:


Client
Server
Hedera EVM
(1) GET /resource
(2) 402 Payment Required intent="session" (includes challengeId)
(3) approve() + open() on-chain
(4) GET /resource Authorization: Payment action="open" (channelId, txHash, voucher)
(5) verify on-chain state
(6) 200 OK + Receipt (streaming response)
(7) HEAD /resource action="voucher" (top-up, same URI)
(8) 200 OK + Receipt
(9) GET /resource action="voucher" (incremental request)
(10) 200 OK + Receipt (additional response)
(11) GET /resource action="close"
(12) close()

```
     |                          |-----------------> |
     |                          |                   |
     |  (13) 200 OK + Receipt   |                   |
     |       (includes          |                   |
     |        txHash)           |                   |
     |<-------------------------|                   |
     |                          |                   |
```

Unlike Tempo session where the client sends a signed transaction for the server to broadcast, in the Hedera session the client broadcasts the `open` (and `topUp`) transactions directly via Hashio JSON-RPC and presents the transaction hash to the server. The server verifies the on-chain state.

Voucher updates and close requests are submitted to the **same resource URI** that requires payment. This allows sessions to work on any endpoint without dedicated payment control plane routes. Servers **SHOULD** support voucher updates via any HTTP method; clients **MAY** use `HEAD` for pure voucher top-ups when no response body is needed.

1.3. 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.

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 payment 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. HTS tokens are transferred via standard ERC-20 interfaces exposed through [HIP-218].

Authorized Signer An address delegated to sign vouchers on behalf of the payer. Defaults to the payer if not specified.

Base Units The smallest indivisible unit of an HTS token. For example, Circle USDC on Hedera uses 6 decimal places; one million base units equals 1.00 USDC.

Hashio JSON-RPC Hedera's EVM-compatible JSON-RPC relay that enables standard Ethereum tooling (e.g., viem, ethers.js) to interact with smart contracts deployed on Hedera's EVM layer.

4. Encoding Conventions

This section defines normative encoding rules for interoperability.

4.1. 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)	`0x742d...f8fe00`
bytes32	66 chars (0x + 64 hex)	`0x6d0f...8e9f`
signature	130-132 chars	65-byte r s v

Table 1

Implementations **MUST** use lowercase hex. Implementations **SHOULD** accept mixed-case input but normalize to lowercase before comparison.

Note: Hedera "long-zero" EVM addresses (e.g., `0x0000000000000000000000000000000000001549` for HTS token 0.0.5449) are valid 20-byte addresses and **MUST** be handled correctly. Implementations **MUST** use case-insensitive comparison for all address fields.

4.2. Numeric Values

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

Field	Encoding	Example
`cumulativeAmount`	Decimal string	`"250000"`
`requestedAt`	Decimal string	`"1736165100"`
`chainId`	JSON number	`296`

Table 2

The `chainId` uses JSON number encoding as values are small enough to avoid precision issues.

4.3. Timestamp Format

HTTP headers and receipt fields use [RFC3339] formatted timestamps: `2026-04-12T12:05:00Z`. Timestamps in EIP-712 signed data use Unix seconds as decimal strings.

5. Channel Escrow Contract

Streaming payment channels require an on-chain escrow contract that holds user deposits and enforces voucher-based withdrawals. On Hedera, this contract is deployed on the EVM layer and interacts with HTS tokens via their ERC-20 interface [HIP-218].

5.1. 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 (HTS via HIP-218)
`authorizedSigner`	address	Authorized signer (0 = payer)
`deposit`	uint128	Total amount deposited
`settled`	uint128	Cumulative amount withdrawn by payee
`closeRequestedAt`	uint64	Timestamp when close was requested (0 if not)

Field	Type	Description
`finalized`	bool	Whether channel is closed

Table 3

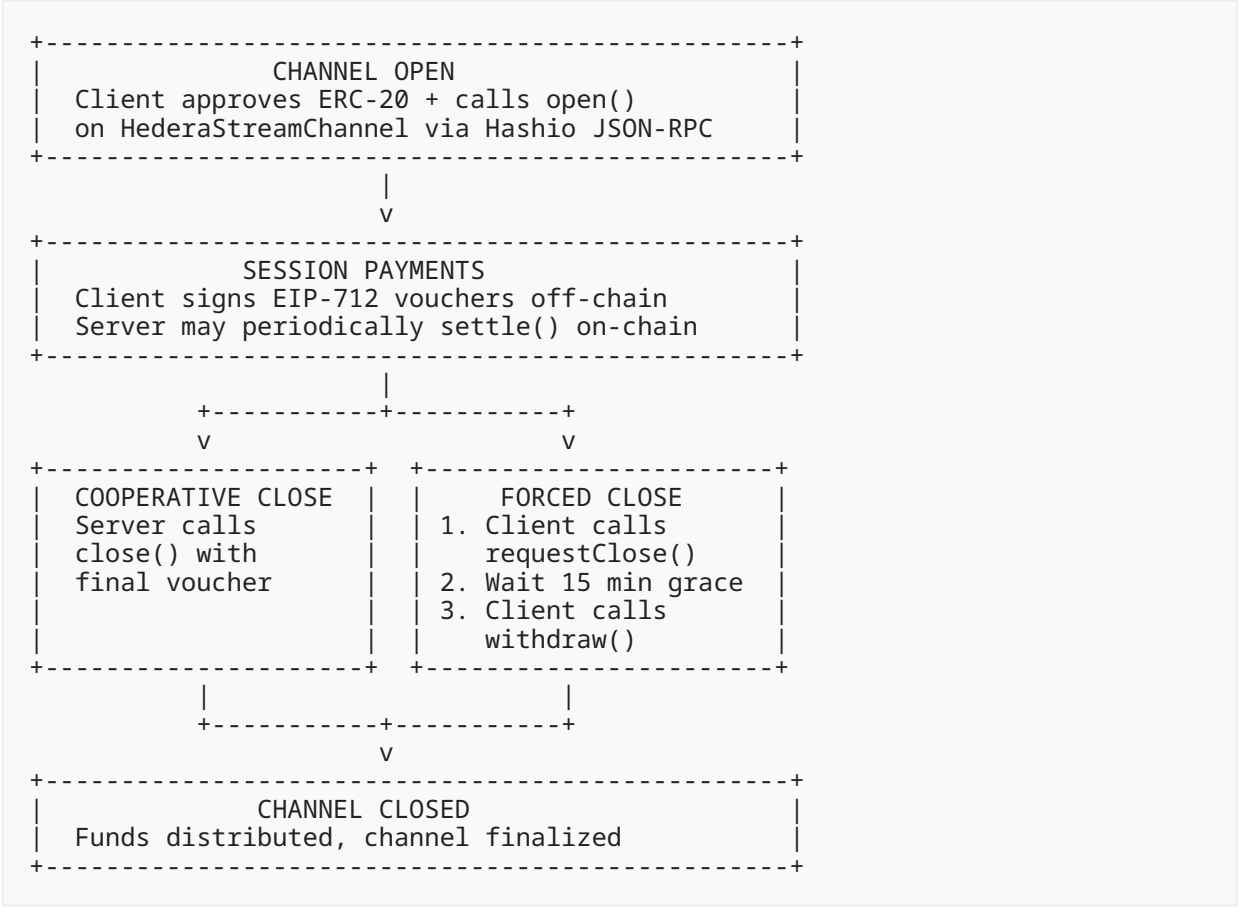
The `channelId` **MUST** be computed deterministically using the escrow contract's `computeChannelId()` function:

```
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. Clients **MUST** use the contract's `computeChannelId()` function or equivalent logic to ensure interoperability.

5.2. Channel Lifecycle

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



5.3. Contract Functions

Compliant escrow contracts **MUST** implement the following functions. The signatures shown are the reference HederaStreamChannel.sol implementation.

5.3.1. open

Opens a new channel with escrowed funds.

Parameter	Type	Description
`payee`	address	Server's withdrawal address
`token`	address	ERC-20 token contract address
`deposit`	uint128	Amount to deposit in base units
`salt`	bytes32	Random value for channelId
`authorizedSigner`	address	Delegated signer; `0x0` = payer

Table 4

Returns the computed `channelId`.

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

The client **MUST** approve the escrow contract to spend `deposit` tokens before calling `open()`. On Hedera, HTS token approvals via the ERC-20 interface require higher gas limits (approximately 1,000,000 gas) due to the HTS precompile overhead.

5.3.2. settle

Server withdraws funds using a signed voucher without closing the channel.

Parameter	Type	Description
`channelId`	bytes32	Channel identifier
`cumulativeAmount`	uint128	Cumulative total authorized
`signature`	bytes	EIP-712 signature

Table 5

The contract computes `delta = cumulativeAmount - channel.settled` and transfers `delta` tokens to the payee.

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

5.3.3. topUp

User adds more funds to an existing channel. If a close request is pending (`closeRequestedAt != 0`), calling `topUp()` **MUST** cancel it by resetting `closeRequestedAt` to zero and emitting a `CloseRequestCancelled` event.

Parameter	Type	Description
`channelId`	bytes32	Existing channel identifier
`additionalDeposit`	uint256	Additional amount in base units

Table 6

```
function topUp(
    bytes32 channelId,
    uint256 additionalDeposit
) external;
```

Note: The `additionalDeposit` parameter is `uint256` (not `uint128`) in HederaStreamChannel.sol; the contract checks for overflow internally.

5.3.4. close

Server closes the channel, settling any outstanding voucher and refunding the remainder to the payer. Only callable by the payee.

Parameter	Type	Description
`channelId`	bytes32	Channel to close
`cumulativeAmount`	uint128	Final cumulative amount
`signature`	bytes	EIP-712 signature

Table 7

Transfers `cumulativeAmount - channel.settled` to payee, refunds `channel.deposit - cumulativeAmount` to payer, and marks channel finalized.

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

5.3.5. requestClose

User requests channel closure, starting a grace period of at least 15 minutes.

Parameter	Type	Description
`channelId`	bytes32	Channel to request closure for

Table 8

Sets `channel.closeRequestedAt` to current block timestamp. The grace period allows the payee time to submit any outstanding vouchers before forced closure.

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

5.3.6. withdraw

User withdraws remaining funds after the grace period expires.

Parameter	Type	Description
`channelId`	bytes32	Channel to withdraw from

Table 9

Requires `block.timestamp >= channel.closeRequestedAt + CLOSE_GRACE_PERIOD`. Refunds all remaining deposit to payer and marks channel finalized.

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

5.3.7. associateSelf

Associates the escrow contract with an HTS token so it can receive transfers. This is a Hedera-specific function with no Tempo equivalent.

Parameter	Type	Description
`token`	address	HTS token to associate

Table 10

```
function associateSelf(
    address token
) external returns (int256 responseCode);
```

This function calls the HTS precompile at address `0x167` to perform token association. Anyone can call it. The escrow contract **MUST** be associated with the payment token before channels using that token can be opened.

5.4. Access Control

The escrow contract **MUST** enforce the following access control:

Function	Caller	Description
`open`	Anyone	Creates channel; caller = payer
`settle`	Payee only	Withdraws with voucher
`topUp`	Payer only	Adds funds
`close`	Payee only	Closes with final voucher

Function	Caller	Description
`requestClose`	Payer only	Initiates forced close
`withdraw`	Payer only	Withdraws after grace
`associateSelf`	Anyone	HTS token association

Table 11

5.5. Signature Verification

The escrow contract **MUST** perform the following signature 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 `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E73 57A4501DDFE92F46681B20A0`. See Section 13.8 for rationale.
2. **Authorized signer verification**: The contract **MUST** recover the signer address from the EIP-712 signature and verify it matches the expected signer:
   - If `channel.authorizedSigner` is non-zero, the recovered signer **MUST** equal `channel.authorizedSigner`
   - Otherwise, the recovered signer **MUST** equal `channel.payer`
3. **Domain binding**: The contract **MUST** use its own address as the `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.

6. Request Schema

The `request` parameter in the `WWW-Authenticate` challenge contains a base64url-encoded JSON object.

6.1. Fields

Field	Type	Required	Description
`amount`	string	**REQUIRED**	Price per unit in base units
`unitType`	string	**OPTIONAL**	Unit being priced (e.g., `"llm_token"`)
`suggestedDeposit`	string	**OPTIONAL**	Suggested deposit in base units

Field	Type	Required	Description
`currency`	string	**REQUIRED**	ERC-20 token address (HTS via HIP-218)
`recipient`	string	**REQUIRED**	Payee address (server's withdrawal address)

Table 12

For the `session` intent, `amount` specifies the price per unit of service in base units (e.g., 6 decimals for USDC), not a total charge. When `unitType` is present, clients can use it together with `amount` to estimate costs before streaming begins. The total cost depends on consumption: `total = amount * units_consumed`.

The optional `suggestedDeposit` indicates the server's recommended channel deposit for typical usage. Clients **MAY** deposit less (if they expect limited usage) or more (for extended sessions). The minimum viable deposit is implementation-defined but **SHOULD** be at least `amount` to cover one unit of service.

Challenge expiry is specified via the `expires` auth-param in the `WWW-Authenticate` header per [I-D.httpauth-payment], using [RFC3339] timestamp format. Unlike the `charge` intent, the session request JSON does not include an `expires` field -- expiry is conveyed solely via the HTTP header.

6.2. Method Details

As of version 00, session-specific request fields are placed in `methodDetails`. A future high-level "session" intent definition may promote common fields to the core schema.

Field	Type	Required	Description
`methodDetails.escrowContract`	string	**REQUIRED**	Escrow contract address
`methodDetails.channelId`	string	**OPTIONAL**	Channel ID if resuming
`methodDetails.minVoucherDelta`	string	**OPTIONAL**	Minimum voucher increment
`methodDetails.chainId`	number	**OPTIONAL**	Hedera chain ID (default: 295)

Table 13

Note: Unlike the Tempo session spec, there is no `feePayer` field in this version. Hedera supports native fee delegation via `feePayerAccountId` but this is deferred to a future revision (see Section 7.3).

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

- **New channel** (no `channelId`): Client generates a random salt locally, computes `channelId` using the formula in Section 5.1, 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. If insufficient, client **SHOULD** either call `topUp()` with the difference or open a new channel.

Servers **MAY** cache `(payer address, payee address, token) -> channelId` mappings to suggest channel reuse, reducing on-chain transactions.

Example (new channel):

```
{
  "amount": "25",
  "unitType": "llm_token",
  "suggestedDeposit": "10000000",
  "currency": "0x0000000000000000000000000000000000006f89a",
  "recipient": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
  "methodDetails": {
    "escrowContract":
      "0x8Aaf6690C2a6397d595F97E224fC19759De6fdaE",
    "chainId": 295
  }
}
```

This requests a price of 0.000025 USDC per LLM token, with a suggested deposit of 10.00 USDC (10000000 base units). The `currency` is Circle USDC on Hedera mainnet (HTS token 0.0.456858, exposed as ERC-20 via HIP-218).

Example (existing channel):

```
{
  "amount": "25",
  "unitType": "llm_token",
  "currency": "0x0000000000000000000000000000000000006f89a",
  "recipient": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
  "methodDetails": {
    "escrowContract":
      "0x8Aaf6690C2a6397d595F97E224fC19759De6fdaE",
    "channelId":
      "0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b"
      "1f6a9c1b3e2d4a5b6c7d8e9f",
    "chainId": 295
  }
}
```

For existing channels, `suggestedDeposit` is omitted since the channel already has funds. The `channelId` tells the client to resume this channel.

7. Fee Payment

7.1. Client-Paid Fees (Default)

In this version, the client pays all transaction fees for channel operations (`open`, `topUp`, ERC-20 `approve`). The client broadcasts these transactions directly via Hashio JSON-RPC.

Hedera's EVM layer has predictable, low transaction fees. However, HTS precompile interactions require higher gas limits than standard ERC-20 operations:

Operation	Recommended Gas Limit
ERC-20 `approve` (HTS)	1,000,000
`open`	1,500,000
`topUp`	1,500,000
`settle`	1,500,000
`close`	1,500,000

Table 14

Clients **MUST** set gas limits appropriate for HTS precompile operations. The default gas estimates from Hashio JSON-RPC may be insufficient.

7.2. Server-Initiated Operations

The `settle` and `close` contract functions are server-originated on-chain transactions. The server pays transaction fees for these operations:

- **Voucher updates** (`action="voucher"`) are off-chain and incur no transaction fees.
- **Settlement** (`settle()`) and channel **close** (`close()`) are initiated by the server using the highest valid voucher. The server covers the fees.
- Servers **MAY** recover settlement costs through pricing or other business logic.

7.3. Fee Delegation (Future)

Hedera natively supports fee delegation via the `feePayerAccountId` field on transactions. A future revision of this specification **MAY** add `feePayer` support to `methodDetails`, enabling the server to pay transaction fees on behalf of the client. This would pair naturally with a pull-mode open flow where the client signs the transaction and the server broadcasts it.

8. Credential Schema

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

8.1. Credential Structure

Field	Type	Required	Description
`challenge`	object	**REQUIRED**	Echo of the challenge parameters
`payload`	object	**REQUIRED**	Session-specific payload

Table 15

Implementations **MUST** ignore unknown fields in credential payloads, request objects, and receipts to allow forward-compatible extensions.

8.2. Credential Lifecycle

A streaming payment session progresses through distinct phases, each corresponding to a payload action:

1. **Open**: Client deposits funds on-chain (broadcasting the transaction directly) and presents the `open` action with the transaction hash. The server verifies the on-chain deposit and validates the initial voucher.
2. **Streaming**: Client submits `voucher` actions with increasing cumulative amounts as service is consumed. The server may periodically settle vouchers on-chain.
3. **Close**: Client sends the `close` action with the final voucher. The server settles on-chain and returns a receipt.

Each action carries action-specific fields directly in the `payload` object, with the `action` field discriminating between phases.

8.3. Payload Actions

The `payload` object uses an `action` discriminator with action-specific fields at the same level:

Field	Type	Required	Description
`action`	string	**REQUIRED**	One of the actions below

Table 16

Action	Description
`open`	Confirms channel is open on-chain
`topUp`	Adds funds to an existing channel
`voucher`	Submits updated cumulative voucher
`close`	Requests server to close channel

Table 17

8.3.1. Open Payload

The `open` action confirms an on-chain channel opening and begins the streaming session. Unlike the Tempo session where the client sends a signed transaction for server broadcast, the Hedera client broadcasts the `open()` transaction itself and presents the transaction hash.

Payload fields (in addition to `action`):

Field	Type	Required	Description
`channelId`	string	**REQUIRED**	Channel identifier (hex bytes32)
`txHash`	string	**REQUIRED**	Transaction hash from open()
`cumulativeAmount`	string	**REQUIRED**	Initial authorized amount (see below)
`signature`	string	**REQUIRED**	EIP-712 voucher signature

Table 18

The client broadcasts the `open()` transaction via Hashio JSON-RPC, waits for the transaction receipt, and presents the `txHash` for server verification.

The server uses the `txHash` to verify the on-chain channel state: deposit amount, payee, token, and that the channel is not finalized.

The initial voucher (`cumulativeAmount` and `signature`) proves the client controls the signing key and establishes the voucher chain. Implementations **MAY** set `cumulativeAmount` to zero or to the first request's cost; both are valid starting points for the cumulative voucher sequence.

Example:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "hedera",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-12T12:05:00Z"
  },
  "payload": {
    "action": "open",
    "channelId":
      "0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c"
      "0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "txHash":
      "0x1a2b3c4d5e6f7890abcdef12345678"
      "90abcdef1234567890abcdef12345678",
    "cumulativeAmount": "2500",
    "signature": "0xabcdef1234567890..."
  }
}
```

Note: `cumulativeAmount` here is `"2500"` (the cost of the first request at 25 base units per token for 100 tokens). Implementations **MAY** also send `"0"`.

The `challenge` object **MUST** echo the challenge parameters from the server's `WWW-Authenticate` header per [I-D.httpauth-payment].

8.3.2. TopUp Payload

The `topUp` action adds funds to an existing channel during a streaming session. The client broadcasts the `topUp()` transaction itself and presents the transaction hash.

Clients **MUST** include a `challenge` object in the Payment credential for `topUp` actions. To obtain a challenge for a top-up outside an active streaming response, clients **MAY** send a `HEAD` request to the protected resource; the server returns 402 with a `WWW-Authenticate` challenge (no body). Servers **MUST** reject `topUp` actions referencing an unknown or expired challenge `id` with problem type `challenge-not-found`.

Payload fields (in addition to `action`):

Field	Type	Required	Description
`channelId`	string	**REQUIRED**	Channel ID
`txHash`	string	**REQUIRED**	Transaction hash from topUp()
`additionalDeposit`	string	**REQUIRED**	Additional amount in base units

Table 19

Example:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "hedera",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-12T12:05:00Z"
  },
  "payload": {
    "action": "topUp",
    "channelId":
      "0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c"
      "0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "txHash":
      "0x2b3c4d5e6f7890abcdef1234567890ab"
      "cdef1234567890abcdef1234567890ab",
    "additionalDeposit": "5000000"
  }
}
```

Upon successful verification, the server updates the channel's available balance. The new deposit is immediately available for voucher authorization.

8.3.3. Voucher Payload

The `voucher` action submits an updated cumulative voucher during streaming.

Payload fields (in addition to `action`):

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

Table 20

Example:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "hedera",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-12T12:05:00Z"
  },
  "payload": {
    "action": "voucher",
    "channelId":
      "0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c"
      "0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "cumulativeAmount": "250000",
    "signature": "0xabcdef1234567890..."
  }
}
```

8.3.4. Close Payload

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

Payload fields (in addition to `action`):

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

Table 21

The server uses the voucher fields to call `close(channelId, cumulativeAmount, signature)` on-chain via Hashio JSON-RPC.

Example:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "hedera",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-12T12:05:00Z"
  },
  "payload": {
    "action": "close",
    "channelId":
      "0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c"
      "0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "cumulativeAmount": "500000",
    "signature": "0xabcdef1234567890..."
  }
}
```

9. Voucher Signing Format

Vouchers use typed structured data signing compatible with [EIP-712]. This section normatively defines the signing procedure; [EIP-712] is referenced for background only.

9.1. Wire Format

Voucher fields are placed directly in the credential `payload` object (alongside `action`) rather than in a nested structure:

Field	Type	Required	Description
`channelId`	string	**REQUIRED**	Channel ID (hex bytes32)
`cumulativeAmount`	string	**REQUIRED**	Cumulative amount (decimal)
`signature`	string	**REQUIRED**	EIP-712 signature (hex)

Table 22

The EIP-712 domain and type definitions are fixed by this specification. Implementations **MUST** reconstruct the full typed data structure using the domain parameters from the challenge (`chainId`, `escrowContract`) before signature verification.

9.2. Type Definitions

The `types` object **MUST** contain exactly:

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

Note: The `EIP712Domain` type is implicit per EIP-712 and **SHOULD NOT** be included in the `types` object.

9.3. Domain Separator

The `domain` object **MUST** contain:

Field	Type	Value
`name`	string	`"Hedera Stream Channel"`
`version`	string	`"1"`
`chainId`	number	Hedera chain ID (295 or 296)
`verifyingContract`	string	Escrow contract address

Table 23

9.4. Signing Procedure

To sign a voucher, implementations **MUST**:

1. Construct the domain separator hash:

```
domainSeparator = keccak256(
  abi.encode(
    keccak256(
      "EIP712Domain(string name,"
      "string version,"
      "uint256 chainId,"
      "address verifyingContract)"
    ),
    keccak256(bytes(name)),
    keccak256(bytes(version)),
    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 signature as 65-byte `r || s || v` where `v` is 27 or 28

9.5. Cumulative Semantics

Vouchers specify cumulative totals, not incremental deltas:

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

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

10. Verification Procedure

10.1. Open Verification

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

1. **Transaction verification**: Wait for the transaction receipt using `txHash`. Verify the transaction succeeded (receipt status = `success`).
2. **On-chain state verification**: Query the escrow contract's `getChannel(channelId)` to verify:
   - Channel exists (deposit > 0)
   - `channel.payee` matches server's address
   - `channel.token` matches `request.currency`
   - `channel.deposit - channel.settled >= amount`
   - Channel is not finalized

- `channel.closeRequestedAt == 0`

3. **Voucher verification**: If `cumulativeAmount` and `signature` are provided, verify the initial voucher:
   - Recover signer from EIP-712 signature
   - Verify canonical low-s values
   - Signer matches `channel.payer` or `channel.authorizedSigner`
   - `voucher.channelId` matches
   - `voucher.cumulativeAmount >= channel.settled`
4. **Initialize** server-side channel state

10.2. TopUp Verification

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

1. **Transaction verification**: Wait for the transaction receipt using `txHash`. Verify the transaction succeeded.
2. **On-chain state verification**: Query the escrow contract to verify:
   - `channel.deposit` increased
   - Channel is not finalized
3. **Update** server-side accounting: increase available balance by `additionalDeposit`.

10.3. Voucher Verification

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

1. Verify voucher signature using EIP-712 recovery
2. Verify canonical low-s values (see Section 13.8)
3. Recover signer and **MUST** verify it matches expected signer from on-chain state
4. Verify `channel.closeRequestedAt == 0`. Servers **MUST** reject vouchers on channels with a pending forced close.
5. Verify monotonicity:
   - `cumulativeAmount > highestVoucherAmount`
   - `(cumulativeAmount - highestVoucherAmount) >= minVoucherDelta`
6. Verify `cumulativeAmount <= channel.deposit`
7. Persist voucher to durable storage before providing service
8. Update `highestVoucherAmount = cumulativeAmount`

Servers **MUST** derive the expected signer from on-chain channel state by querying the escrow contract. The expected signer is `channel.authorizedSigner` if non-zero, otherwise `channel.payer`. Servers **MUST NOT** trust signer claims in HTTP payloads.

Servers **MUST** persist the highest voucher to durable storage before providing the corresponding service. Failure to do so may result in unrecoverable fund loss if the server crashes after service delivery.

10.4. Idempotency

Servers **MUST** treat voucher submissions idempotently:

- Resubmitting a voucher with the same `cumulativeAmount` as the highest accepted **MUST** return 200 OK with the current `highestAmount`
- Submitting a voucher with lower `cumulativeAmount` than highest accepted **MUST** return 200 OK with the current `highestAmount` (not an error)
- Clients **MAY** safely retry voucher submissions after network failures

10.5. Rejection and Error Responses

If verification fails, servers **MUST** return an appropriate HTTP status code with a Problem Details [RFC9457] response body:

Status	When
400 Bad Request	Malformed payload or missing fields
402 Payment Required	Invalid signature or signer mismatch
410 Gone	Channel finalized or not found

Table 24

Error responses use Problem Details format:

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

Problem type URIs:

Type URI	Description
`.../session/invalid-signature`	Voucher signature invalid

Type URI	Description
`.../session/signer-mismatch`	Signer not authorized
`.../session/amount-exceeds-deposit`	Exceeds deposit
`.../session/delta-too-small`	Below minVoucherDelta
`.../session/channel-not-found`	No such channel
`.../session/channel-finalized`	Channel closed
`.../session/challenge-not-found`	Challenge expired
`.../session/insufficient-balance`	Insufficient balance

Table 25

All problem type URIs above are prefixed with `https://paymentauth.org/problems`.

For errors on the Payment Auth protected resource, servers **MUST** return 402 with a fresh `WWW-Authenticate: Payment` challenge per [I-D.httpauth-payment].

11. Server-Side Accounting

Servers **MUST** maintain per-session accounting state to track authorized funds versus consumed service.

11.1. Accounting State

For each active session identified by `(challengeId, channelId)`, servers **MUST** maintain:

Field	Type	Description
`acceptedCumulative`	uint128	Highest valid voucher accepted
`spent`	uint128	Cumulative amount charged
`settledOnChain`	uint128	Last settled amount (informational)

Table 26

The `available` balance is computed as:

```
available = acceptedCumulative - spent
```

11.2. Per-Request Processing

For each request carrying a Payment credential with `intent="session"`, servers **MUST** follow this procedure:

1. **Voucher acceptance** (if provided in credential):
   - Verify signature and monotonicity per Section 10.3
   - If valid, persist the new `acceptedCumulative`
   - If invalid, return 402 with a fresh challenge
2. **Balance check**:
   - Compute `available = acceptedCumulative - spent`
   - Compute `cost` for this request
   - If `available < cost`: return 402 with Problem Details including `requiredTopUp = cost - available`
3. **Charge and deliver** (if `available >= cost`):
   - **MUST persist** `spent := spent + cost` BEFORE or atomically with delivering service
   - Deliver the response (or next chunk for streaming)
   - Return `Payment-Receipt` header
4. **Receipt generation**:
   - Include balance state in receipt

11.3. Crash Safety

To prevent fund loss from server crashes:

- Servers **MUST** persist `spent` increments BEFORE delivering corresponding service.
- Servers **MUST** persist `acceptedCumulative` BEFORE relying on the new balance for service authorization.
- Implementations **SHOULD** use transactional storage or write-ahead logging to ensure atomicity.

11.4. Request Idempotency

To prevent double-charging on retries:

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

Example idempotent request:

```
GET /api/chat HTTP/1.1
Host: api.example.com
Idempotency-Key: req_a1b2c3d4e5f6
Authorization: Payment eyJ...
```

11.5. Cost Calculation

The `cost` for a request depends on the pricing model declared in the challenge. Servers **MUST** support at least one of:

- **Fixed cost**: A predetermined amount per request
- **Usage-based fees**: Pricing proportional to resource consumption

For streaming responses (SSE, chunked), 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

11.6. Insufficient Balance During Streaming

When a streaming response exhausts `available` balance:

1. Server **MUST** stop delivering additional content
2. Server **MAY** hold the connection open awaiting a voucher top-up
3. Server **MAY** close the response; client retries with a higher voucher
4. If client submits a voucher update, server **SHOULD** resume delivery if the connection is still open

For SSE responses, servers **MUST** emit a `payment-need-voucher` event when balance is exhausted:

```
event: payment-need-voucher
data: {"channelId":"0x6d0f4fdf...",
  "requiredCumulative":"250025",
  "acceptedCumulative":"250000",
  "deposit":"500000"}
```

The `payment-need-voucher` event data **MUST** be a JSON object containing:

Field	Type	Required	Description
`acceptedCumulative`	string	**REQUIRED**	Current highest voucher
`channelId`	string	**REQUIRED**	Channel identifier

Field	Type	Required	Description
`deposit`	string	**REQUIRED**	Current on-chain deposit
`requiredCumulative`	string	**REQUIRED**	Minimum next voucher

Table 27

The `deposit` field allows the client to determine the correct recovery action. When `requiredCumulative` exceeds `deposit`, the client **MUST** submit `action="topUp"` before sending a new voucher. When `requiredCumulative` is within `deposit`, the client can submit `action="voucher"` directly.

After emitting `payment-need-voucher`, the server **MUST** pause delivery until a valid voucher is accepted. Servers **SHOULD** close the stream if no voucher is received within a reasonable timeout (e.g., 60 seconds).

Servers **SHOULD NOT** deliver service beyond the authorized balance under any circumstances. See Section 13.3 for rate limiting requirements.

12. Settlement Procedure

12.1. Settlement Timing

Servers **MAY** settle at any time using their own criteria:

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

Settlement frequency is an implementation detail left to servers.

The `close()` function settles any delta between the provided `cumulativeAmount` and `channel.settled`. If the server has already settled the highest voucher via `settle()`, calling `close()` with the same amount will only refund the payer the remaining deposit.

12.2. Cooperative Close

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

1. Server receives the signed close request
2. Server calls `close(channelId, cumulativeAmount, signature)` on-chain via Hashio JSON-RPC
3. Contract settles any delta and refunds remainder
4. Server returns receipt with transaction hash

Servers **SHOULD** close promptly when clients request -- the economic incentive is to claim earned funds immediately.

The server **MUST** set a gas limit of at least 1,500,000 for the `close()` call due to HTS precompile overhead.

12.3. Forced Close

If the server does not respond to close requests:

1. Client calls `requestClose(channelId)` on-chain
2. 15-minute grace period begins
3. Server can still `settle()` or `close()` during the grace period
4. After grace period, client calls `withdraw(channelId)`
5. Client receives all remaining (unsettled) funds

Clients **SHOULD** wait at least 16 minutes after `requestClose()` before calling `withdraw()` to account for block time variance.

12.4. Sequential Sessions

A single channel supports sequential sessions. Each session uses the same cumulative voucher counter. When a new session begins on a channel, the previous session's spending state is irrelevant -- the channel's `highestVoucherAmount` is the source of truth for the next voucher's minimum value.

12.5. 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 when topping up during a long-lived response.

For voucher-only updates (no response body needed), clients **MAY** use `HEAD` requests. Servers **SHOULD** support voucher credentials on `HEAD` requests for resources that require session payment.

12.6. Receipt Generation

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

For SSE responses, the final receipt **SHOULD** be delivered as an event:

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

The session intent extends the receipt with balance tracking:

Field	Type	Description
`method`	string	`"hedera"`
`intent`	string	`"session"`
`status`	string	`"success"`
`timestamp`	string	[RFC3339] response time
`challengeId`	string	Challenge identifier
`channelId`	string	Channel identifier
`acceptedCumulative`	string	Highest voucher accepted
`spent`	string	Total charged so far
`reference`	string	Transaction or channel ref
`units`	number	**OPTIONAL**: Units consumed
`txHash`	string	**OPTIONAL**: Transaction hash

Table 28

The `reference` field satisfies the core MPP receipt `reference` requirement. It is set to `txHash` when a transaction was broadcast (open, close), otherwise set to `channelId` (voucher).

The `txHash` field is **OPTIONAL** because not every response involves an on-chain settlement -- voucher updates are off-chain.

Example receipt (per-request with metering):

```
{
  "method": "hedera",
  "intent": "session",
  "status": "success",
  "timestamp": "2026-04-12T12:08:30Z",
  "challengeId": "c_8d0e3b5a9f2c1d4e",
  "channelId":
    "0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c"
    "0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
  "reference":
    "0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c"
    "0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
  "acceptedCumulative": "250000",
  "spent": "237500",
  "units": 500
}
```

Example receipt (on close with settlement):

```
{
  "method": "hedera",
  "intent": "session",
  "status": "success",
  "timestamp": "2026-04-12T12:10:00Z",
  "challengeId": "c_8d0e3b5a9f2c1d4e",
  "channelId":
    "0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c"
    "0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
  "reference":
    "0x1a2b3c4d5e6f7890abcdef12345678"
    "90abcdef1234567890abcdef12345678",
  "acceptedCumulative": "250000",
  "spent": "250000",
  "txHash":
    "0x1a2b3c4d5e6f7890abcdef12345678"
    "90abcdef1234567890abcdef12345678"
}
```

13. Security Considerations

13.1. 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)

The escrow contract enforces:

- `cumulativeAmount > channel.settled` (monotonicity)
- `cumulativeAmount <= channel.deposit` (cap)

13.2. No Voucher Expiry

Vouchers have no `validUntil` field. This simplifies the protocol:

- Channels have no expiry -- 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 inactive for extended periods (e.g., 30+ days).

13.3. Denial of Service

To mitigate voucher flooding, servers **MUST** implement rate limiting:

- Servers **SHOULD** limit voucher submissions to 10 per second per session
- Servers **MAY** implement additional IP-based rate limiting
- Servers **MUST** enforce `minVoucherDelta` when present
- Servers **SHOULD** skip expensive signature verification for vouchers that do not advance state (return 200 OK with current `highestAmount` per Section 10.4)

Servers **SHOULD** perform format validation before expensive ECDSA signature recovery.

To mitigate channel griefing via dust deposits:

- Servers **SHOULD** enforce a minimum deposit (e.g., 1 USDC equivalent)
- Servers **MAY** reject channels below this threshold

13.4. Front-Running Protection

Cumulative voucher semantics prevent front-running attacks. If a client submits a higher voucher while a server's `settle()` transaction is pending, the settlement will still succeed -- it merely leaves additional unsettled funds.

13.5. Cross-Contract Replay Prevention

The EIP-712 domain includes `verifyingContract`, binding vouchers to a specific escrow contract address.

13.6. Escrow Guarantees

The escrow contract provides:

- **Payer protection**: Funds only withdrawn with valid voucher signature
- **Payee protection**: Deposited funds guaranteed
- **Forced close**: 15-minute grace period protects both parties

13.7. Authorized Signer

The `authorizedSigner` field allows delegation of signing authority to a hot wallet while the main wallet only deposits funds.

Security considerations for delegated signing:

- Clients using `authorizedSigner` delegation **SHOULD** limit channel deposits to acceptable loss amounts
- Clients **SHOULD** rotate authorized signers periodically
- Clients **SHOULD NOT** reuse signers across multiple high-value channels
- If the authorized signer key is compromised, an attacker can drain the entire channel deposit

13.8. Signature Malleability

ECDSA signatures are malleable: for any valid signature `(r, s)`, the signature `(r, -s mod n)` is also valid. To prevent signature substitution attacks, implementations **MUST** enforce canonical signatures:

- Signatures **MUST** use "low-s" values with `s <= secp256k1_order / 2`
- The secp256k1 half-order is: `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E73 57A4501DDFE92F46681B20A0`
- Servers **MUST** reject signatures with `s` values exceeding this threshold

Accepted signature formats:

- 65-byte `(r, s, v)` format where `v` is 27 or 28
- 64-byte EIP-2098 compact format

The HederaStreamChannel.sol contract uses Solady's `SignatureCheckerLib` which enforces these requirements.

13.9. Voucher Context and User Experience

The voucher message contains only `channelId` and `cumulativeAmount`. Wallet implementations are encouraged to:

- Decode `channelId` components when the derivation formula is known
- Display the payee address and token in human-readable form
- Show cumulative vs. incremental amounts clearly

13.10. Session Attribution

Vouchers are bound to channels but not to specific HTTP sessions or API requests. The `challengeId` provides correlation across requests. Servers **MUST** implement challenge-to-voucher mapping for:

- Dispute resolution
- Usage accounting
- Audit trails

13.11. Cross-Session Replay Prevention

Vouchers use cumulative amount semantics: each voucher authorizes a total payment up to `cumulativeAmount`, and the on-chain contract enforces strict monotonicity (`cumulativeAmount > channel.settled`). A voucher can only ever advance the channel state forward.

A separate `sessionHash` binding is unnecessary:

- **Cross-session replay is harmless**: If a voucher from session A is presented in session B, it can only authorize funds up to the amount already committed.
- **Cross-resource replay**: Vouchers authorize cumulative payment on a channel, not access to specific resources. Resource authorization is handled at the application layer via `challengeId`.

13.12. Chain Finality

Hedera achieves asynchronous Byzantine Fault Tolerant (aBFT) consensus with deterministic finality in approximately 3-5 seconds [HEDERA-DOCS]. Once a transaction reaches consensus, it cannot be reversed.

For high-value channels, servers **SHOULD**:

1. Re-verify channel state periodically during long-lived sessions
2. Monitor for `ChannelClosed` or `CloseRequested` events
3. Cease service delivery if the channel becomes invalid

13.13. HTS Token Association

Before an escrow contract can receive HTS tokens, it **MUST** be associated with the token via the `associateSelf()` function (see Section 5.3.7). This is a one-time operation per token. If the escrow contract is not associated with the payment token, `open()` will fail with a transfer error.

Servers deploying escrow contracts **MUST** ensure the contract is associated with all supported payment tokens before advertising session challenges.

13.14. Gas Limit Considerations

Hedera's EVM layer routes HTS token operations through the HTS precompile at address `0x167`. This precompile has higher gas requirements than standard ERC-20 operations. Implementations **MUST** set appropriate gas limits (see Section 7) to avoid transaction failures.

The default gas estimates from Hashio JSON-RPC (`eth_estimateGas`) may underestimate gas for HTS precompile calls. Implementations **SHOULD** use hardcoded minimum gas limits for escrow operations.

13.15. Grace Period Rationale

The 15-minute forced close grace period balances competing concerns:

- **Payer protection**: Ensures timely fund recovery
- **Payee protection**: Provides time to detect close requests and submit final settlements
- **Block time variance**: Allows margin for timestamp variations

14. IANA Considerations

14.1. Payment Intent Registration

This document registers the following payment intent in the "HTTP Payment Intents" registry established by [I-D.httpauth-payment]:

Intent	Methods	Description	Reference
`session`	`hedera`	Streaming payment channel	This document

Table 29

Contact: Tom Rowbotham (tom@xeno.money)

14.2. Problem Type Registration

This document registers the following problem types in the "HTTP Problem Types" registry established by [RFC9457]:

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

Table 30

All type URIs above are prefixed with `https://paymentauth.org/problems`.

Each problem type is defined in Section 10.5.

15. References

15.1. 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>.

[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>.

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

15.2. Informative References

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

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

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

[HEDERA-DOCS] Hedera, "Hedera Documentation", 2026, <https://docs.hedera.com>.

[HIP-218] Hedera, "HIP-218: Smart Contract Verification", 2022, <https://hips.hedera.com/hip/hip-218>.

[CIRCLE-USDC-HEDERA] Circle, "Circle USDC on Hedera", 2026, <https://www.circle.com/multi-chain-usdc/hedera>.

Appendix A. Example

Note: In examples throughout this appendix, hex values shown with `...` (e.g., `"0x6d0f4fdf..."`) are abbreviated. Actual values **MUST** be full-length as specified in Section 4.

A.1. Challenge

```
HTTP/1.1 402 Payment Required
WWW-Authenticate: Payment
  id="kM9xPqWvT2nJrHsY4aDfEb",
  realm="api.llm-service.com",
  method="hedera",
  intent="session",
  expires="2026-04-12T12:05:00Z",
  request="<base64url-encoded JSON below>"
```

The `request` decodes to:

```
{
  "amount": "25",
  "unitType": "llm_token",
  "suggestedDeposit": "10000000",
  "currency":
    "0x0000000000000000000000000000000000006f89a",
  "recipient":
    "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
  "methodDetails": {
    "escrowContract":
      "0x8Aaf6690C2a6397d595F97E224fC19759De6fdaE",
    "chainId": 295
  }
}
```

Note: Challenge expiry is in the header `expires` auth-param, not in the request JSON. The client generates a random salt locally for new channels.

This requests 0.000025 USDC per LLM token, with a suggested deposit of 10.00 USDC (10000000 base units).

A.2. Open Credential

The client first broadcasts `approve()` and `open()` to Hedera EVM via Hashio JSON-RPC, then retries the **same resource URI** with the open credential:

```
GET /api/chat HTTP/1.1
Host: api.llm-service.com
Authorization: Payment <base64url credential>
```

The credential payload for an open action:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "hedera",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-12T12:05:00Z"
  },
  "payload": {
    "action": "open",
    "channelId": "0x6d0f4fdf...",
    "txHash": "0x1a2b3c4d...",
    "cumulativeAmount": "2500",
    "signature": "0xabcdef1234567890..."
  }
}
```

A.3. Voucher Top-Up (Same Resource URI)

During streaming, clients submit updated vouchers to the **same resource URI**. `HEAD` is recommended for pure top-ups when no response body is needed:

```
HEAD /api/chat HTTP/1.1
Host: api.llm-service.com
Authorization: Payment <base64url credential
  with action="voucher">
```

Or with a regular request:

```
GET /api/chat HTTP/1.1
Host: api.llm-service.com
Authorization: Payment <base64url credential
  with action="voucher">
```

The credential payload for a voucher update:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "hedera",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-12T12:05:00Z"
  },
  "payload": {
    "action": "voucher",
    "channelId": "0x6d0f4fdf...",
    "cumulativeAmount": "250000",
    "signature": "0x1234567890abcdef..."
  }
}
```

A.4. Close Request (Same Resource URI)

```
GET /api/chat HTTP/1.1
Host: api.llm-service.com
Authorization: Payment <base64url credential
  with action="close">
```

The credential payload for a close request:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "hedera",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-04-12T12:05:00Z"
  },
  "payload": {
    "action": "close",
    "channelId": "0x6d0f4fdf...",
    "cumulativeAmount": "500000",
    "signature": "0xabcdef1234567890..."
  }
}
```

The voucher fields contain the final cumulative amount for on-chain settlement.

Appendix B. Reference Implementation

This appendix provides reference implementation details. These are informative and not normative.

B.1. Solidity Interface

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

interface IHederaStreamChannel {
    struct Channel {
        bool finalized;
        uint64 closeRequestedAt;
        address payer;
        address payee;
        address token;
        address authorizedSigner;
        uint128 deposit;
        uint128 settled;
    }

    function CLOSE_GRACE_PERIOD()
        external view returns (uint64);
    function VOUCHER_TYPEHASH()
        external view returns (bytes32);

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

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

    function topUp(
        bytes32 channelId,
        uint256 additionalDeposit
    ) external;

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

    function requestClose(
        bytes32 channelId
    ) external;

    function withdraw(
        bytes32 channelId
    ) external;

    function getChannel(
```

```
        bytes32 channelId
    ) external view returns (Channel memory);

    function getChannelsBatch(
        bytes32[] calldata channelIds
    ) external view returns (Channel[] memory);

    function computeChannelId(
        address payer,
        address payee,
        address token,
        bytes32 salt,
        address authorizedSigner
    ) external view returns (bytes32);

    function getVoucherDigest(
        bytes32 channelId,
        uint128 cumulativeAmount
    ) external view returns (bytes32);

    function domainSeparator()
        external view returns (bytes32);

    function associateSelf(
        address token
    ) external returns (int256 responseCode);

    event ChannelOpened(
        bytes32 indexed channelId,
        address indexed payer,
        address indexed payee,
        address token,
        address authorizedSigner,
        bytes32 salt,
        uint256 deposit
    );

    event Settled(
        bytes32 indexed channelId,
        address indexed payer,
        address indexed payee,
        uint256 cumulativeAmount,
        uint256 deltaPaid,
        uint256 newSettled
    );

    event CloseRequested(
        bytes32 indexed channelId,
        address indexed payer,
        address indexed payee,
        uint256 closeGraceEnd
    );

    event CloseRequestCancelled(
        bytes32 indexed channelId,
        address indexed payer,
        address indexed payee
    );
```

```
    event TopUp(
        bytes32 indexed channelId,
        address indexed payer,
        address indexed payee,
        uint256 additionalDeposit,
        uint256 newDeposit
    );

    event ChannelClosed(
        bytes32 indexed channelId,
        address indexed payer,
        address indexed payee,
        uint256 settledToPayee,
        uint256 refundedToPayer
    );

    event ChannelExpired(
        bytes32 indexed channelId,
        address indexed payer,
        address indexed payee
    );

    error ChannelAlreadyExists();
    error ChannelNotFound();
    error ChannelFinalized();
    error InvalidSignature();
    error InvalidToken();
    error InvalidPayee();
    error AmountExceedsDeposit();
    error AmountNotIncreasing();
    error DepositOverflow();
    error ZeroDeposit();
    error NotPayer();
    error NotPayee();
    error TransferFailed();
    error CloseNotReady();
}
```

B.2. Deployed Contracts

Network	Chain ID	Contract Address
Hedera Testnet	296	`0x8Aaf6690C2a6397d595F97E224fC19759De6fdaE`
Hedera Mainnet	295	`0x8Aaf6690C2a6397d595F97E224fC19759De6fdaE`

Table 31

Both deployments are fully verified on Sourcify.

B.3. Supported Tokens

Token	Network	HTS Token ID	EVM Address
USDC	Testnet	0.0.5449	`0x00...1549`
USDC	Mainnet	0.0.456858	`0x00...06f89a`

Table 32

HTS tokens are exposed as ERC-20 via [HIP-218].

B.4. Contract Source

The reference implementation is available at: `contracts/src/HederaStreamChannel.sol` in the mppx-hedera repository.

B.5. TypeScript Client Library

The `mppx-hedera` npm package provides client and server implementations:

- `mppx-hedera/client` -- `hederaSession()` client
- `mppx-hedera/server` -- `hedera.session()` server
- `mppx-hedera/server/sse` -- SSE transport for metered streaming

Appendix C. Schema Definitions (JSON Schema)

C.1. Session Request Schema

```
{
  "$schema":
    "https://json-schema.org/draft/2020-12/schema",
  "$id":
    "https://paymentauth.org/schemas/"
    "hedera-session-request.json",
  "title": "Hedera Session Request",
  "type": "object",
  "required": [
    "amount", "currency",
    "recipient", "methodDetails"
  ],
  "properties": {
    "amount": {
      "type": "string",
      "pattern": "^[0-9]+$"
    },
    "unitType": {
      "type": "string"
    },
    "suggestedDeposit": {
```

```
        "type": "string",
        "pattern": "^[0-9]+$"
      },
      "currency": {
        "type": "string",
        "pattern": "^0x[0-9a-fA-F]{40}$"
      },
      "recipient": {
        "type": "string",
        "pattern": "^0x[0-9a-fA-F]{40}$"
      },
      "methodDetails": {
        "$ref": "#/$defs/methodDetails"
      }
    },
    "$defs": {
      "methodDetails": {
        "type": "object",
        "required": ["escrowContract"],
        "properties": {
          "escrowContract": {
            "type": "string",
            "pattern": "^0x[0-9a-fA-F]{40}$"
          },
          "channelId": {
            "type": "string",
            "pattern": "^0x[0-9a-fA-F]{64}$"
          },
          "minVoucherDelta": {
            "type": "string",
            "pattern": "^[0-9]+$"
          },
          "chainId": {
            "type": "integer",
            "enum": [295, 296]
          }
        }
      }
    }
}
```

C.2. Session Payload Schema

```
{
  "$schema":
    "https://json-schema.org/draft/2020-12/schema",
  "$id":
    "https://paymentauth.org/schemas/"
    "hedera-session-payload.json",
  "title": "Hedera Session Payload",
  "type": "object",
  "required": ["action"],
  "properties": {
    "action": {
      "enum": [
        "open", "topUp", "voucher", "close"
      ]
    },
    "txHash": {
      "type": "string",
      "pattern": "^0x[0-9a-fA-F]{64}$"
    },
    "channelId": {
      "type": "string",
      "pattern": "^0x[0-9a-fA-F]{64}$"
    },
    "cumulativeAmount": {
      "type": "string",
      "pattern": "^[0-9]+$"
    },
    "signature": {
      "type": "string",
      "pattern": "^0x[0-9a-fA-F]{128,130}$"
    },
    "additionalDeposit": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description":
        "Additional deposit amount in base units "
        "(topUp action only)"
    }
  }
}
```

C.3. Session Receipt Schema

Servers **MUST** include `Payment-Receipt` only on successful processing of a session action (2xx).

```
{
  "$schema":
    "https://json-schema.org/draft/2020-12/schema",
  "$id":
    "https://paymentauth.org/schemas/"
    "hedera-session-receipt.json",
```

```
  "title": "Hedera Session Receipt",
  "type": "object",
  "required": [
    "method", "intent", "status",
    "timestamp", "challengeId",
    "channelId", "reference",
    "acceptedCumulative", "spent"
  ],
  "properties": {
    "method": { "const": "hedera" },
    "intent": { "const": "session" },
    "status": { "const": "success" },
    "timestamp": {
      "type": "string",
      "format": "date-time"
    },
    "challengeId": { "type": "string" },
    "channelId": {
      "type": "string",
      "pattern": "^0x[0-9a-fA-F]{64}$"
    },
    "reference": {
      "type": "string",
      "description":
        "txHash when a tx was broadcast "
        "(open, close); channelId otherwise"
    },
    "acceptedCumulative": {
      "type": "string",
      "pattern": "^[0-9]+$"
    },
    "spent": {
      "type": "string",
      "pattern": "^[0-9]+$"
    },
    "units": {
      "type": "integer"
    },
    "txHash": {
      "type": "string",
      "pattern": "^0x[0-9a-fA-F]{64}$"
    }
  }
}
```

Appendix D. Acknowledgements

The author thanks the Tempo team for the MPP session payment channel design and the mppx ecosystem architecture that this specification builds upon. HederaStreamChannel.sol is a port of Tempo's TempoStreamChannel.sol adapted for Hedera's EVM layer and HTS token ecosystem.

Authors' Addresses

Tom Rowbotham
Email: tom@xeno.money

Lindsay Walker
Hedera / Hashgraph
Email: lindsay.w@swirldslabs.com