Workgroup:	Network Working Group
Internet-Draft:	draft-tempo-session-00
Published:	17 April 2026
Intended Status:	Informational
Expires:	19 October 2026

Authors:

L. Horne, *Tempo Labs*
G. Konstantopoulos, *Tempo Labs*
D. Robinson, *Tempo Labs*
B. Ryan, *Tempo Labs*
J. Moxey, *Tempo Labs*

Tempo Session Intent for HTTP Payment Authentication

Abstract

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

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 19 October 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

This document may not be modified, and derivative works of it may not be created, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction 5
  1.1. Use Case: LLM Token Streaming 5
  1.2. Session Flow 6
  1.3. Concurrency Model 8
2. Requirements Language 8
3. Terminology 8
4. Encoding Conventions 8
  4.1. Hexadecimal Values 9
  4.2. Numeric Values 9
  4.3. Timestamp Format 10
5. Channel Escrow Contract 10
  5.1. Channel State 10
  5.2. Channel Lifecycle 11
  5.3. Contract Functions 11
    5.3.1. open 11
    5.3.2. settle 12
    5.3.3. topUp 12
    5.3.4. close 13
    5.3.5. requestClose 13
    5.3.6. withdraw 14
  5.4. Access Control 14
  5.5. Signature Verification 14
6. Request Schema 15
  6.1. Fields 15
  6.2. Method Details 16

- 7. Fee Payment 17
  - 7.1. Server-Paid Fees 17
  - 7.2. Client-Paid Fees 18
  - 7.3. Server-Initiated Operations 18
  - 7.4. Server Requirements 18
  - 7.5. Client Requirements 18
- 8. Credential Schema 19
  - 8.1. Credential Structure 19
  - 8.2. Credential Lifecycle 19
  - 8.3. Payload Actions 19
    - 8.3.1. Open Payload 20
    - 8.3.2. TopUp Payload 21
    - 8.3.3. Voucher Payload 22
    - 8.3.4. Close Payload 23
- 9. Voucher Signing Format 24
  - 9.1. Wire Format 24
  - 9.2. Type Definitions 24
  - 9.3. Domain Separator 25
  - 9.4. Signing Procedure 25
  - 9.5. Cumulative Semantics 26
- 10. Verification Procedure 26
  - 10.1. Open Verification 26
  - 10.2. TopUp Verification 27
  - 10.3. Voucher Verification 27
  - 10.4. Idempotency 28
  - 10.5. Rejection and Error Responses 28
- 11. Server-Side Accounting 29
  - 11.1. Accounting State 29
  - 11.2. Per-Request Processing 30
  - 11.3. Crash Safety 30

11.4. Request Idempotency 31
11.5. Cost Calculation 31
11.6. Insufficient Balance During Streaming 31
12. Settlement Procedure 33
12.1. Settlement Timing 33
12.2. Cooperative Close 33
12.3. Forced Close 33
12.4. Sequential Sessions 34
12.5. Voucher Submission Transport 34
12.6. Receipt Generation 34
13. Security Considerations 36
13.1. Replay Prevention 36
13.2. No Voucher Expiry 36
13.3. Denial of Service 36
13.4. Front-Running Protection 37
13.5. Cross-Contract Replay Prevention 37
13.6. Escrow Guarantees 37
13.7. Authorized Signer 37
13.8. Signature Malleability 38
13.9. Voucher Context and User Experience 38
13.10. Session Attribution 38
13.11. Cross-Session Replay Prevention 39
13.12. Chain Reorganization 39
13.13. Grace Period Rationale 39
14. IANA Considerations 40
14.1. Payment Intent Registration 40
14.2. Problem Type Registration 40
15. References 41
15.1. Normative References 41
15.2. Informative References 41

Appendix A. Example 42
A.1. Challenge 42
A.2. Open Credential 42
A.3. Voucher Top-Up (Same Resource URI) 43
A.4. Close Request (Same Resource URI) 44
Appendix B. Reference Implementation 44
B.1. Solidity Interface 45
B.2. Deployed Contracts 47
B.3. Contract Source 47
Appendix C. Schema Definitions (JSON Schema) 47
C.1. Session Request Schema 47
C.2. Session Payload Schema 49
C.3. Session Receipt Schema 49
Appendix D. Acknowledgements 50
Authors' Addresses 50

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.

1.1. Use Case: LLM Token Streaming

Consider an LLM inference API that charges per output token:

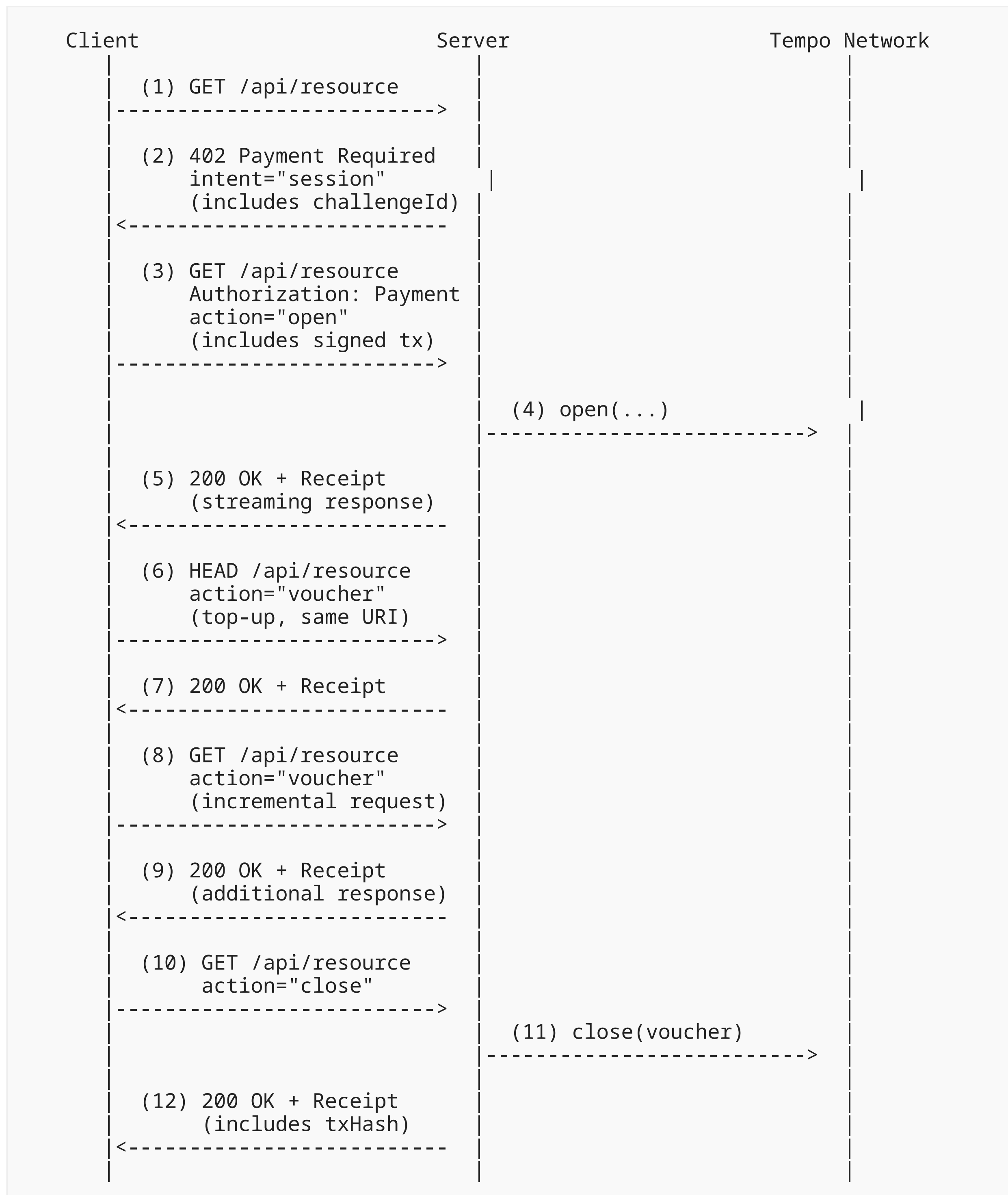
1. Client requests a streaming completion (SSE response)
2. Server returns 402 with a `session` challenge

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.

1.2. Session Flow

The following diagram illustrates the Tempo session flow:



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 [TIP-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.

Base Units
The smallest indivisible unit of a TIP-20 token. TIP-20 tokens use 6 decimal places; one million base units equals 1.00 tokens.

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)	`0x742d35cc6634c0532925a3b844bc9e7595f8fe00`
bytes32	66 chars (0x + 64 hex)	`0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f`
signature	130-132 chars (0x + 128-130 hex)	65-byte (r‖s‖v) or 64-byte EIP-2098 compact

Table 1

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

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 (Unix seconds)	`"1736165100"`
`chainId`	JSON number	`42431`

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: `2025-01-06T12:05:00Z`. 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.

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	[TIP-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

Table 3

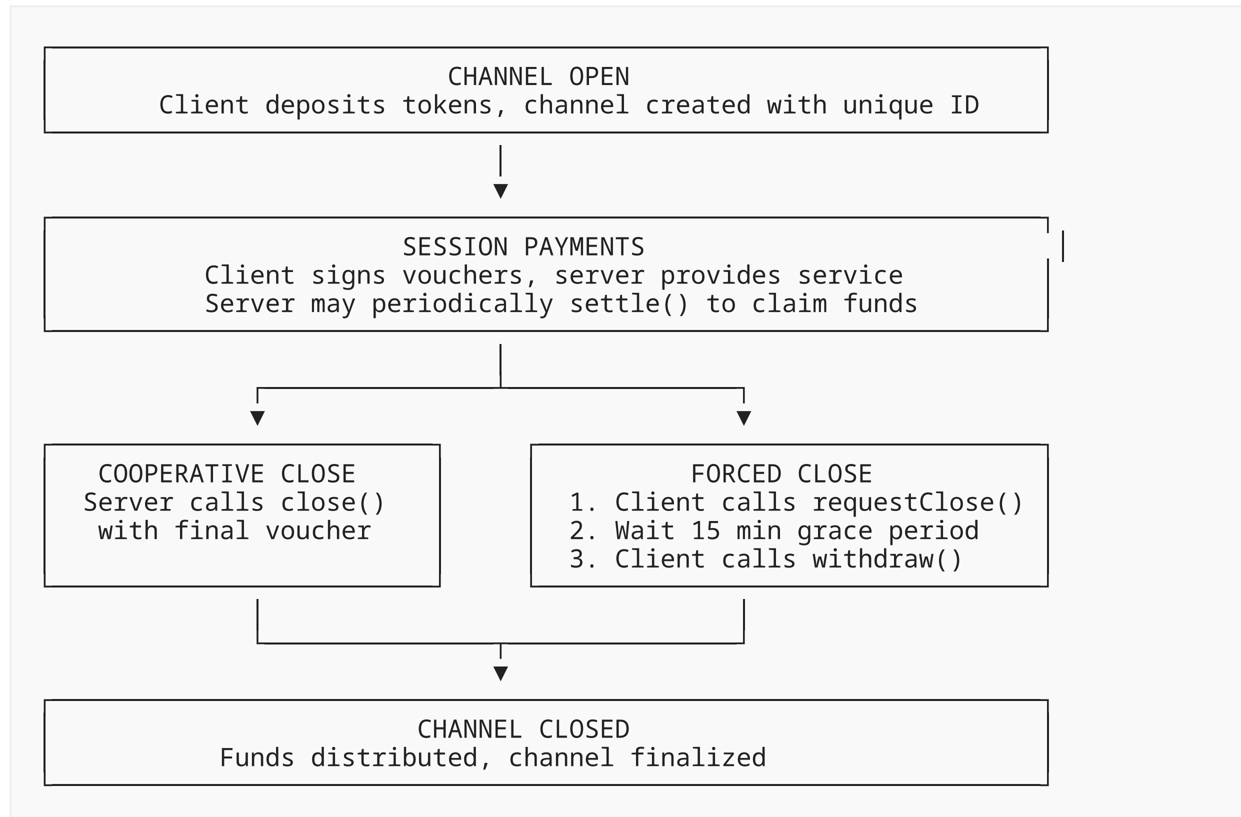
The `channelId` **MUST** be computed deterministically using the escrow contract's `computeChannelId()` function:

```
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 a reference implementation; alternative implementations **MAY** use different parameter types (e.g., `uint256` instead of `uint128`) as long as the semantics are preserved.

5.3.1. open

Opens a new channel with escrowed funds.

Parameter	Type	Description
`payee`	address	Server's address authorized to withdraw funds
`token`	address	[TIP-20] token contract address

Parameter	Type	Description
`deposit`	uint128	Amount to deposit in base units (6 decimals)
`salt`	bytes32	Random value for deterministic channelId computation
`authorizedSigner`	address	Delegated signer; use `0x0` to default to payer

Table 4

Returns the computed `channelId`.

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

5.3.2. settle

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

Parameter	Type	Description
`channelId`	bytes32	Unique channel identifier
`cumulativeAmount`	uint128	Cumulative total authorized (not delta)
`signature`	bytes	EIP-712 signature from authorized signer

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`	uint128	Additional amount to deposit in base units

Table 6

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

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 for settlement
`signature`	bytes	EIP-712 signature from authorized signer

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 for which to request closure

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;
```

5.4. Access Control

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

Function	Caller	Description
`open`	Anyone	Creates channel; caller becomes payer
`settle`	Payee only	Withdraws funds using voucher
`topUp`	Payer only	Adds funds to existing channel
`close`	Payee only	Closes channel with final voucher
`requestClose`	Payer only	Initiates forced close
`withdraw`	Payer only	Withdraws after grace period

Table 10

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 `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0`. 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 for the channel:
   - 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 (see note below)
`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**	[TIP-20] token address (e.g., `"0x20c0..."`)
`recipient`	string	**REQUIRED**	Payee address (server's withdrawal address)—equivalent to the on-chain payee

Table 11

For the `session` intent, `amount` specifies the price per unit of service in base units (6 decimals), 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**	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 (server policy hint)
`methodDetails.feePayer`	boolean	**OPTIONAL**	If `true`, server pays transaction fees (default: `false`)
`methodDetails.chainId`	number	**OPTIONAL**	Tempo chain ID (default: 42431)

Table 12

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": "0x20c0000000000000000000000000000000000000",
  "recipient": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
  "methodDetails": {
    "escrowContract": "0x1234567890abcdef1234567890abcdef12345678",
    "chainId": 42431
  }
}
```

This requests a price of 0.000025 tokens per LLM token, with a suggested deposit of 10.00 tokens. The client generates a random salt locally.

Example (existing channel):

```
{
  "amount": "25",
  "unitType": "llm_token",
  "currency": "0x20c0000000000000000000000000000000000000",
  "recipient": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
  "methodDetails": {
    "escrowContract": "0x1234567890abcdef1234567890abcdef12345678",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "chainId": 42431
  }
}
```

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

7. Fee Payment

When a challenge includes `methodDetails.feePayer: true`, the server commits to paying transaction fees on behalf of the client. In the `session` intent, `feePayer` affects only the client-originated channel funding transactions (`open` and `topUp`).

7.1. Server-Paid Fees

When `feePayer: true` for `open` or `topUp`:

1. **Client signs with placeholder**: The client signs the Tempo Transaction [TEMPO-TX-SPEC] with `fee_payer_signature` set to a placeholder value (`0x00`) and `fee_token` left empty. The client uses signature domain `0x76`.
2. **Server receives credential**: The server extracts the client-signed transaction from the credential payload.

3. **Server adds fee payment signature**: The server selects a `fee_token` (any USD-denominated TIP-20 stablecoin) and signs the transaction using signature domain `0x78`. This signature commits to the transaction including the `fee_token` and client's address.
4. **Server broadcasts**: The final transaction contains both signatures:
   - Client's signature (authorizing the channel operation)
   - Server's `fee_payer_signature` (committing to pay fees)

7.2. Client-Paid Fees

When `feePayer: false` or omitted, the client **MUST** set `fee_token` to a valid USD TIP-20 token address and include valid fee payment fields so the transaction is executable without server fee sponsorship. The server broadcasts the transaction as-is.

7.3. Server-Initiated Operations

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

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

The `feePayer` field applies only to `open` and `topUp` operations where the client provides a signed transaction.

7.4. Server Requirements

When acting as fee payer for `open` or `topUp`:

- Servers **MUST** maintain sufficient balance of a USD TIP-20 token to pay transaction fees
- Servers **MAY** use any USD-denominated TIP-20 token with sufficient AMM liquidity as the fee token
- Servers **MUST** validate the transaction matches challenge and channel parameters before adding fee payer signature
- Servers **MUST** reject credentials with unknown `action` values

7.5. Client Requirements

- When `feePayer: true`: Clients **MUST** sign with `fee_payer_signature` set to `0x00` and `fee_token` empty or `0x80` (RLP null)
- When `feePayer: false` or omitted: Clients **MUST** set `fee_token` to a valid USD TIP-20 token and have sufficient balance to pay fees

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 from the server's WWW-Authenticate header
`payload`	object	**REQUIRED**	Session-specific payload object

Table 13

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 and presents the `open` action to begin the session. The server verifies the on-chain deposit and validates the initial zero-amount 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 `"open"`, `"topUp"`, `"voucher"`, `"close"`

Table 14

Action-specific fields are placed directly in the `payload` object alongside `action`. See each action's definition for required fields.

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

Table 15

8.3.1. Open Payload

The `open` action confirms an on-chain channel opening and begins the streaming session. The client provides a signed transaction for the server to broadcast.

Payload fields (in addition to `action`):

Field	Type	Required	Description
`type`	string	**REQUIRED**	`"transaction"`
`channelId`	string	**REQUIRED**	Channel identifier (hex-encoded bytes32)
`transaction`	string	**REQUIRED**	Signed transaction bytes
`authorizedSigner`	string	**OPTIONAL**	Address delegated to sign vouchers
`cumulativeAmount`	string	**REQUIRED**	Initial cumulative amount (typically `"0"`)
`signature`	string	**REQUIRED**	EIP-712 voucher signature for the initial amount

Table 16

The `transaction` field contains the complete signed Tempo Transaction (type 0x76) [TEMPO-TX-SPEC] serialized as RLP and hex-encoded. The server broadcasts the transaction, optionally adding a fee payer signature if `feePayer: true` was specified in the challenge (see Section 7).

The server recovers the `payer` address from the signed transaction and uses it to compute the `channelId` deterministically (see Section 5.1). The `authorizedSigner` is inferred from the calldata inside `transaction` and verified when the transaction is signed.

The initial voucher (`cumulativeAmount` and `signature`) proves the client controls the signing key and establishes the voucher chain.

Example:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "tempo",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2025-01-06T12:05:00Z"
  },
  "payload": {
    "action": "open",
    "type": "transaction",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "transaction": "0x76f901...signed transaction bytes...",
    "cumulativeAmount": "0",
    "signature": "0xabcdef1234567890..."
  }
}
```


Note: The `transaction` field contains RLP-encoded transaction bytes. When provided, the `signature` field is the EIP-712 voucher signature (65 bytes r‖s‖v or 64 bytes EIP-2098 compact).

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. Like `open`, the client provides a signed transaction for the server to broadcast.

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
`type`	string	**REQUIRED**	`"transaction"`
`channelId`	string	**REQUIRED**	Channel ID
`transaction`	string	**REQUIRED**	Signed transaction bytes
`additionalDeposit`	string	**REQUIRED**	Additional amount to deposit in base units

Table 17

Example:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "tempo",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2025-01-06T12:05:00Z"
  },
  "payload": {
    "action": "topUp",
    "type": "transaction",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "transaction": "0x76f901...signed topUp transaction bytes...",
    "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 18

Example:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "tempo",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2025-01-06T12:05:00Z"
  },
  "payload": {
    "action": "voucher",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "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 for settlement
`signature`	string	**REQUIRED**	EIP-712 voucher signature

Table 19

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

Example:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "tempo",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2025-01-06T12:05:00Z"
  },
  "payload": {
    "action": "close",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "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 identifier (hex-encoded bytes32)
`cumulativeAmount`	string	**REQUIRED**	Cumulative amount authorized (decimal string)
`signature`	string	**REQUIRED**	EIP-712 signature (hex-encoded)

Table 20

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. The domain separator is computed from the `domain` object using the canonical type string `EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)`.

9.3. Domain Separator

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

Field	Type	Value
`name`	string	`"Tempo Stream Channel"`
`version`	string	`"1"`
`chainId`	number	Tempo chain ID (e.g., `42431`)
`verifyingContract`	string	Escrow contract address from challenge

Table 21

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` (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`

10. Verification Procedure

10.1. Open Verification

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

1. **Transaction verification**: Decode the signed transaction from `transaction`, verify it calls `open()` on the expected escrow contract with correct parameters. Recover the `payer` address from the transaction, infer `authorizedSigner` from the calldata, and compute `channelId` deterministically (see Section 5.1). If `feePayer: true`, add fee payer signature using domain `0x78` (see Section 7) and broadcast. Otherwise, broadcast as-is.
2. Query the escrow contract to verify channel state:
   - Channel exists with the computed `channelId`
   - `channel.payee` matches server's address
   - `channel.token` matches `request.currency`
   - `channel.deposit - channel.settled >= amount` (sufficient available balance)
   - Channel is not finalized
   - `channel.closeRequestedAt == 0` (no pending close request)

3. If `cumulativeAmount` and `signature` are provided, verify the initial voucher:
   - Recover signer from EIP-712 signature
   - Verify signature uses canonical low-s values (see Section 13.8)
   - Signer matches `channel.payer` or `channel.authorizedSigner`
   - `voucher.channelId` matches
   - `voucher.cumulativeAmount >= channel.settled` (at or above current settlement)
4. Initialize server-side channel state

10.2. TopUp Verification

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

1. **Transaction verification**: Decode the signed transaction from `transaction`, verify it calls `topUp()` on the expected escrow contract with the specified `additionalDeposit` amount. If `feePayer: true`, add fee payer signature using domain `0x78` (see Section 7) and broadcast. Otherwise, broadcast as-is.
2. Query the escrow contract to verify updated channel state:
   - `channel.deposit` increased by `additionalDeposit`
   - 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 signature uses 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` (no pending close request). Servers **MUST** reject vouchers on channels with a pending forced close to prevent service delivery that cannot be settled.
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 22

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
https://paymentauth.org/problems/session/invalid-signature	Voucher or close request signature invalid
https://paymentauth.org/problems/session/signer-mismatch	Signer is not authorized for this channel
https://paymentauth.org/problems/session/amount-exceeds-deposit	Voucher amount exceeds channel deposit
https://paymentauth.org/problems/session/delta-too-small	Amount increase below `minVoucherDelta`
https://paymentauth.org/problems/session/channel-not-found	No channel with this ID exists
https://paymentauth.org/problems/session/channel-finalized	Channel has been closed
https://paymentauth.org/problems/session/challenge-not-found	Challenge ID unknown or expired
https://paymentauth.org/problems/session/insufficient-balance	Insufficient authorized balance for request

Table 23

For errors on the Payment Auth protected resource (the initial request carrying `Authorization: Payment`), 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. This section defines the normative requirements for balance tracking, crash safety, and idempotency.

11.1. Accounting State

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

Field	Type	Description
`acceptedCumulative`	uint128	Highest valid voucher amount accepted (monotonically increasing)

Field	Type	Description
`spent`	uint128	Cumulative amount charged for delivered service (monotonically increasing)
`settledOnChain`	uint128	Last cumulative amount settled on-chain (informational)

Table 24

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 a voucher is provided in the credential):
    - Verify signature and monotonicity per Section 10.3
    - If valid, persist the new `acceptedCumulative` value to durable storage
    - If invalid, return 402 with a fresh challenge
2. **Balance check**:
    - Compute `available = acceptedCumulative - spent`
    - Compute `cost` for this request (see Section 11.5)
    - If `available < cost`: return 402 with Problem Details including `requiredTopUp = cost - available`
3. **Charge and deliver** (if `available >= cost`):
    - **MUST persist** `spent := spent + cost` to durable storage BEFORE or atomically with delivering the metered service
    - Deliver the response (or next chunk/token window for streaming)
    - Return `Payment-Receipt` header with current balance state
4. **Receipt generation**:
    - Include balance state in receipt (see Section 12.6)

11.3. Crash Safety

To prevent fund loss from server crashes:

- Servers **MUST** persist `spent` increments BEFORE delivering corresponding service. If the server crashes after persisting but before delivery, the client may retry and be charged again (see Idempotency below).

- 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 between state updates and service delivery.

11.4. Request Idempotency

To prevent double-charging on retries and network failures:

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

If idempotency is not implemented, servers **MUST** document this limitation and warn clients that retries may incur additional charges.

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 (e.g., tokens generated, bytes transferred, compute time)

For metered resources, servers compute cost during or after service delivery. 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 (client must top-up)

11.6. Insufficient Balance During Streaming

When a streaming response exhausts `available` balance:

1. Server **MUST** stop delivering additional metered content
2. Server **MAY** hold the connection open awaiting a voucher top-up

3. Server **MAY** close the response; client then retries with higher voucher
4. If client submits a voucher update (request to same URI or any endpoint protected by the same payment handler), server **SHOULD** resume delivery on the original connection if still open

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

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

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

Field	Type	Required	Description
`acceptedCumulative`	string	**REQUIRED**	Current highest accepted voucher amount (base units)
`channelId`	string	**REQUIRED**	Channel identifier (hex-encoded bytes32)
`deposit`	string	**REQUIRED**	Current on-chain deposit in the escrow contract (base units)
`requiredCumulative`	string	**REQUIRED**	Minimum cumulative amount the next voucher must authorize (base units)

Table 25

The `deposit` field allows the client to determine the correct recovery action. When `requiredCumulative` exceeds `deposit`, the client **MUST** submit `action="topUp"` to increase the on-chain deposit 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 advancing `acceptedCumulative` is accepted. Servers **SHOULD** close the stream if no voucher is received within a reasonable timeout (for example, 60 seconds). Clients **SHOULD** respond by sending a voucher credential to any endpoint protected by the same payment handler.

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 accrued)
- 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
3. Contract settles any delta and refunds remainder to payer
4. Server returns receipt with transaction hash

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

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 (wall-clock time via `block.timestamp`)
3. Server can still `settle()` or `close()` during 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. This ensures clients receive at least one receipt even if the stream is interrupted, while also providing accurate final state when the stream completes normally.

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

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

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

The base Payment Auth spec defines core receipt fields. The session intent extends the receipt with balance tracking:

Field	Type	Description
`method`	string	`"tempo"`
`intent`	string	`"session"`
`status`	string	`"success"`
`timestamp`	string	[RFC3339] response time

Field	Type	Description
challengeId	string	Challenge identifier for audit correlation
channelId	string	The channel identifier
acceptedCumulative	string	Highest voucher amount accepted
spent	string	Total amount charged so far
units	number	**OPTIONAL**: Units consumed this request (e.g., tokens, bytes)
txHash	string	**OPTIONAL**: On-chain transaction hash (present on settlement/close)

Table 26

The `txHash` field serves as the core spec's `reference` field in [I-D.httpauth-payment]. It is **OPTIONAL** because not every response involves an on-chain settlement—voucher updates are off-chain.

The `units` field indicates what was consumed for **this specific request**. When the challenge includes `unitType`, clients can use it to interpret the unit of measure. Clients can compute cost as `units × amount` from the challenge.

Example receipt (per-request with metering):


```
{
  "method": "tempo",
  "intent": "session",
  "status": "success",
  "timestamp": "2025-01-06T12:08:30Z",
  "challengeId": "c_8d0e3b5a9f2c1d4e",
  "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
  "acceptedCumulative": "250000",
  "spent": "237500",
  "units": 500
}
```


Example receipt (on close with settlement):

```
{
  "method": "tempo",
  "intent": "session",
  "status": "success",
  "timestamp": "2025-01-06T12:10:00Z",
  "challengeId": "c_8d0e3b5a9f2c1d4e",
  "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
  "acceptedCumulative": "250000",
  "spent": "250000",
  "txHash":
"0x1a2b3c4d5e6f7890abcdef1234567890abcdef1234567890abcdef1234567890"
}
```


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—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) to reduce storage requirements and operational liability. Servers **MAY** refuse to accept vouchers for channels with no activity exceeding a configured threshold.

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 for unauthenticated requests
- Servers **MUST** enforce `minVoucherDelta` when present to prevent tiny increments
- 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 (field presence, hex encoding, length checks) before expensive ECDSA signature recovery to minimize computational cost of malformed requests.

To mitigate channel griefing via dust deposits:

- Servers **SHOULD** enforce a minimum deposit (e.g., 1 USD 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 that the server can claim later.

13.5. Cross-Contract Replay Prevention

The EIP-712 domain includes `verifyingContract`, binding vouchers to a specific escrow contract address. This prevents replay of vouchers across different escrow contract deployments.

13.6. Escrow Guarantees

The escrow contract provides:

- **Payer protection**: Funds only withdrawn with valid voucher signature
- **Payee protection**: Deposited funds guaranteed (cannot be drained)
- **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. This reduces exposure of the primary key during streaming sessions.

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 for the same message. 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: `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0`
- 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

Implementations **SHOULD** use established libraries (e.g., OpenZeppelin ECDSA) that enforce these requirements.

13.9. Voucher Context and User Experience

The voucher message contains only `channelId` and `cumulativeAmount`. The `channelId` is derived from channel parameters including payer, payee, token, salt, and authorized signer, cryptographically binding these values.

However, wallet signing interfaces may only display the raw `channelId` bytes, making it difficult for users to verify payment details. 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. When a payee operates multiple services using the same channel, voucher-to-service attribution is an implementation concern.

The `challengeId` in the challenge 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`). This means a voucher can only ever advance the channel state forward -- it cannot be "replayed" to extract additional funds because the settlement watermark only moves in one direction.

A separate `sessionHash` binding is therefore 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. The server tracks `highestVoucherAmount` per session and rejects vouchers that do not advance state.
- **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` correlation.

This simplification aligns the spec with the deployed `TempoStreamChannel` contract and the `@tempo/stream-channels` package, neither of which include a session hash in the voucher type.

13.12. Chain Reorganization

On Tempo networks, finality is achieved within approximately 500ms. However, 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

If a chain reorganization invalidates an accepted transaction, the server **SHOULD**:

1. Stop accepting vouchers for that channel
2. Return 410 Gone with problem type `channel-not-found`
3. Log the incident for investigation

13.13. Grace Period Rationale

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

- **Payer protection**: Ensures timely fund recovery if the server becomes unresponsive
- **Payee protection**: Provides time to detect close requests and submit final settlements, even during network congestion or maintenance windows
- **Block time variance**: Allows margin for timestamp variations in on-chain enforcement

Implementations **MAY** use different grace periods in their escrow contracts, but **MUST** clearly document the value and ensure clients are aware.

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	Applicable Methods	Description	Reference
`session`	`tempo`	Streaming payment channel	This document

Table 27

Contact: Tempo Labs (contact@tempo.xyz)

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	Reference
`https://paymentauth.org/problems/session/invalid-signature`	Invalid Signature	402	This document
`https://paymentauth.org/problems/session/signer-mismatch`	Signer Mismatch	402	This document
`https://paymentauth.org/problems/session/amount-exceeds-deposit`	Amount Exceeds Deposit	402	This document
`https://paymentauth.org/problems/session/delta-too-small`	Delta Too Small	402	This document
`https://paymentauth.org/problems/session/channel-not-found`	Channel Not Found	410	This document
`https://paymentauth.org/problems/session/channel-finalized`	Channel Finalized	410	This document
`https://paymentauth.org/problems/session/challenge-not-found`	Challenge Not Found	402	This document
`https://paymentauth.org/problems/session/insufficient-balance`	Insufficient Balance	402	This document

Table 28

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-ryan-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>.

[TEMPO-TX-SPEC] Tempo Labs, "Tempo Transaction Specification", n.d., <https://docs.tempo.xyz/protocol/transactions/spec-tempo-transaction>.

[TIP-20] Tempo Labs, "TIP-20 Token Standard", n.d., <https://docs.tempo.xyz/protocol/tip20/spec>.

Appendix A. Example

Note: In examples throughout this appendix, hex values shown with `...` (e.g., `"0x6d0f4fdf..."`) are abbreviated for readability. 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="tempo",
  intent="session",
  expires="2025-01-06T12:05:00Z",
  request="<base64url-encoded JSON below>"
```

The `request` decodes to:

```
{
  "amount": "25",
  "unitType": "llm_token",
  "suggestedDeposit": "10000000",
  "currency": "0x20c0000000000000000000000000000000000000",
  "recipient": "0x742d35cc6634c0532925a3b844bc9e7595f8fe00",
  "methodDetails": {
    "escrowContract": "0x9d136eEa063eDE5418A6BC7bEafF009bBb6CFa70",
    "chainId": 42431
  }
}
```

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 a price of 0.000025 tokens per LLM token, with a suggested deposit of 10.00 pathUSD (10000000 base units).

A.2. Open Credential

The client retries the **same resource URI** with the open credential:

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

The credential payload for an open action:


```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "tempo",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2025-01-06T12:05:00Z"
  },
  "payload": {
    "action": "open",
    "type": "transaction",
    "channelId":
"0x6d0f4fdf1f2f6a1f6c1b0fbd6a7d5c2c0a8d3d7b1f6a9c1b3e2d4a5b6c7d8e9f",
    "transaction": "0x76f901...signed transaction bytes...",
    "cumulativeAmount": "0",
    "signature": "0xabcdef1234567890..."
  }
}
```


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

During streaming, clients submit updated vouchers to the **same resource URI**. This can use any HTTP method; 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-encoded credential with action="voucher">
```

Or with a regular request that also retrieves content:

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

The credential payload for a voucher update:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "tempo",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2025-01-06T12:05:00Z"
  },
  "payload": {
    "action": "voucher",
    "channelId": "0x6d0f4fdf...",
    "cumulativeAmount": "250000",
    "signature": "0x1234567890abcdef..."
  }
}
```

A.4. Close Request (Same Resource URI)

Close requests are also sent to the same resource URI:

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

The credential payload for a close request:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.llm-service.com",
    "method": "tempo",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2025-01-06T12:05:00Z"
  },
  "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 ITempoStreamChannel {
    struct Channel {
        address payer;
        address payee;
        address token;
        address authorizedSigner;
        uint128 deposit;
        uint128 settled;
        uint64 closeRequestedAt;
        bool finalized;
    }

    function CLOSE_GRACE_PERIOD() external view returns (uint64);
    function VOUCHER_TYPEHASH() external view returns (bytes32);
    function CLOSE_REQUEST_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,
        uint128 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 getCloseRequestDigest(
        bytes32 channelId,
        uint64 requestedAt
    ) external view returns (bytes32);

    function domainSeparator() external view returns (bytes32);

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

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

B.2. Deployed Contracts

Network	Chain ID	Contract Address
Moderato (Testnet)	42431	0x9d136eEa063eDE5418A6BC7bEafF009bBb6CFa70

Table 29

B.3. Contract Source

The reference implementation is available at: https://github.com/tempoxyz/tempo/tree/main/tips/ref-impls/src/TempoStreamChannel.sol

Appendix C. Schema Definitions (JSON Schema)

This appendix provides JSON Schema definitions for implementations that prefer JSON Schema over CDDL.

C.1. Session Request Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://paymentauth.org/schemas/session-request.json",
  "title": "Session Request",
  "type": "object",
  "required": ["amount", "currency", "recipient", "methodDetails"],
  "properties": {
    "amount": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Price per unit in base units (decimal string)"
```

```
    },
    "unitType": {
      "type": "string",
      "description": "Unit type being priced (e.g., llm_token, byte)"
    },
    "suggestedDeposit": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Suggested channel deposit in base units"
    },
    "currency": {
      "type": "string",
      "pattern": "^0x[0-9a-fA-F]{40}$",
      "description": "TIP-20 token address (mixed-case accepted, normalized
to lowercase)"
    },
    "recipient": {
      "type": "string",
      "pattern": "^0x[0-9a-fA-F]{40}$",
      "description": "Payee address (mixed-case accepted, normalized to
lowercase)"
    },
    "methodDetails": { "$ref": "#/$defs/methodDetails" }
  },
  "$defs": {
    "methodDetails": {
      "type": "object",
      "required": ["escrowContract"],
      "properties": {
        "escrowContract": {
          "type": "string",
          "pattern": "^0x[0-9a-fA-F]{40}$"
        },
        "channelId": {
          "type": "string",
          "pattern": "^0x[0-9a-fA-F]{64}$",
          "description": "OPTIONAL: for channel reuse"
        },
        "minVoucherDelta": {
          "type": "string",
          "pattern": "^[0-9]+$",
          "description": "OPTIONAL: server policy hint"
        },
        "feePayer": {
          "type": "boolean",
          "default": false,
          "description": "If true, server pays transaction fees"
        },
        "chainId": { "type": "integer" }
      }
    }
  }
}
```

C.2. Session Payload Schema


```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://paymentauth.org/schemas/session-payload.json",
  "title": "Session Payload",
  "type": "object",
  "required": ["action"],
  "properties": {
    "action": { "enum": ["open", "topUp", "voucher", "close"] },
    "transaction": {
      "type": "string",
      "pattern": "^0x[0-9a-fA-F]+$",
      "description": "Signed transaction bytes"
    },
    "channelId": {
      "type": "string",
      "pattern": "^0x[0-9a-fA-F]{64}$",
      "description": "Channel identifier"
    },
    "cumulativeAmount": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Cumulative amount authorized (decimal string)"
    },
    "signature": {
      "type": "string",
      "pattern": "^0x[0-9a-fA-F]{128,130}$",
      "description": "EIP-712 voucher signature"
    }
  }
}
```


C.3. Session Receipt Schema

Servers **MUST** include `Payment-Receipt` only on successful processing of a session action (2xx responses). On error responses (4xx/5xx), servers **MUST** return Problem Details and **MUST NOT** include a `Payment-Receipt` header. The `status` field is always `"success"` because receipts represent successful acceptance; failures are communicated via HTTP status codes and Problem Details.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://paymentauth.org/schemas/session-receipt.json",
  "title": "Session Receipt",
  "type": "object",
  "required": ["method", "intent", "status", "timestamp", "challengeId",
"channelId", "acceptedCumulative", "spent"],
  "properties": {
    "method": { "const": "tempo" },
    "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}$"
    },
    "acceptedCumulative": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Highest voucher amount accepted"
    },
    "spent": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Total amount charged so far"
    },
    "units": {
      "type": "integer",
      "description": "OPTIONAL: Units consumed this request"
    },
    "txHash": {
      "type": "string",
      "pattern": "^0x[0-9a-fA-F]{64}$",
      "description": "OPTIONAL: On-chain transaction hash (present on
settlement/close)"
    }
  }
}
```


Appendix D. Acknowledgements

The authors thank the Tempo community for their feedback on session payment design.

Authors' Addresses

Liam Horne
Tempo Labs
Email: liam@tempo.xyz

Georgios Konstantopoulos
Tempo Labs
Email: georgios@tempo.xyz

Dan Robinson
Tempo Labs
Email: dan@tempo.xyz

Brendan Ryan
Tempo Labs
Email: brendan@tempo.xyz

Jake Moxey
Tempo Labs
Email: jake@tempo.xyz