# TAP

## TOKEN ACCESS PROTOCOL

*Token-by-token payments for LLM inference, with bilateral halt for fair, low-waste generation.*

# Abstract

Paying for language-model inference today means paying for the whole response, regardless of whether it was worth receiving. A consumer who detects mid-generation that the model has gone off-topic, hallucinated, or violated their format requirements has no mechanism to stop the meter — they pay for every token the model produces, including the tokens they will discard. A producer serving the request must trust that the consumer will pay after delivery; in the API-key model this is enforced by pre-funded accounts, but in agent-driven and trustless settings it remains an open problem.

Tap is a payment protocol that resolves both sides of this asymmetry. It builds on the x402 HTTP payment standard for session discovery and channel bootstrap, then extends x402's one-shot model into the variable-cost streaming case the v2 specification explicitly leaves open. Consumers and producers open a Solana state channel at session start via a standard x402 payment exchange. Output then streams from producer to consumer token-by-token; with each token (or small batch of tokens), the consumer signs an incrementing payment commitment that the producer can settle on-chain at any time. Either party can halt at any token boundary: the consumer halts when output quality degrades, the producer halts when commitments stop arriving. Maximum loss for either side at any moment in the session is bounded by a small, configurable number of tokens — typically a fraction of a cent.

This paper specifies the protocol, the on-chain program that enforces settlement, the SDK that wraps existing model providers, and the integration points with x402. We describe the trust model, analyze adversarial scenarios, and walk through a reference implementation that wraps Gemini 2.5 Flash, with adapter scaffolding for Anthropic, OpenAI, and locally-hosted Llama via Ollama. The protocol is designed for: mid-stream exposure for either party bounded by a small configurable number of tokens (a fraction of a cent at typical pricing); halt-to-stop latency on the order of the configured grace period (defaulting to 200 ms); and per-session settlement enforced by a single on-chain transaction at session close. These properties are enforced by the on-chain program by construction; this paper does not present empirical measurements.

Tap targets the LLM inference market specifically. The same primitives generalize to other metered services, but generalizing is not the contribution; making fair-and-bounded inference payments work end-to-end is.

# Contents

# 1. The waste and trust problem in LLM inference payments

A request to a language-model API today is an all-or-nothing transaction. The consumer commits to paying for whatever the model produces; the producer commits to delivering the model's output. Pricing is denominated per token, but the unit of commercial agreement is the response. By the time a consumer can evaluate output quality, they have already paid for the tokens that produced it.

This works adequately when output quality is high and predictable. It works poorly under three conditions that are routine in production agent workflows:

## 1.1 Wasted output

A non-trivial fraction of LLM responses are not useful. The model misunderstands the prompt, drifts off-topic mid-response, hallucinates a fact the consumer can verify is wrong, or violates a structural requirement (returning prose when JSON was requested, exceeding a length budget, including disallowed content). Production agent systems handle these failures by retrying with modified prompts, switching to fallback models, or escalating to human review. None of these recover the cost of the failed generation.

The cost of wasted output is not theoretical. A consumer running a workflow that calls a frontier model 1,000 times per day, with a 5–10% rate of unusable responses at $0.02 average cost per response, wastes $30–60 per day per workflow. At fleet scale this is real money. More importantly, the consumer cannot detect the failure until the response has fully arrived and been parsed — which is after they have paid for every token, including the ones that revealed the failure.

A consumer that could halt generation at the moment the failure becomes apparent — token 200 of a planned 2,000-token response, when the model first goes off-topic — would pay 10% of the original cost and return the rest of the budget to a useful retry. The economics of agentic workflows change materially when this is possible.

## 1.2 Trust asymmetry

Today's pay-after-delivery model assumes the producer trusts the consumer. This is enforced operationally by API keys, pre-funded accounts, and KYC. In closed ecosystems where every consumer has an account, this works.

It breaks for autonomous agents. An agent making a one-shot request to a service it has never used before cannot pre-fund. An agent operating across many small services cannot maintain accounts at each. Coinbase's x402 protocol, released in 2025 and refined to v2 in 2026, addresses this for fixed-price one-shot requests by having the client send signed payment with the initial request. But x402 v2 explicitly defers variable-cost cases — including LLM inference, where the price depends on the token count the model chooses to produce — to future work; the specification mentions an "upto" scheme for these cases but flags it as unimplemented.

The trust asymmetry runs both ways. A consumer who pays upfront for a maximum trusts the producer not to overcharge or misreport actual usage. A consumer who pays after delivery has the producer trusting them not to refuse payment. Neither direction is solved by current standards in a way that handles streaming, variable-cost output.

### 1.3 The shape of the problem

Both wasted output and trust asymmetry stem from the same structural mismatch: payment is committed at one point in time (request submission or response completion), but value flows continuously across the response. The commitment is coarser than the flow. To resolve both problems at once, the commitment cadence must match the flow cadence — value and payment must move together, token by token.

This is what Tap provides. The remainder of this paper specifies how.

## 2. The core idea: bilateral halt with bounded exposure

Tap turns one transaction into many. Instead of a single up-front or after-the-fact settlement, the consumer and producer settle continuously as tokens are produced. Each token (or small batch of tokens) is paid for at the moment it is delivered. Both sides can stop at any token boundary.

The protocol takes its name from a domestic analogy that captures the mental model exactly. Imagine a metered water tap filling a bucket. The supplier provides water; the meter charges per drop; the homeowner watches the bucket fill. If the water runs clear, the homeowner lets it flow until the bucket is full, then closes the tap and pays for what came out. If the water runs cloudy or shows sediment, the homeowner closes the tap immediately, paying only for the small amount that fell into the bucket before the contamination was noticed. Neither party is wronged: the supplier was paid for every drop they delivered; the homeowner paid only for water that turned out to be useful. The meter ran fairly across the entire interaction.

This is what Tap does for LLM inference. Tokens are drops; the response is a bucket; the consumer's evaluator is the eye watching for contamination; closing the tap is halting the stream. The meter — the on-chain payment program — charges fairly for what flowed before the tap closed, regardless of whether the bucket filled completely. The remainder of this section formalizes the analogy.

### 2.1 The session model

A Tap session has three phases:

- **Open.** The consumer locks a deposit (typically the maximum the session could cost) into a Solana program. This deposit will eventually be split between the consumer and the producer based on what is actually delivered.

- **Stream.** The producer generates tokens. As tokens flow, the consumer signs incrementing payment commitments — each one says "of the deposit, the producer is owed this much so far." The producer holds the latest commitment as proof of payment. No on-chain transactions occur during the stream.
- **Settle.** When generation completes, or either side halts, the producer submits the latest commitment to the on-chain program. The program pays the producer the committed amount and refunds the unused portion of the deposit to the consumer. One on-chain transaction closes the session.

The signed commitment is the heart of the protocol. It is not a promise to pay later — it is a direct authorization to pay from funds already locked at session open. The consumer cannot fail to pay because the money is already escrowed; only the split is being decided. The producer cannot overcharge because the program enforces that the paid amount cannot exceed the most recent signed commitment.

## 2.2 What halt actually means

Halt is not a special operation. It is the absence of further commitments.

If the consumer decides at token 423 that the output is unacceptable, they stop signing new commitments. The producer detects within a small grace period (typically 200ms) that no new commitment has arrived and stops generation. The producer settles using the most recent commitment they hold, which corresponds to roughly token 423. Both sides agreed to a final price; both sides walk away whole; the consumer paid for what they received and not for what would have come next.

Symmetrically, if the producer suspects the consumer is going to stop signing — because, say, the latest commitment arrived later than expected — they can stop generating. The consumer cannot then claim more output than was delivered; the channel state reflects exactly what happened.

The halt is bilateral by construction. Either side withholds their next move (signing a commitment, generating a token), and the session winds down at the token boundary they chose.

## 2.3 Why exposure is bounded

At any moment during a session, the producer has delivered some number of tokens for which they have not yet received a commitment, and the consumer has signed a commitment for some number of tokens they have not yet received output for. Both quantities are bounded by the protocol's batching parameters — typically a handful of tokens, worth a fraction of a cent.

If the consumer stops signing, the producer is at risk for the few tokens they generated past the last commitment. They halt within the grace period. Loss: a fraction of a cent.

If the producer stops generating, the consumer is at risk for any commitment they signed past the last delivered token. The producer cannot claim more than what they signed for. Loss: also a fraction of a cent.

Compared to the alternatives — pay everything upfront and trust the producer to deliver, or pay nothing upfront and trust the consumer to pay after — Tap's exposure window is tiny and configurable. We treat this as the protocol's central security property: at no point can either party lose more than a small bound, even if the other party is fully malicious.

# 3. Why Solana, why state channels

## 3.1 The fee floor problem

Per-token payment requires per-token settlement. If each token's payment incurs an on-chain transaction fee, the fee dominates the payment for any reasonable token price. Solana's base transaction fee of approximately $0.00075 already exceeds the cost of an output token from many models; adding priority fees during congestion makes it worse. Direct on-chain settlement at the token granularity is not economically viable on any current public chain.

State channels resolve this. A channel is opened with one on-chain transaction, settled with one on-chain transaction, and conducts an arbitrary number of payments off-chain in between. The cost of the protocol is a fixed two-transaction overhead per session, amortized across however many tokens flow through the session. For a 2,000-token response at $0.005 per 1,000 tokens, the response payment is $0.01 and the channel overhead is $0.0015 — manageable. For shorter responses, multiple sessions can share a channel by leaving it open across requests, driving the per-session overhead toward zero.

This is the same insight Lightning applied to Bitcoin micropayments a decade ago. The novelty here is not the channel construction; it is the application of the construction to LLM inference, and the protocol details that make per-token streaming work cleanly within the channel.

## 3.2 Why Solana specifically

Three properties of Solana make it well-suited to this application:

- **Low base fee.** Solana's transaction fees are 10–100x lower than Ethereum L1 and roughly 5–20x lower than active L2s under load. The on-chain overhead of opening and closing channels stays small enough not to exclude short sessions.
- **Fast finality.** Solana's roughly 400ms slot time means a settlement transaction confirms in under a second under normal conditions. Halt-to-payout latency is short enough that producers can settle quickly after each session without holding open positions for minutes.
- **Account model.** Solana's program-derived addresses (PDAs) and rent-exempt accounts allow channels to exist as on-chain state without requiring time-locked transaction templates. This produces a simpler channel construction than Lightning's, with fewer cryptographic moving parts.

Tap is not strictly Solana-specific — the construction would work on any chain with sub-cent fees, sub-second finality, and a programmable settlement layer. But Solana is currently the only major chain that meets all three constraints simultaneously, and the reference implementation is built against Solana for this reason.

## 3.3 The relationship to x402

Tap is built on top of x402 rather than alongside it. The x402 specification, released by Coinbase in 2025 and refined to v2 in early 2026, defines an HTTP-native payment standard: an unauthenticated client requests a paid resource, the server responds with HTTP 402 and a structured payment-requirements payload, the client constructs a signed payment instruction and retries with the payload in an X-PAYMENT request header, and the server settles on-chain and returns the response with an X-PAYMENT-RESPONSE header. The standard handles fixed-price one-shot payments cleanly and is well-supported across major payment infrastructure.

The x402 v2 specification, however, explicitly defers two scenarios to future work: variable-cost responses, where the price depends on the work performed, and streaming responses, where value flows continuously across the response. The specification gestures at an "upto" payment scheme for variable-cost cases but flags the implementation as future work; streaming is not addressed at all. These are precisely the scenarios that LLM inference inhabits.

Tap fills this gap. The protocol uses x402 directly for the parts of the flow that x402 already handles well, and extends it for the parts it does not:

- **Discovery via x402.** A Tap-enabled producer advertises itself using a standard x402 payment-requirements response. A client encountering the producer for the first time receives a familiar 402 response with payment requirements pointing to the channel-open endpoint. Any x402-aware client can find Tap producers without specialized discovery code.
- **Channel bootstrap via x402.** The transaction that opens a Tap channel is itself an x402 payment: the consumer's funding transaction follows the x402 payment-payload format, settles via the same facilitator infrastructure, and produces an X-PAYMENT-RESPONSE that the consumer treats as the channel-open confirmation. From x402's perspective, opening a Tap channel is a one-shot payment to a special endpoint.
- **Streaming via Tap.** Once the channel is open, the streaming protocol takes over. Per-token commitment exchange, halt detection, and dispute resolution are all Tap mechanics, occurring at a layer that x402 does not specify.
- **Settlement via x402-compatible facilitators.** When a Tap session closes, the on-chain settlement transaction can be submitted through the same facilitator pattern x402 uses, allowing producers to delegate Solana transaction submission to existing infrastructure rather than running their own settlement validators.

This integration is deliberate. Tap is positioned as the streaming extension to x402's variable-cost and streaming gap rather than as a parallel standard. Consumers that already speak x402 can consume Tap services with minimal changes; producers running on x402 facilitators can offer Tap streaming without changing their settlement infrastructure. The protocols compose at the application layer with no overlap and no contradiction.

Where this paper specifies wire formats and protocol primitives in subsequent sections, we refer to the x402 specification for the discovery and bootstrap layers and define new structures only for the streaming-specific mechanics that x402 does not cover.

# 4. Protocol specification

## 4.1 Roles

A Tap session has two parties:

- **Consumer.** The party paying for inference. Holds the channel deposit. Signs payment commitments. May halt at any token boundary.
- **Producer.** The party providing inference. Receives commitments. May halt at any token boundary. Submits the final commitment for on-chain settlement.

Both parties have Solana keypairs. The consumer's primary wallet authorizes a session keypair (described in Section 4.5) that signs commitments without further user interaction during the session.

## 4.2 Channel lifecycle

### 4.2.1 Opening

Channel opening uses x402 as the bootstrap mechanism. The flow proceeds as follows:

1. The consumer makes an HTTP request to the producer's streaming endpoint.
2. The producer responds with HTTP 402 and a payment-requirements payload (x402 standard format) advertising channel-based payment with the producer's pricing parameters.
3. The consumer constructs a payment payload conforming to x402's payment-instruction format, with the payment recipient being the channel program PDA seeded by (consumer_pubkey, producer_pubkey, channel_nonce) and the amount being the desired channel deposit.
4. The consumer retries the original request with the X-PAYMENT header containing the signed payment payload.
5. The producer (or its x402 facilitator) submits the funding transaction to Solana. On confirmation, the channel program creates the channel state and emits an event.

6. The producer returns an X-PAYMENT-RESPONSE header confirming channel open, along with the streaming response.

The funding transaction itself specifies, beyond the standard x402 fields:

- the deposit amount (the maximum the session can cost)
- the consumer's session-key public key, registered as the authorized signer for in-session commitments
- the producer's recipient address
- the agreed input_price and output_price (in micro-USDC per token), separately denominated to reflect the asymmetric cost structure of LLM inference
- the channel duration (after which the consumer can reclaim the deposit if the producer fails to settle)
- the trailing buffer (described in Section 4.6)
- the prepaid_input amount (input_token_count × input_price), which the on-chain program records as a settlement floor — the producer is guaranteed to receive at least this much regardless of subsequent off-chain events, compensating the unconditional cost of running prefill on the prompt (see §4.9 for how input_token_count is established at session open)

From the consumer's perspective, this is a single HTTP request that happens to carry an x402 payment. From x402's perspective, the request settles a one-shot payment to a special endpoint. From the channel program's perspective, it is the channel-open transaction. All three views describe the same on-chain action.

### 4.2.2 Streaming

The producer begins generating tokens. As each token (or small batch of tokens) is delivered, the consumer signs and sends a payment commitment. The commitment is an Ed25519 signature over a structured message:

```
{
  "schema":          "tap.v1.commit",
  "channel_id":      <PDA pubkey>,
  "sequence":        <u64, monotonically increasing>,
  "cumulative_paid": <u64, micro-USDC, monotonically non-decreasing>,
  "tokens_received": <u32>,
  "timestamp_ms":    <u64>
}
```

The producer holds the most recent commitment received. Each new commitment must satisfy two invariants: sequence > previous_sequence and cumulative_paid >= previous_cumulative_paid. A commitment violating either invariant is rejected and does not advance the channel state.

cumulative_paid is the protocol's central state variable. It does not represent a per-token payment; it represents the total amount the consumer has authorized to be paid to the producer if the channel were

settled at this moment. Settlement uses the latest signed cumulative_paid; intermediate commitments are subsumed by later ones and need not be retained.

### 4.2.3 Pausing and halting

Tap distinguishes between two states that arise when expected next actions stop arriving: pause and halt.

A pause is a temporary suspension of generation while the producer waits for a delayed commitment, or a temporary suspension of commitment-signing while the consumer waits for a delayed token. The party waiting does not abort; they simply stop advancing the session and hold their position. This is the appropriate response to brief network jitter, transient packet loss, mobile connection handoffs, and other routine interruptions that resolve in tens to hundreds of milliseconds.

A halt is a session-ending decision. It occurs when the pause has lasted long enough to indicate that the counterparty is not going to resume. The party that detects the halt initiates settlement using the most recent commitment they hold.

The protocol specifies three timing parameters that govern this:

- **Grace period (default 200ms).** After this duration without an expected next action, the party enters the paused state. Generation or signing stops; existing state is held.
- **Pause timeout (default 5 seconds).** After this duration in the paused state, the session is considered halted. The party initiates settlement.
- **Total session timeout (channel duration).** Set at channel open. After this, the channel is eligible for unilateral close regardless of session state.

The grace period and pause timeout are independent. A consumer on a flaky mobile connection may experience repeated short pauses (network blips falling within the pause window), each resolving without halting; a consumer who has actually disconnected will see the pause window expire and the session settle. The producer's behavior is symmetric: a delayed commitment triggers a pause, not an immediate halt; a sustained absence of commitments through the pause window triggers settlement.

There is no explicit halt or pause message exchanged between the parties. Both states are inferred from absence of expected actions. This is intentional: explicit messages would create attack surface (forged halts, mid-session denial-of-service via spoofed pauses) without improving the legitimate cases that timing-based detection already handles.

Producers publish their grace and pause-timeout values in the session-open metadata (Section 4.8), allowing consumers to choose providers whose timing parameters match their network conditions. A producer serving a global agent network with high jitter tolerance might publish a 1-second grace period and 30-second pause timeout; a producer serving low-latency colocated workloads might publish 50ms and 1 second. The protocol does not mandate a specific configuration; it requires only that the parameters be agreed at session open.

### 4.2.4 Settlement

Either party can initiate settlement by submitting the latest commitment they hold to the on-chain channel program. The program performs the following steps:

- Verifies the commitment signature against the consumer's session-key public key registered at channel open.
- Verifies channel_id and that the channel is in the active state.
- Verifies prepaid_input ≤ cumulative_paid ≤ deposit.
- Transfers cumulative_paid USDC from the channel PDA to the producer.
- Refunds (deposit - cumulative_paid) to the consumer.
- Marks the channel closed.

If both parties hold commitments and the lower-sequence one is submitted first, the higher-sequence one can be submitted within a brief dispute window (default 30 seconds) to supersede. After the window, settlement is final.

## 4.3 Adaptive batching

Signing one commitment per token is feasible but wasteful. Modern models stream at 50–200 tokens per second; one commitment per token means hundreds of signatures per second across the network. The protocol therefore batches commitments adaptively.

The consumer's runtime maintains a current batch size K (in tokens). Commitments are sent every K tokens rather than every single token. The cumulative_paid field still represents the exact value owed — batching only changes the cadence of commitment messages, not their semantics. A commitment covering 5 tokens is identical to 5 separate commitments with the same final cumulative_paid.

K is tuned by an algorithm modeled on TCP congestion control:

1. Start at K=1 (one commitment per token).
2. If the producer is consistently waiting for commitments — that is, the producer's pending unpaid value approaches the configured maximum — increase K multiplicatively.
3. If the producer is comfortably caught up, decrease K additively toward 1.
4. Cap K at K_max, computed so that K × output_price ≤ MAX_UNPAID_VALUE.

On a 50ms round-trip path with token generation at ~100 tokens per second, K settles at approximately 5 in steady state — large enough to keep the network pipeline full, small enough that the producer's unpaid exposure stays under a fraction of a cent. On a fast local path (sub-10ms RTT) K stays near 1 and settlement is effectively per-token.

## 4.4 Halt granularity and quality evaluation

The consumer's halt decision is application-level. The protocol provides the mechanism (stop signing) but does not specify the policy (when to stop). Common evaluators include:

- **Schema validation.** If the response is expected in a structured format (JSON, XML, a specific markup), check incoming tokens against a streaming validator and halt on detected violation.
- **Topic adherence.** Run a small classifier over accumulated output checking for off-topic drift; halt when score crosses a threshold.
- **Length budget.** Halt when accumulated tokens exceed a configured maximum, regardless of whether the model intends to produce more.
- **Content policy.** Halt on detection of disallowed content (profanity, PII, copyrighted text), useful for compliance-sensitive workflows.
- **Manual halt.** In interactive consumer applications (chat UIs), expose a stop button that triggers halt directly from user action.

Multiple evaluators can run in parallel. The consumer SDK provides a plugin interface: any function from accumulated output to {continue, halt} can be registered and runs concurrently with token consumption. The first evaluator to return halt stops the session.

## 4.5 Session keys

Hardware wallets and browser-extension wallets are not designed to sign hundreds of messages per second without user interaction. Direct wallet signing of every Tap commitment would be unusable for human-driven sessions and unsafe for autonomous agents.

At channel open, the consumer's primary wallet authorizes a session keypair to sign commitments on its behalf. The session key is generated in-memory at session start, registered in the channel program as part of open_channel, signs commitments without further user interaction during the session, and is destroyed at session close.

The session key cannot exceed the channel deposit because the on-chain program enforces cumulative_paid <= deposit at settlement. A compromised session key results in loss bounded by the channel deposit; the consumer's primary wallet remains uncompromised. This is a strictly better security model than full-wallet signing for high-frequency authorization workloads, and it is the same pattern used by Solana's existing session-key implementations in Squads, Privy, and similar wallet infrastructure.

## 4.6 Trailing buffer

In any streaming protocol, the case where the network drops mid-session must be handled. Without specific provision, a disconnect produces an asymmetry: the producer has delivered some tokens for which they have not received commitment, and the consumer has received tokens for which they have not authorized payment. Resolving fairness via dispute is expensive.

Tap specifies a trailing buffer mechanism. At channel open, the consumer pre-authorizes the producer to claim up to a small fixed quantity of tokens (typically 5–10) beyond the most recent signed commitment, in the case where the channel closes without further commitment exchange. The on-chain program enforces this bound: a producer may settle for cumulative_paid + (trailing_buffer × output_price) without further consumer signature, but no more. The trailing buffer applies only to output streaming; input cost is already secured by prepaid_input at channel open and is unaffected.

The trailing buffer is not a vulnerability for the consumer in normal operation. Its purpose is to make graceful disconnection a no-op. When the network drops, both parties settle whole without dispute. In the malicious case — a consumer halting deliberately to avoid the last batch's payment — the producer is compensated for an amount approximately equal to what they delivered. The maximum loss for an honest consumer in any disconnection scenario is the buffer value, typically equivalent to a small fraction of a cent.

## 4.7 Cross-request channel reuse

Channels can be reused across requests to the same producer. A consumer that expects to make many requests against the same provider opens a channel once with a larger deposit, runs many sessions through it (each ending with a commitment-update rather than a settlement), and settles the channel only periodically — daily, weekly, or when the deposit nears exhaustion.

The protocol supports this directly: the channel state machine has no notion of a "request" that closes the channel. A session is just a stretch of streaming inside an active channel. The producer maintains the latest cumulative_paid across all sessions; the consumer continues incrementing it. On final settlement, the producer is paid the total across all sessions in one transaction.

This drives the per-session protocol overhead toward zero for high-volume consumers. A consumer doing 10,000 model calls against the same provider over a month pays for one open and one close transaction in total — not 20,000.

## 4.8 Producer-published metadata

A producer is required to publish its operational parameters as part of the x402 payment-requirements payload returned at session-open negotiation. This is what allows a consumer to know, before committing any payment, exactly what they are agreeing to.

The published metadata includes:

- **Pricing.** Two prices in micro-USDC: input_price (per prompt token, charged once at channel open) and output_price (per generated token, charged incrementally as output streams). Pricing is denominated separately because real-world LLM inference has asymmetric input/output cost ratios, typically in the range of 1:3 to 1:5. The producer also publishes its tokenizer identifier

(e.g. “cl100k_base”, “claude-tokenizer-v2”, “llama-3-tokenizer”), allowing consumers to count input tokens locally and verify the producer’s count before opening the channel.

- **Limits.** Maximum unpaid value the producer will tolerate before halting; minimum and maximum channel deposit; maximum session duration.
- **Timing.** Grace period and pause timeout values used by this producer; expected token generation rate (used by clients to size adaptive batching).
- **Trust parameters.** Trailing buffer size; dispute window length; whether the producer accepts session-key authorization.
- **Capability.** Supported model identifiers; supported response formats (text, JSON, structured output); supported evaluator-halt signals (the producer may, optionally, accept hints from the consumer about why a halt is occurring, used for analytics).
- **Identity.** Producer pubkey, recipient address, optional verifiable credentials linking the operating entity to a known organization.

Consumers verify these parameters against their own policy before opening a channel. A consumer's runtime can be configured to refuse channels with unfavorable parameters — for example, refusing producers with trailing buffers larger than 10 tokens, or producers whose token price exceeds a configured maximum. This shifts the trust burden from session-time to discovery-time: by the time the channel opens, the consumer has already audited the terms.

Publishing this metadata in a standardized format also enables third-party indexers and reputation systems. A registry of Tap-enabled producers, ranking them by historical reliability, parameter generosity, and complaint rate, becomes possible without any protocol-level cooperation. We expect such registries to emerge organically as the ecosystem grows; the protocol does not specify them, but it does specify the metadata that makes them possible.

## 4.9 Input token negotiation and prefill semantics

Output tokens stream and can be halted at any boundary. Input tokens cannot. By the time the model has begun generating output, the producer has already executed the model’s prefill pass over the entire prompt — an irreversible compute cost that is, on most current architectures, a non-trivial fraction of total inference cost (10–40% depending on prompt length and model). The protocol must therefore charge for input tokens up front and protect the producer’s prefill investment, while still preventing the producer from charging for input that was never actually processed.

Tap resolves this with a session-open negotiation that establishes input cost before any prefill occurs and locks it on-chain as part of the channel-open transaction. The flow:

7. The consumer submits the prompt to the producer’s discovery endpoint as part of the initial HTTP request. No payment is attached at this stage; the request is unauthenticated.

8. The producer tokenizes the prompt with its declared tokenizer and returns HTTP 402 with the standard payment-requirements payload extended to include input_token_count (the producer's authoritative count for this prompt) and prepaid_input (input_token_count × input_price). All other operational parameters from §4.8 are also returned, exactly as for any Tap session-open.
9. The consumer optionally verifies the count by tokenizing the prompt locally with the producer's declared tokenizer. Tokenizers are deterministic and publicly available for every major model family. Disagreement implies the producer is using a non-standard tokenizer or attempting to inflate the input charge; the consumer aborts before paying anything. Casual consumers may skip this verification and accept the producer's count on faith, with reputation as the backstop.
10. The consumer constructs the channel-open transaction with deposit_micro covering the full session budget and prepaid_input set to the agreed input cost. The on-chain program records both fields and enforces, at every subsequent settle and close, the invariant prepaid_input ≤ cumulative_paid ≤ deposit.
11. On channel-open confirmation, the producer begins prefill. From this point forward the producer's input cost is secured: regardless of subsequent consumer behaviour, the on-chain program will pay out prepaid_input to the producer at settlement or close.
12. Output streaming proceeds as specified in §4.2.2. The first signed commitment in the session has cumulative_paid ≥ prepaid_input by construction; subsequent commitments accumulate output cost on top.

This design closes the input-token freeloading attack completely. A consumer cannot send a prompt, induce the producer to run prefill, and then refuse to pay; the prepaid_input is locked on-chain before the producer learns of channel confirmation, and the producer does not begin prefill until that confirmation is observed. Reciprocally, a producer cannot inflate the input charge unilaterally, because the consumer sees input_token_count in the 402 response and either verifies it or accepts it knowingly. What remains is the residual case of a producer accepting prepaid_input and never delivering output (§5.3.8), which the protocol bounds to the input cost but does not eliminate at v1.

For multi-turn sessions over a reused channel (§4.7), the same negotiation occurs at each turn before the new prompt is processed. The channel program does not need to know about turns; cumulative_paid simply incorporates the new prepaid_input as the consumer signs the first commitment of the new turn, advancing the floor that the producer can claim.

# 5. Trust model and adversarial analysis

## 5.1 What the protocol guarantees

Tap provides bounded-loss guarantees, not unconditional fair exchange. Specifically:

- **The consumer cannot lose more than (current_cumulative_paid + trailing_buffer × output_price)** at any moment in the session, regardless of producer behavior. If the producer disappears, takes payment without delivering, or delivers garbage, the consumer's loss is bounded by what they have already authorized plus the trailing buffer.
- **The producer cannot lose more than (delivered_output_tokens × output_price - latest_cumulative_paid)** at any moment, regardless of consumer behavior. If the consumer stops signing, refuses to settle, or attempts to claim more than they paid, the producer's loss is bounded by tokens delivered past the latest commitment, capped at the K used in adaptive batching.
- **Neither party can be forced to extend the session.** Either party halts by ceasing their next action. There is no mechanism by which the other party can compel further generation or further commitment.

These guarantees hold under the standard cryptographic assumptions (Ed25519 unforgeability, SHA-256 collision resistance) and the assumption that the underlying Solana ledger eventually permits transaction submission. Indefinite censorship of the chain would defeat any payment protocol; we accept this as a precondition rather than addressing it.

## 5.2 What the protocol does not guarantee

Some properties are explicitly not provided:

- **Output quality.** The protocol cannot verify that the tokens delivered are subjectively good. A producer can commit to delivering 1,000 tokens and deliver 1,000 tokens of grammatical garbage; the protocol will pay them. Quality is the responsibility of the consumer's evaluator (Section 4.4) and of the producer's reputation, not of the cryptographic protocol.
- **Model honesty.** The protocol cannot verify that the producer used the model they claimed. A consumer paying for Claude 3.5 Sonnet output cannot, from the protocol layer alone, confirm that they did not receive output from a cheaper model. Mitigations include trusted execution environments (where the producer runs inference in an attested enclave) and reputation systems; these are out of scope for v1.
- **Privacy.** Channel openings, settlements, and commitments-on-dispute are visible on-chain. The producer learns the consumer's pubkey at session open. In-session commitment exchange is peer-to-peer and not visible to third parties unless a party publishes them, but the existence of the channel is public. Tap is not a privacy-preserving protocol.

## 5.3 Adversarial scenarios

### 5.3.1 Consumer attempts to consume without paying

The consumer requests inference, receives some quantity of tokens, and refuses to sign further commitments past some point. The producer's runtime detects the absence of commitments within the grace period and halts generation. The producer settles using the most recent commitment held, plus the trailing buffer. The consumer has paid for the tokens received plus a small buffer overage. The attacker's gain is bounded by the trailing buffer (sub-cent); they have lost access to the remainder of their channel deposit pending settlement. The attack is not profitable.

### 5.3.2 Producer attempts to charge without delivering

The producer accepts the channel and sends nothing, or sends garbage tokens, or otherwise fails to provide useful output. The consumer's evaluator detects the failure and stops signing commitments. The producer can only settle with the latest commitment they hold, which corresponds to the last useful tokens delivered (or zero, if none were delivered). The producer cannot extract payment beyond what was actually accepted by the consumer.

### 5.3.3 Consumer submits stale commitment on settlement

The consumer initiates settlement using an older signed commitment than the most recent one held by the producer, hoping the producer will not contest in time. The producer detects the discrepancy via on-chain monitoring during the dispute window and submits the higher-sequence commitment, which supersedes the consumer's submission. The consumer's deposit may be reduced by a small dispute penalty if configured.

### 5.3.4 Producer submits stale commitment

Symmetric to 5.3.3 in the opposite direction. The consumer submits the higher-sequence commitment within the dispute window; the producer's stale claim is superseded.

### 5.3.5 Consumer pads context to inflate token count

Token-priced billing creates an incentive for the producer to inflate the token count. A malicious producer could pad responses with whitespace, repeated tokens, or low-information content to maximize charged tokens.

This is the genuinely hard adversarial case for token-priced inference, and the protocol does not solve it cryptographically. Mitigations are application-level: the consumer's evaluator can detect padding (high-frequency repetition, low entropy, deviation from expected response shape) and halt; reputation systems penalize producers with statistical anomalies; for high-value workloads, attested execution provides cryptographic assurance of model identity. We treat token-padding as a known limitation that the protocol's halt mechanism partially mitigates but does not eliminate.

### 5.3.6 Network partition mid-session

The network drops between consumer and producer in the middle of a session. The producer has delivered some tokens past the last commitment; the consumer has signed some commitments past the last delivered token. After the partition heals, neither party may want to resume. The producer settles using the last commitment received, plus the trailing buffer. Both parties end whole within the buffer's bounds. No dispute is required for the typical case.

### 5.3.7 Producer inflates the input token count

The producer reports an input_token_count larger than the prompt actually tokenizes to, hoping to claim a higher prepaid_input. The consumer's SDK, having retrieved the producer's declared tokenizer_id from the §4.8 metadata, runs the same tokenizer locally and detects the discrepancy. Counts are deterministic, so honest disagreement is impossible — any mismatch implies misbehavior. The consumer aborts the channel-open transaction; no on-chain state is created and no payment flows. The attack costs the producer a 402 round-trip's worth of compute and zero consumer funds. Reputation systems penalize producers with statistical anomalies in tokenization accuracy. This attack is not profitable.

Consumers that skip local verification accept the producer's count on trust. This is acceptable for low-stakes sessions and producers with established reputation, and unacceptable for adversarial settings — the protocol exposes the choice to the consumer rather than mandating one cost model.

### 5.3.8 Producer accepts prepaid input and delivers no output

The producer accepts a channel-open carrying a non-zero prepaid_input, then never streams output (or streams a single token of garbage), and settles immediately for prepaid_input. The consumer has paid the input cost for no useful output. This is the residual asymmetric risk that the input-token negotiation in §4.9 does not eliminate; it bounds rather than removes.

The protocol mitigates the attack on three axes. First, the consumer's loss is bounded: the maximum extractable amount per session is exactly the input cost, with no exposure on the output budget, since the producer cannot exceed prepaid_input without a signed commitment from the consumer. Second, the producer is required to publish a maximum time-to-first-token in the §4.8 metadata; consumer SDKs treat violation as a halt trigger and surface the event to reputation infrastructure. Third, this failure mode is statistically detectable across sessions — a producer with an anomalous rate of input-only settlements is identifiable to registries (§4.8) and to the consumer's own historical tracking. Honest producers have no reason to settle without delivering output; the asymmetric pattern is itself the signal.

The cryptographic answer to this attack is TEE-attested execution (§10.2): an enclave can sign "I have begun output generation for prompt P," making input payment contingent on attested progress rather than the producer's assertion of progress. We treat this as the v2 closure of the residual risk and accept reputation plus bounded loss as the v1 mitigation. The protocol's honesty about this scope cut is itself the design choice — pretending otherwise would understate the asymmetry that any non-attested streaming payment system inherits.

### 5.4 Comparison to existing trust models

| Model | Consumer trusts producer for... | Producer trusts consumer for... |
|---|---|---|
| **Pre-paid API (current)** | Full request value, every request | Nothing |
| **Post-paid invoice (current enterprise)** | Nothing | Full monthly bill, with collections recourse |
| **x402 fixed-price one-shot** | Full request value | Nothing |
| **Tap** | ≤ 1 batch + trailing buffer (sub-cent) | ≤ 1 batch (sub-cent) |

Tap's contribution is not unconditional safety — that is impossible without a trusted third party — but a structural reduction in the trust window from "full request value" to "a few tokens of inference cost." In production agent workflows where individual requests are small and frequent, this is the difference between an unsafe pattern and a safe one.

## 6. Reference implementation

### 6.1 Components

The reference implementation comprises four components, all open-sourced under the MIT license:

- **On-chain program.** An Anchor 0.32 program implementing the channel-open, settle, dispute, and close instructions, organised one-task-per-file across the four instructions, two state accounts (Channel and CommitMessage), and the constants/errors/events modules. Deployed to Solana devnet under a documented program ID.
- **Producer SDK.** A Python library that wraps streaming model endpoints with payment validation. Adapters for the Gemini HTTP API, the Anthropic Python SDK, the OpenAI Python SDK, and Ollama HTTP servers. Handles commitment verification, batch tracking, halt-on-no-commit, and on-chain settlement.
- **Consumer SDK.** A Python library that exposes a streaming iterator wrapping the underlying model call. Handles session-key generation, commitment signing, evaluator plugins, halt coordination, and settlement initiation. Designed to be drop-in for the standard streaming patterns in the supported model SDKs.
- **Demo agent.** A reference agent that consumes Gemini 2.5 Flash output via the consumer SDK with a configurable evaluator. Demonstrates the halt-on-quality flow with a React dashboard showing token-by-token cost accrual, signed commitments, and the on-chain settlement event.

## 6.2 Producer SDK usage

Wrapping an existing streaming endpoint requires a few lines of integration. Example for an Anthropic-backed producer:

```
from tap import TapProducer
from anthropic import Anthropic

producer = TapProducer(
    program_id="<deployed program id>",
    keypair=load_producer_keypair(),
    input_price_micro=3,    # micro-USDC per input (prompt) token
    output_price_micro=15,  # micro-USDC per output (generated) token
    tokenizer_id="claude-tokenizer-v2",
    max_unpaid_value=5000,  # max unpaid tokens before halt
    trailing_buffer=10,
)

@producer.handler("/v1/messages")
async def handle(request, channel):
    client = Anthropic()
    stream = client.messages.stream(**request)
    async for token in producer.wrap_stream(stream, channel):
        yield token
```

The wrap_stream call handles all protocol mechanics: it accepts incoming commitments, verifies their signatures, monitors the unpaid-token count, and stops the iteration when commitments lapse beyond the grace period. The application code is identical to a non-Tap streaming handler.

## 6.3 Consumer SDK usage

Consuming a Tap-enabled endpoint similarly looks like a normal streaming call, with an evaluator passed alongside:

```
from tap import TapConsumer, evaluators

consumer = TapConsumer(
    program_id="<deployed program id>",
    wallet=load_consumer_wallet(),
)

session = consumer.open_session(
    producer_url="https://provider.example.com/v1/messages",
    deposit_micro=20_000,           # max session cost: $0.02
    evaluator=evaluators.compose([
        evaluators.json_schema(expected_schema),
        evaluators.topic_drift(reference="customer support"),
        evaluators.length_cap(max_tokens=2000),
    ]),
)
```

```
async for token in session.stream(messages=[...]):
    print(token, end="")

# session.settle() runs automatically on close
```

The evaluator plugin interface accepts any callable that takes accumulated output and returns continue or halt. Common evaluators are provided as utility functions; custom evaluators can be defined inline.

### 6.4 Demo flow

The demo runs against the program deployed on Solana devnet. A typical session looks like this:

1. Consumer requests a JSON response from the prompt form. The browser POSTs to the runner backend, which calls the consumer SDK; the consumer SDK opens a channel against the producer with a configurable deposit (default $0.05).
2. Producer (wrapping Gemini 2.5 Flash) begins streaming output tokens; the consumer signs an incrementing commitment every K tokens.
3. Real-time dashboard shows tokens accumulating, cost ticker climbing, signed commitments being exchanged.
4. With the JSON-schema evaluator armed, asking for prose induces the schema validator to flip to HALT mid-stream. The consumer stops signing; the producer detects the absence of commitments within the configured grace period and stops generating.
5. The producer submits the latest commitment via the settle instruction; after the dispute window, close moves cumulative_paid USDC to the producer and refunds the remainder to the consumer. Both transactions are linked from the dashboard timeline.
6. A side panel contrasts the same request against a direct API: the full deposit would have been paid for unusable output, with no recovery path.

The demo's purpose is to make the protocol's central value proposition visible: a real, on-chain, irreversibly-settled refund for the unused portion of a generation that went wrong.

## 7. Properties enforced by construction

This section catalogs properties Tap guarantees as logical or mathematical consequences of the specification, rather than as empirical claims. Each follows from the on-chain program (Appendix A) or the wire format (Appendix B); none requires measurement.

Settlement is deterministic. Given a valid signed commitment with prepaid_input ≤ cumulative_paid ≤ deposit, the settle instruction transfers exactly cumulative_paid micro-USDC to the producer and refunds exactly (deposit − cumulative_paid) to the consumer. No other distribution of funds is reachable through

the program. The producer cannot extract more than the consumer signed; the consumer cannot recover more than the program refunds. (Appendix A.2.)

Every channel reaches a terminal state. A channel resolves to one of two outcomes within bounded time: settled (one party submitted a commitment) or closed-by-timeout (duration_secs elapsed without settlement). Both paths are callable by either party. The on-chain close instruction enforces the prepaid_input floor when no commitment exists, so a producer who completed prefill but never received an output commitment is still compensated for the prefill cost. There is no execution trace through Appendix A in which a channel's funds become permanently inaccessible.

Input cost is fixed at session open. The negotiation in §4.9 ensures that, given a published tokenizer_id and a prompt, both parties can compute prepaid_input independently before any payment flows. The on-chain program records prepaid_input as part of channel state at open, fixing it for the session's lifetime. Neither party can change the input charge after the fact; disagreements at the negotiation stage abort the session before any on-chain state is created. This is what closes the freeloading attack analyzed in §5.3.7.

Channel-open is x402-compliant. The funding transaction conforms to the x402 payment-payload format defined in Appendix B.1. An x402 facilitator that does not understand Tap can still settle the open transaction as a one-shot payment to the channel program PDA, which is the only behavior the channel-open requires from the facilitator. Tap's streaming-specific extensions live in extra fields that x402-only clients ignore.

These are the substantive guarantees Tap provides today. They do not include performance characteristics — throughput overhead, halt-latency distribution, or fee impact under specific workloads — because those depend on deployment and workload, not on the protocol itself.

# 8. Limitations

This section enumerates the known limitations of the protocol as specified. We discuss them not because we have answers but because acknowledging them is part of responsible specification work.

## 8.1 Quality is not cryptographically verifiable

As discussed in Section 5, the protocol can verify that a producer delivered tokens but not that those tokens are subjectively useful. A producer can commit to and deliver high-token-count low-quality output. Halt-on-evaluator partially mitigates this by detecting common quality failures, but a sophisticated adversary can produce output that passes naive evaluators. Mitigations are application-level (better evaluators, reputation systems) and infrastructure-level (TEE attestation), not protocol-level.

## 8.2 Token padding

Producers earn per-token, creating an incentive to maximize token count. While most production providers (Anthropic, OpenAI) are not credibly going to pad token counts on a per-customer basis, the protocol does not provide cryptographic protection against this. Consumer-side detection (entropy checks, repetition flagging) handles obvious cases; subtle padding is harder. In practice, the reputational cost to a major API provider of being caught padding tokens is significant, and we expect this risk to remain low for established providers.

### 8.3 Model substitution

Consumers paying for high-quality model output have no protocol-level guarantee that the producer ran the claimed model. A producer could substitute a cheaper model for some fraction of requests undetectably. TEE-attested execution resolves this for compliance-sensitive workloads; for general use, reputation and spot-checking remain the primary mitigations.

### 8.4 Channel-open overhead for short sessions

Solana’s base + priority fees on the channel-open and close transactions make Tap economically viable only for sessions of approximately a cent or more, or for high-volume consumers who reuse channels. Short, infrequent sessions are better served by direct one-shot payment via x402 or a similar protocol. Tap is not the right tool for every payment shape.

### 8.5 Privacy

Channel openings and settlements link consumer and producer pubkeys on-chain. For workloads where transaction metadata is sensitive, application-layer privacy measures (mixing, blinded routing, separate identity per session) are required. The protocol provides no native privacy.

## 9. Beyond LLM inference

Tap is specified for LLM inference because that is the workload most acutely in need of fair, fine-grained payment. But the protocol's design choices — streaming commitments over a state channel, bilateral halt, bounded exposure, x402-compatible bootstrap — are not specific to language models. They apply to any service where value is delivered as a continuous flow and where the consumer benefits from being able to halt mid-flow.

This section briefly notes the categories where the same architecture could be applied without modification, deferring full specification of these applications to subsequent work. We mention them not as commitments but as evidence that the protocol's primitives are general enough to support adjacent use cases as the ecosystem matures.

## 9.1 Audio streaming

Text-to-speech and music streaming services produce output as a continuous time-series. Per-second or per-chunk payment with halt-on-quality maps onto the same channel construction as token-based inference. The unit shifts from "output token" to "audio second," but the protocol mechanics are identical: the consumer signs incrementing commitments as audio streams, halts on detected quality issues (broken encoding, content violations, off-topic content for spoken responses), and settles on session close.

## 9.2 Live and on-demand video

Video streaming, particularly for long-form content where viewers may abandon early, benefits from per-minute or per-segment settlement. The protocol's halt mechanism allows a viewer to stop paying at the moment they stop watching, with the channel settling for actual consumption. The trust model is the same: producer's exposure bounded by the unpaid window, consumer's by the trailing buffer. Distribution-platform integrations may layer additional revenue-sharing logic on top of the base protocol; the channel construction itself is unchanged.

## 9.3 Cloud compute and GPU rental

Per-second compute pricing for training jobs, inference at scale, and rendering workloads is currently bundled into hourly or longer billing increments. Tap's channel construction supports per-second metering with the same primitives: the consumer signs commitments as compute time elapses, halts when their job finishes or when output indicates the compute is misbehaving, and the channel settles for actual elapsed time. This is particularly valuable for spot compute and decentralized GPU networks where consumers cannot pre-commit to long-duration contracts.

## 9.4 Metered API access

APIs that bill per-call but charge for variable amounts of work per call (database queries with variable result sizes, search APIs with variable index hits, geospatial APIs with variable computation depth) currently approximate variable cost with tiered pricing or flat rates. Tap allows true per-unit-of-work pricing: the response streams a header indicating cost as work proceeds; the consumer signs commitments incrementally; the session settles for actual work performed.

## 9.5 What changes, what doesn't

Across these applications, the on-chain program, the channel state machine, and the x402 bootstrap flow are unchanged. What changes is the per-application metadata published at session open (what the unit is, what the price is, what halt signals make sense), the consumer-side evaluator (what indicates quality

issues for this content type), and the producer-side wrapper that adapts the protocol to the underlying service's streaming format.

The protocol is, in this sense, a substrate. LLM inference is the application this paper specifies because it is the most urgent and the easiest to demonstrate. The same substrate supports the broader category of streaming digital services. We expect implementations for the applications above to follow the same general architecture as the LLM reference implementation, with application-specific adapters layered above the channel program.

# 10. Future work

This section describes extensions to the protocol that have been considered but deliberately deferred from v1. These are not bugs in the current specification; they are deliberate scope cuts that allow v1 to ship as a focused, well-specified system. We list them so that implementers and reviewers know what is coming and what is intentionally absent.

## 10.1 Hash-locked atomic delivery

Streaming applies to outputs delivered incrementally. Some outputs — generated images, complete audio files, compiled artifacts — are delivered as atomic payloads where partial delivery has no value. For these cases, a hash-locked atomic exchange protocol can extend Tap's trust model to single-payload deliveries: the consumer's payment is locked behind a hash; the producer reveals the preimage to claim payment, simultaneously delivering the decryption key for the encrypted payload. v2 would specify this construction, allowing Tap channels to handle mixed streaming and atomic deliveries within a single session.

## 10.2 TEE-attested model identity

Section 5.2 notes that the protocol cannot verify the model used by the producer. For high-value workloads where model substitution is a concern, attestation via trusted execution environments (Intel SGX, AMD SEV, AWS Nitro Enclaves) provides cryptographic assurance that a specific model binary executed within an audited environment. v2 would specify how attestations are exchanged at session open and how consumers verify them. This is straightforward in principle; the work is in defining the attestation format and the verifier infrastructure.

## 10.3 Hub-routed sessions

v1 assumes direct channels between consumer and producer. As the ecosystem grows, consumers will want to consume from many producers without maintaining a separate channel with each. v2 would specify a hub-routing extension in which a consumer maintains a channel with a routing hub, the hub

maintains channels with producers, and payments route through correlated commitments on each leg. This is structurally similar to Lightning Network's HTLC routing and has been explored in the prior literature; the work is in adapting the construction to Tap's specific commitment format and in specifying hub discovery and reputation.

### 10.4 Decentralized facilitators

x402 facilitators currently operate as trusted intermediaries that submit on-chain transactions on behalf of clients. v2 would specify a decentralized facilitator model — a network of independent operators whose participation is incentivized economically and whose correctness is enforced by stake-slashing for misbehavior. This addresses the centralization concern that any facilitator-based protocol inherits.

### 10.5 Consumer-side privacy

v1 reveals the consumer's pubkey to the producer at channel open. For privacy-sensitive workloads (medical inference, legal queries, sensitive personal questions), v2 would specify how consumers can interact with producers via blinded channels — using stealth addresses, blind-signed commitments, or zero-knowledge proofs of channel ownership. The mechanisms are well-understood from the broader cryptography literature; the work is in specifying which mechanism best fits Tap's session model.

### 10.6 Standardized evaluator library

v1 defines the evaluator interface but leaves implementations to consumers. A standard library of high-quality evaluators — JSON schema validation, topic-drift detection, content-policy enforcement, length budgets, hallucination detection — would lower the integration barrier for new consumers. v2 would publish such a library alongside the SDK, with rigorous benchmarks for evaluator accuracy and latency. This is engineering work, not protocol work, but it materially affects adoption.

## 11. Conclusion

Tap addresses a specific, immediate problem: paying for LLM inference today wastes consumer money on output that turns out to be unusable, and exposes both consumers and producers to trust assumptions that don't hold in agent-driven workflows. The fix is to settle continuously rather than at request boundaries, with bilateral halt at any token boundary, on rails fast and cheap enough to make the per-token granularity economically viable.

The protocol is built on x402, extending its one-shot payment model into the variable-cost streaming case the v2 specification explicitly leaves open. Producers running on x402 facilitators can offer Tap streaming without changing their settlement infrastructure. Clients that already implement x402 can discover and

connect to Tap services using familiar primitives. Tap is positioned as the streaming extension to x402, not as a competing standard.

The protocol is not theoretical. Solana state channels handle the off-chain commitment exchange. A small Anchor program enforces settlement. Reference SDK integrations wrap Gemini, Anthropic, OpenAI, and Llama-family models with a few lines of code. The reference implementation is deployed to devnet, and the demo flow shows end-to-end halt-and-refund running against the live program.

We expect Tap to find adoption first among AI infrastructure providers — the entities running agent workflows at scale, who feel the cost of wasted output most directly and who have the technical sophistication to integrate a new payment rail. Wider adoption depends on what those early adopters demonstrate the protocol can do; the protocol's value will be established by what it enables in production, not by what we claim about it on paper.

The path forward is straightforward: deploy, measure, harden, and iterate. We invite review of the specification, critique of the design choices, and competing implementations. The contribution we hope to make is not the only possible answer to the LLM payment problem — it is a specific, shipped answer that demonstrates the problem is solvable with current infrastructure and a small, focused protocol layered on the x402 foundation.

# Appendix A — On-chain program interface

Public instruction interface of the Tap channel program. Implementations must conform to these signatures to interoperate.

## A.1 open_channel

```
open_channel(
  consumer:        Pubkey,
  producer:        Pubkey,
  session_key:     Pubkey,
  deposit_micro:   u64,
  input_price:     u64,
  output_price:    u64,
  prepaid_input:   u64,
  duration_secs:   u32,
  dispute_secs:    u32,
  trailing_buffer: u32,
) -> ChannelState
```

Creates a new channel PDA and transfers deposit_micro USDC into it from the consumer’s USDC token account. Registers session_key as the authorized signer for commitments. Records prepaid_input as the on-chain settlement floor: at any subsequent settle or close, the producer is guaranteed to receive at least prepaid_input regardless of off-chain commitment state, and the consumer can recover at most

(deposit_micro - prepaid_input). This bounds the producer's exposure on prefill and is the mechanism that closes the input-token freeloading attack described in §5.3.

### A.2 settle

```
settle(
  channel_id:      Pubkey,
  commit_message:  CommitMessage,
  commit_signature:Signature,
) -> SettleEvent
```

Verifies the signature against the channel's session_key, validates monotonicity, and enforces the bound prepaid_input ≤ cumulative_paid ≤ deposit. Transfers cumulative_paid micro-USDC to the producer, refunds the remainder to the consumer, and marks the channel as settling. Begins the dispute window. If no commitment exists at settle time (the producer is settling on prefill alone), the program treats cumulative_paid as exactly prepaid_input.

### A.3 dispute

```
dispute(
  channel_id:            Pubkey,
  superseding_message:   CommitMessage,
  superseding_signature: Signature,
) -> DisputeEvent
```

Within the dispute window, accepts a higher-sequence commitment and adjusts the settled amounts accordingly. Reverses any over-refund or under-payment from the initial settle.

### A.4 close

```
close(
  channel_id: Pubkey,
) -> CloseEvent
```

After the dispute window, finalizes the channel and reclaims rent. Either party can call this once the dispute window has elapsed.

## Appendix B — Wire format

### B.1 Session-open negotiation (x402-compatible)

Tap reuses x402's discovery and bootstrap headers rather than introducing parallel headers for the same purpose. A producer advertises Tap support by responding with HTTP 402 and an X-PAYMENT-

REQUIREMENTS header containing a standard x402 payment-requirements payload, with a Tap-specific payment scheme name ("tap.v1.channel") and Tap-specific extra fields:

```
X-PAYMENT-REQUIREMENTS: <base64-encoded JSON>

{
  "scheme":  "tap.v1.channel",
  "network": "solana-mainnet",
  "asset":   "<USDC mint address>",
  "recipient": "<channel program address>",
  "extra": {
    "producer_pubkey": "<base58>",
    "input_price":       <u64 micro-USDC per prompt token>,
    "output_price":      <u64 micro-USDC per generated token>,
    "tokenizer_id":      "<identifier>",
    "input_token_count": <u32, this prompt's token count>,
    "prepaid_input":     <u64, input_token_count × input_price>,
    "max_unpaid":      <u64 micro-USDC>,
    "trailing_buffer": <u32 tokens>,
    "duration_secs":   <u32>,
    "channel_open_url":"<URL>",
    "stream_url":      "<URL>"
  }
}
```

The consumer constructs a standard x402 X-PAYMENT header containing a signed Solana transaction that opens the channel. The producer (or its x402 facilitator) submits the transaction and returns an X-PAYMENT-RESPONSE confirming the channel open, along with a Tap-specific extra field carrying the channel ID:

```
X-PAYMENT-RESPONSE: <base64-encoded JSON>

{
  "tx_hash":    "<base58>",
  "settlement": "confirmed",
  "extra": {
    "channel_id":   "<base58 PDA>",
    "channel_state":"active"
  }
}
```

Clients implementing only x402 will treat the channel-open as an opaque one-shot payment and ignore the extra fields. Clients implementing Tap will parse the extra fields and proceed with streaming. The compatibility is bidirectional: x402 facilitators that do not understand Tap can still settle the channel-open transaction correctly.

## B.2 Streaming commitments

During an active session, commitments flow from consumer to producer in HTTP request headers. The X-TAP-COMMIT header — distinct from x402's X-PAYMENT, since this is in-session signaling rather than payment instruction — carries a base64-encoded payload:

```
X-TAP-COMMIT: <base64-encoded JSON>

{
  "schema":          "tap.v1.commit",
  "channel_id":      "<base58>",
  "sequence":        <u64>,
  "cumulative_paid": <u64 micro-USDC>,
  "tokens_received": <u32>,
  "timestamp_ms":    <u64>,
  "signature":       "<base58>"
}
```

## B.3 Streaming responses

Producer responses use Server-Sent Events (text/event-stream) with one event per token or token batch. Each event includes the token content and a producer-side acknowledgment of the latest received commitment, allowing the consumer to confirm that the producer has registered their most recent payment authorization.

This format is compatible with existing model provider streaming conventions (OpenAI, Anthropic), so consumer SDKs can layer Tap commitment exchange over standard streaming clients with minimal changes.