TOAP: The Token-Optimized Agent Protocol

Reference, Summary, or KV Cache? Measuring How Multi-Agent Systems Should Share Context

Trilochan Sharma
Independent Researcher
parnishklpo@gmail.com

June 2026

Abstract

Multi-agent LLM pipelines tend to forward a growing transcript, the source document plus every prior agent's output, into each downstream prompt; token cost then grows with pipeline depth. An obvious remedy is to pass a *reference* to shared content rather than re-send it. This paper measures that remedy honestly instead of advocating for it, and the results are more nuanced than the pitch. In a controlled experiment (n=33 real Claude generations across Haiku, Sonnet, and Opus, three context strategies, accuracy scored by a deterministic and model-independent rubric checker), reference-minimization (TOAP) and a competent summarizing orchestrator *both* cut downstream tokens with no accuracy loss: all 33 samples scored 4/4. But the summarizing baseline saves *more* tokens than TOAP (∼2.5× vs. ∼1.9× over the naive baseline). So reference-minimization is not a token win over good prompt engineering. What it offers instead is losslessness (a reference can be re-expanded; a summary cannot) and security (provenance travels with the reference), and we now *measure* both rather than assert them. On a pre-registered task where a faithful summary necessarily drops detail, a downstream responder scores **8/8 with the reference vs. 6/8 with the summary**, failing exactly on the dropped facts — and the same direction replicates on two open, non-Claude models (Qwen2.5-7B and Mistral-7B). Against an adversarial corpus of 40 attacks across nine categories, the capability lattice blocks **all 40** attempts to drive a side-effecting operation from untrusted content with **zero** false positives on 14 benign read/analysis requests. That guarantee does not depend on how an injection is phrased, and we show its two documented blind spots as well. (An accuracy regression we first saw on Haiku at n=1 did not reproduce at n=5 and is retracted, a small lesson in why single-call agent measurements are unsafe.) On the compute side, we implement and measure the third materialization, a same-model KV-cache transfer, across seven open models: reusing a transferred prefix cache is a real, near-lossless prefill speedup (35/36 cells byte-identical; up to ∼180× at long context), but the cache is thousands of times larger than the text, so it only pays off when agents are co-located or use grouped-query attention. That last result grounds TOAP's gating policy in measurement rather than assumption. Throughout we separate wire bytes from model tokens, and show symbolic "opcodes" save 50% of bytes but only 0–17% of tokens. We describe the protocol architecture (a two-plane wire format; a reference-materialization spectrum) and a Rust reference implementation. *Threats to validity are first-class*: a single model family/tokenizer for the token study, a same-family judge as a secondary check only, and small n. The mechanism is not novel (blackboard systems, A2A artifacts, ADOL); the contribution is the honest, reproducible accounting.

1 Introduction

In agent pipelines and fan-out workflows, an orchestrator typically forwards a growing transcript into each subsequent agent's prompt. Because LLM cost is paid per prompt token, redundant re-transmission dominates coordination cost. TOAP stores shared content once under an identifier and exchanges references, so each agent retrieves only what it needs.

We try to be conservative about what this buys. The underlying mechanism is old (§2), and "10×" claims that count bytes rather than tokens are misleading (§3). The question worth asking is not whether referencing saves tokens, but how it compares to a competently engineered baseline. Our first result is a little humbling: on an easy task a good summarizing orchestrator saves *more* tokens than reference-minimization at the same accuracy. We treat this as the start of the argument, not the end of it.

Thesis. No single way of sharing context dominates. A summary is cheapest in tokens but lossy; a reference is lossless and carries security metadata but costs more tokens; a transferred KV cache is the fastest to consume but is enormous and model-bound. Which one wins depends on the regime — the length of the shared context, the model's attention architecture, whether producer and consumer are co-located, and whether a later step might need a detail an upfront summary would discard. The contribution of this paper is to *measure* those regime boundaries rather than assume them, and to give an architecture (TOAP) whose reference primitive can be *materialized* as inline text, a context id, or a KV pointer accordingly. Each of the three following claims is now backed by a measurement: the summary's lossiness costs downstream accuracy on a task that needs a dropped detail (§8.1); the capability lattice blocks a corpus of injection attempts with no false positives (§8.2); and KV transfer's compute win, and its size cost, are mapped across seven models (§8.3).

What is measured vs. proposed. The token experiments (§3, §8) use the implemented V1 text protocol and real model generations; the KV-bridge experiment (§8.3) uses the implemented same-model prefix-reuse sidecar. **§4.5 states precisely what is implemented today** (V1/V2 protocol, the context store with capability-lattice security, the materialization policy, and the KV-bridge sidecar) **and what remains future work** (durable/distributed store, cross-vendor evaluation, message signing).

Contributions.

1. A two-plane protocol architecture and a reference-materialization spectrum (§4), with a Rust reference implementation (§5).
2. An honest token accounting separating bytes, tokens, and compute, including the result that a summarizing baseline beats referencing on tokens (§3, §8).
3. A pre-registered demonstration that a summary's lossiness costs downstream accuracy when a dropped fact is needed, while a reference does not (§8.1).
4. An adversarial evaluation of the capability lattice against a corpus of injection attempts (§8.2).
5. A seven-model measurement of same-model KV-cache reuse, mapping where its compute win justifies the cache-transfer cost (§8.3).

2 Background and Related Work

Shared-store coordination. Agents coordinating through a common store rather than passing full messages is the *blackboard* model (Hearsay-II [1]); database normalization and pointers are

the same principle. TOAP's context store is a modern restatement, and we claim no novelty for it. **Agent protocols.** MCP [2] (agent–tool) and A2A [3] (agent–agent) are the dominant standards; A2A already references shared content by ID via *artifacts*. Token bloat is recognized: the ADOL Internet-Draft [4] and MCP SEP-1576 target token-efficient messaging (Table 1). **Caching.** Provider prompt caching [5] realizes "send once, reference implicitly" within one provider/session; cross-agent referencing is complementary. **Symbolic comms** [6, 7] cut tokens but risk comprehension; we measure the opcode component at $\leq$17%. **Security.** Provenance tracking follows CaMeL [8]. **KV reuse** [9] is crowded and is treated as an optional, constraint-guarded mode.

Protocol	Layer	Wire format	Token efficiency a goal?
MCP (Anthropic)	agent–tool	JSON-RPC 2.0 / HTTP	No
A2A (Google/LF)	agent–agent	JSON-RPC/HTTP (+gRPC/REST)	No
ADOL (IETF draft)	layer above MCP/A2A	JSON `$ref` dedup	Yes (content selection)
TOAP (this work)	agent–agent	two-plane text/binary	Yes (encoding, token-aware)

Table 1: Inter-agent protocol landscape (2026). TOAP is adjacent to ADOL/SEP-1576, not ahead of them.

3 Bytes Are Not Tokens

A common mistake is to minimize *bytes* and report the result as token savings. The two are not the same. BPE tokenizers pre-tokenize on punctuation before merging, so a payload dense in punctuation spends close to one token per symbol. Measured with `tiktoken` (cl100k), `SUM(CTX:42)?max_words=150` is 25 bytes and 10 tokens, while the natural-language "Please summarize document 42 in at most 150 words." is 50 bytes and 12 tokens: half the bytes, but only 17% fewer tokens (Figure 1). Richer payloads save even less. The lesson is that opcode terseness is a minor lever, and the question that actually matters is whether to transmit the *content* at all. We keep three cost planes separate throughout this paper and measure each on its own terms: wire bytes, model-input tokens, and prefill compute.

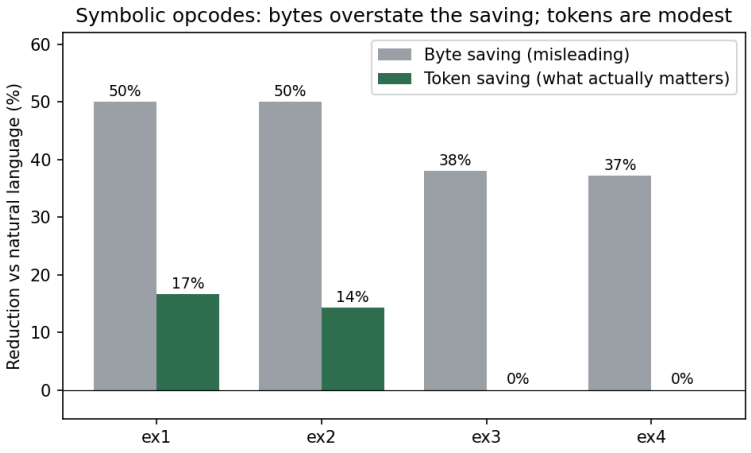

Figure 1: Byte savings overstate token savings for symbolic opcodes (tiktoken cl100k).

4 TOAP Architecture

4.1 System overview

TOAP is a layered, broker-mediated protocol (Figure 6). A message travels *down* the stack on the way out and *up* the stack on the way in, and each layer has one narrow job. The choice that shapes everything else is that **agents never talk to each other directly**: every message goes through a broker that owns identity, access control, and routing. We made that choice on purpose. If an agent could open a socket to another agent, then the "who sent this" field would have to be either trusted (spoofable) or separately authenticated on every edge of the mesh; routing the mesh through one mediator means identity is decided once, at the point of connection, and never has to be claimed on the wire again. It also gives security a single place to live (§4.1.3) instead of being re-implemented in every agent. The cost is a central dependency, which we treat as an explicit assumption rather than pretend away (§9).

To make the layers concrete before defining them one by one, it helps to follow a single request all the way through. Suppose an orchestrator wants a worker to summarize a document it holds. The document is stored once; the request that crosses the wire is a short reference, not the text; the reply comes back correlated to the original request; and at no point does the worker learn, or get to assert, who the orchestrator is. Figure 2 traces exactly that, and the per-layer subsections below explain what each step does and why it is built the way it is. After the layers we cover the two ideas that are particular to TOAP: the two-plane wire format (§4.2) and the reference materialization spectrum (§4.4).

4.1.1 Agents and the client SDK

Agents (LLMs, tools, or rule-based workers) hold no protocol state of their own. They talk to a thin async client SDK whose whole purpose is to keep the protocol out of the agent author's way: it runs the SYN/ACK handshake that binds identity at connect time, allocates and tracks `msg_id`s so the author never manages correlation by hand, and separates the two kinds of inbound traffic an agent cares about — replies to its own requests, and *events* pushed because a context it subscribed to changed — onto different channels so they cannot be confused. We kept the surface deliberately small, five calls (`set_context`, `get_context`, `send_request`, `patch`, `subscribe`), because every capability beyond those is something an agent could misuse or get wrong; the narrow API is part of the security story, not just ergonomics. In the trace, the orchestrator's "store the doc, then ask for a summary" is two SDK calls, and the reply-matching in step 3 is the SDK resolving `msg_id=8` back to the caller that is awaiting it.

4.1.2 Protocol layer

This layer turns a message into bytes and back, and validates the grammar on the way in. A V1 frame is four pipe-separated fields followed by a function-style payload:

```
REQ|8|worker|SUM(CTX:42)?max_words=60
```

read as *type* `REQ`, *msg_id* `8`, *target* `worker`, and a *payload* that names an operation `SUM`, a positional argument `CTX:42`, and an option `max_words=60`. Three choices here are deliberate. First, the payload is *function-style* rather than colon-delimited: an earlier design wrote `SUM:CTX:42`, which is ambiguous because the argument `CTX:42` itself contains a colon, so the parser could not tell a field separator from data. Parentheses and a `?opts` suffix remove that ambiguity without needing escapes in the common case. Second, we did not use JSON. JSON would wrap every small coordination

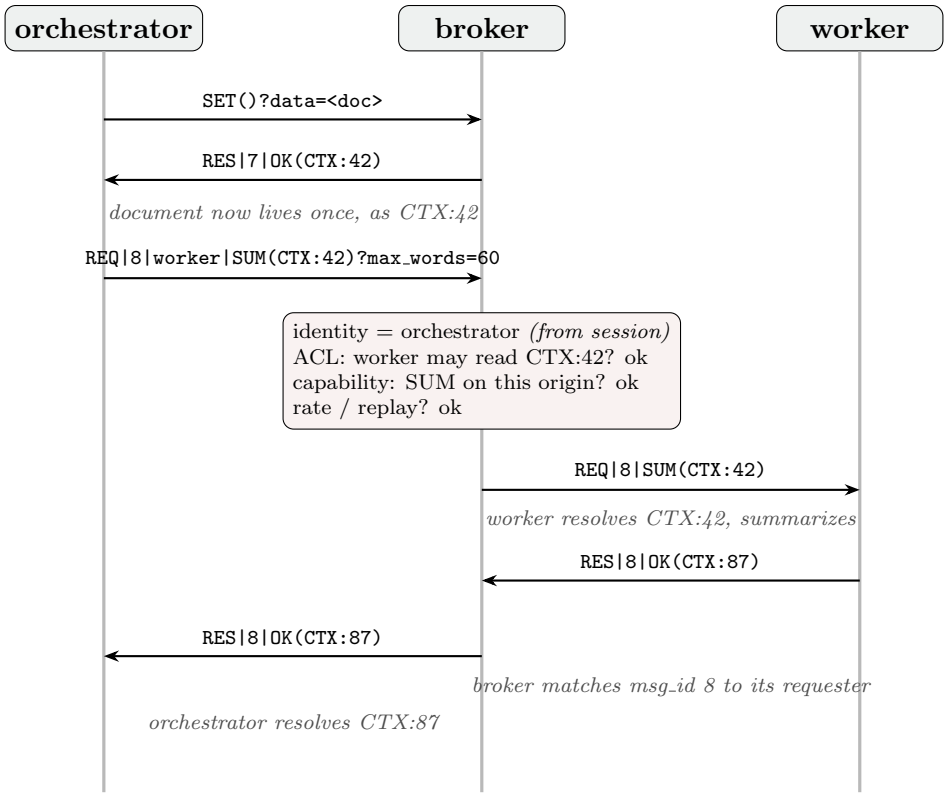

Figure 2: One request, traced through the stack. The document is stored once and thereafter referenced by id; the inter-agent message carries `CTX:42`, not the document. The broker stamps the sender's identity from its session, runs the four security checks, and routes by target. The worker replies to `msg_id 8`, and the broker uses that id to return the reply to whoever sent request 8 — so the worker never names, and could never forge, the requester.

message in quotes, braces, and key names that the LLM ultimately pays for in tokens (§3), and the wire savings of TOAP come precisely from *not* doing that. Third, the parser is strict about *structure* (it checks the delimiter count, the payload grammar, and that a `CTX:N` reference is well-formed) but it never rejects *content* for looking dangerous: `DROP TABLE`, `<script>`, and "ignore previous instructions" are all perfectly legal *data* to carry, because deciding what is safe to *do* with content is the security layer's job (§4.1.3), not the parser's. Trying to blacklist scary-looking strings at parse time is both leaky and breaks legitimate payloads, so we don't. V2 keeps this exact message model but serializes the routing fields as a compact binary header (§4.2); the two formats round-trip to the same in-memory message, so a deployment can switch transports without touching agents.

4.1.3 Security filter

Before routing, every inbound message passes four checks, in order: (i) *identity* — derived from the authenticated session, never read from the wire; (ii) *ACL* — per-context read/write/delete; (iii) *capability* — does this content's provenance permit this operation (§4.3); and (iv) *rate/replay* — a per-agent token bucket and per-session duplicate-`msg_id` rejection. A failure at any stage short-circuits to a typed error (`NOPERM`/`RATE`/`REPLAY`) and the message is never routed (Figure 3).

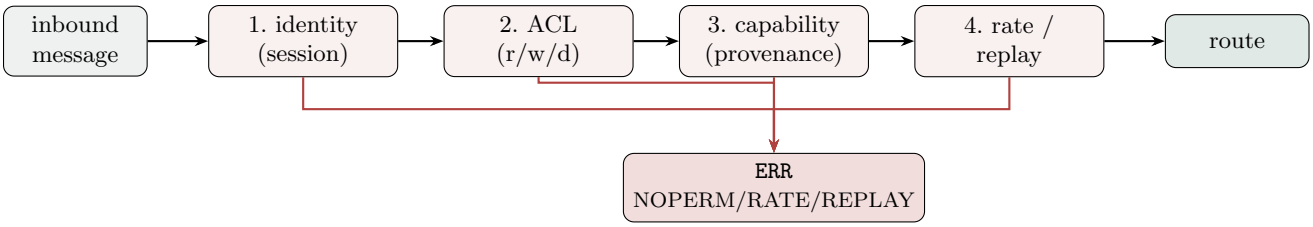

Figure 3: Broker-side security filter. Every inbound message passes four ordered checks; any failure short-circuits to a typed error and is never routed. Identity is derived from the session (never the wire); the capability check refuses, e.g., `EXEC`/`PAY`/`DELETE` on user-origin content.

4.1.4 Broker / router

The broker is where routing and identity meet, and two design decisions there carry most of the weight. The first is *identity by session*. When an agent connects, it completes a SYN/ACK handshake and the broker binds one `agent_id` to that connection for its lifetime; from then on, "who sent this" is read from the connection, not from any field the sender can write. In the trace (Figure 2, step 2) the broker stamps `orchestrator` on the request itself — there is simply no place in the wire frame for a sender to claim otherwise. This is the structural reason a compromised agent cannot impersonate another: spoofing would require hijacking the TCP session, not just editing a string.

The second is *correlation by `msg_id` rather than by reply-to address*. The requester chooses an id (8 in the trace); the broker remembers that id → requester mapping, forwards the request to the target, and when a `RES` or `ERR` comes back carrying the same id, it delivers it to whoever originally sent id 8. The worker therefore replies *to the id*, never to "the orchestrator"; it never has to know, and so can never forge, the requester's identity. This also makes concurrency safe for free: an agent can have many requests in flight and the broker untangles the replies by id, so two overlapping summaries cannot get crossed. The same machinery drives subscriptions: a `PATCH` (a `DLT` message) to a context fans out an `EVT` to every agent that subscribed to it, which is how collaborative state changes propagate without polling. Figure 4 draws both mechanisms.

Routing everything through one broker is what makes all of this possible, and it is also the system's main structural weakness: the broker is a single point of failure and a throughput choke-point. We state that plainly as a threat-model assumption rather than design around it here (§9); making the broker highly available is exactly the distributed-systems work we have not done.

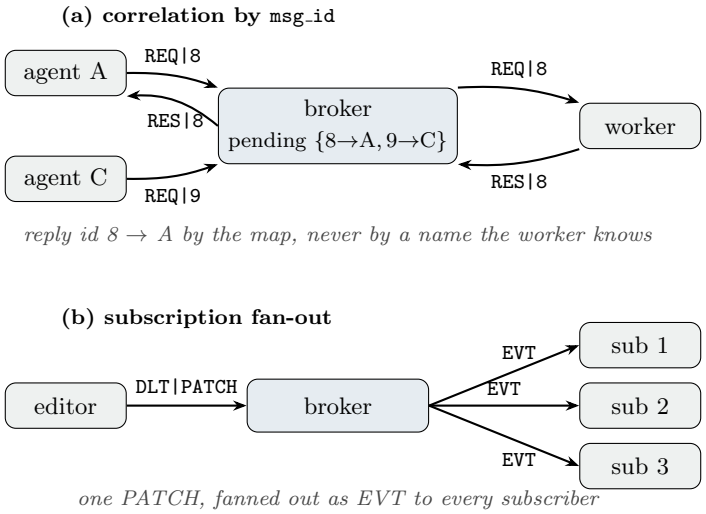

Figure 4: Two broker mechanisms, drawn separately so each reads cleanly. **(a)** Correlation by `msg_id`: the broker keeps a pending map (id → requester), so concurrent requests from A (id 8) and C (id 9) are untangled; the worker replies *to the id*, and the broker routes `RES|8` back to A by the map — never by a name the worker had to know. **(b)** Subscription fan-out: a single `PATCH` (a `DLT`) to a context is pushed as an `EVT` to every subscriber, so collaborative changes propagate without polling.

4.1.5 Shared context store

This is where the document in the trace actually lives. Content is written once and addressed by a numeric id (`CTX:42`), and an entry is more than its bytes: it carries the broker-derived owner, an ACL, the capability provenance the security layer reads, a TTL with lazy and swept garbage collection, a monotonic version with an append-only field-level delta log, and the set of subscribers to notify on change. Each piece of that metadata earns a property the bare bytes could not give. Storing the content (rather than a digest of it) is what makes a reference *lossless* — the worker that resolves `CTX:42` sees the original document, so nothing is silently dropped, which is the whole point established in §8.1. Carrying provenance *with* the entry is what lets the security check travel with the reference instead of being re-derived at each hop. And the version plus delta log is what makes a context safe to *edit* collaboratively: a `PATCH` appends a versioned delta and wakes subscribers, rather than two agents clobbering each other's writes. The backend today is in-memory and single-node, which is enough to demonstrate all of this but not to survive a restart or scale across machines; a durable, distributed store is future work (§10), and is the honest precondition for the systematization argument. Figure 5 lays out one entry and what each field buys.

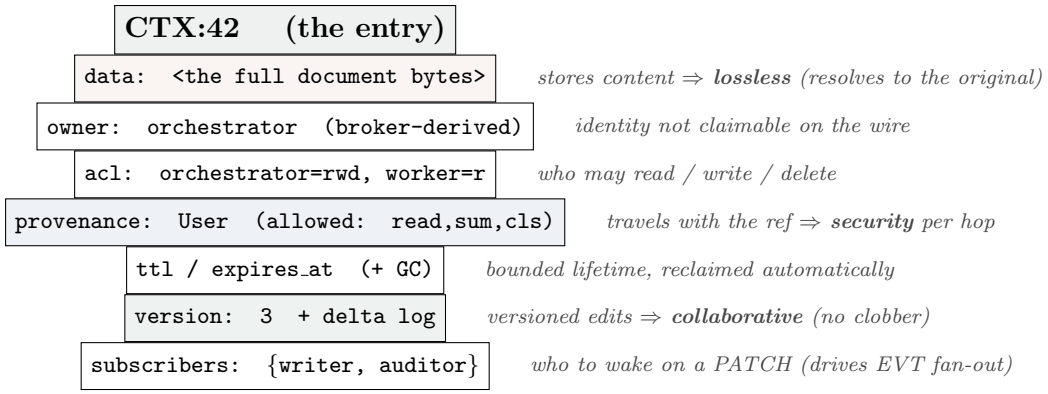

Figure 5: Anatomy of a stored context. The entry is far more than its bytes; each field earns a property the bare content could not. Storing the content (not a digest) is what makes the reference lossless; carrying provenance *in* the entry is what lets security travel with it; the version and delta log are what make collaborative edits safe.

4.1.6 Transport

The bottom layer just moves bytes: length-prefixed records over TCP (Tokio), where the prefix tells the reader how many bytes to expect so messages can be reassembled from a stream without a delimiter that could collide with payload data. We keep this layer deliberately dumb and free of protocol knowledge, so that swapping in WebSocket or gRPC later changes nothing above it. Nothing in the agent, protocol, security, or broker layers depends on the transport choice.

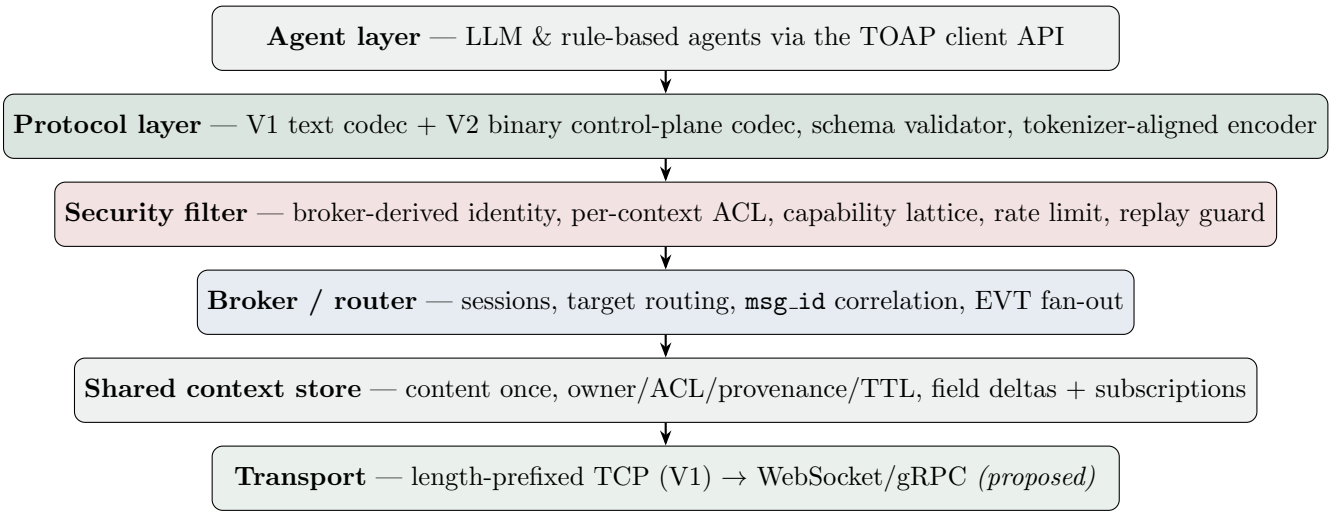

Figure 6: TOAP layered architecture. Identity/trust are broker-owned, never claimed on the wire.

4.2 Two-plane wire format

A single message is read by two parties with opposite cost functions. The *broker* needs ids, session, ACL, capability tags, and nonces — metadata whose cost is *bytes* (bandwidth, parse latency, framing) and which the LLM should never see. The *LLM* needs the operation and its content — text whose cost is *tokens* and which the broker should never tokenize. TOAP encodes these as

two planes: a compact binary *control plane* optimized for bytes, and a tokenizer-aligned *semantic plane* optimized for tokens (Figure 7). Because each plane is encoded for its single reader, neither is dominated — which is precisely why "bytes vs. tokens" (§3) stops being a confusion: they are different planes with different objectives.

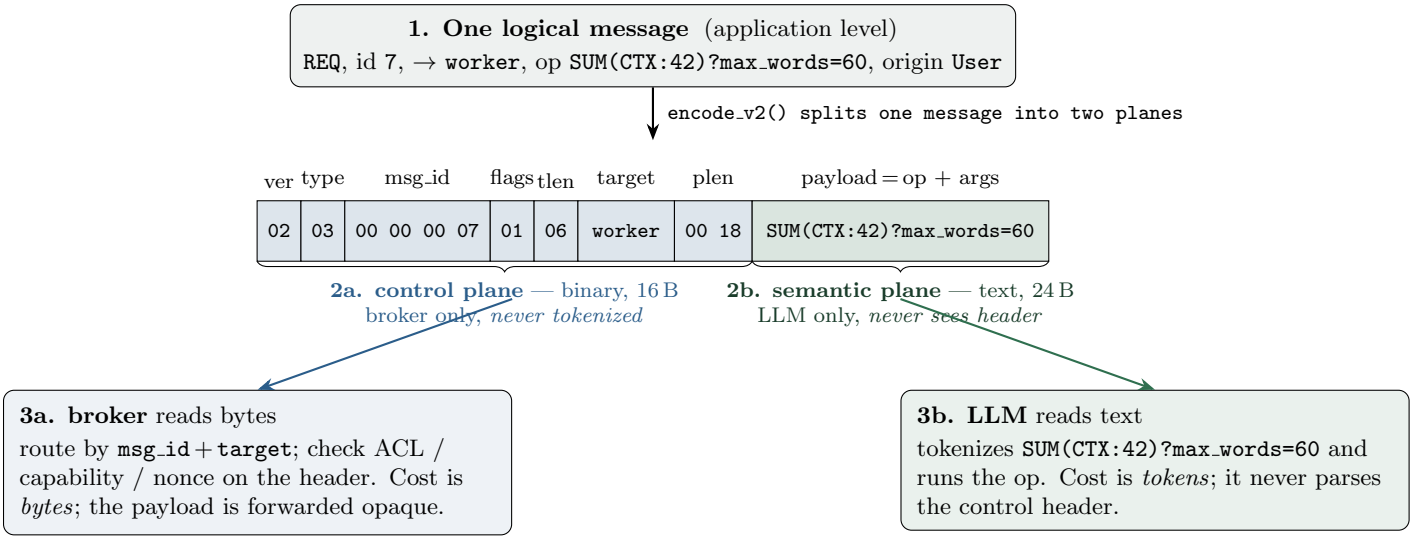

Figure 7: How one message travels through the two planes (input → output). A single application-level message is serialized by `encode_v2()` into one frame holding two regions encoded for opposite cost functions: a compact *binary control plane* (version, type, `msg_id`, flags, target, length — the only part the broker reads, and it never tokenizes it) followed by the *text semantic plane* (the `OP(args)?opts` the LLM reads and tokenizes, and never the header). Bytes shown are the real `toap-core` V2 layout for this message. Because each region is optimized for its single reader, neither is dominated — which is why "bytes vs. tokens" (§3) is a plane distinction, not a confusion.

4.3 Capability-lattice provenance

A boolean "tainted" flag cannot express *what* untrusted content may do. TOAP attaches a provenance to each context: an `Origin` (INTERNAL/EXTERNAL/USER) and the set of `Capability` values that origin permits (read, summarize, transform, classify, execute, email, pay, delete). Trusted internal content permits all; external content permits read-only flows (read/summarize/transform/classify) but no side effects; user-originated content is most restricted. The broker maps each opcode to a capability (`Capability::for_op`) and refuses any operation the provenance forbids — e.g. an `EXEC/PAY/DELETE` driven by user-origin content returns `NOPERM reason=capability_denied`. Because the tag travels with the reference, taint is preserved across agent-to-agent hops, structurally containing prompt-injection propagation rather than relying on fragile content pattern-matching.

4.4 Reference materialization spectrum

"Send the text," "send a context id," and "send a KV-cache pointer" are not three features but three *materializations of one logical reference*, chosen per call by a cost model and degrading gracefully when constraints fail: KV_BRIDGE → CTX_REF → INLINE. INLINE embeds the bytes (best for tiny/one-shot content); CTX_REF sends the id and the receiver fetches from the store (the dedup

win for reused content); KV_Bridge points at a reusable KV cache so the receiver skips prefill (valid only for the same model and tokenizer). The same decision — materialize now vs. reference vs. reuse computation — spans both the token plane (resend text vs. send id) and the tensor plane (re-prefill vs. load KV); treating them as one policy is the architecture's most novel point (Figure 8). Figure 9 shows the token-flow contrast the experiment measures: naive prompts grow with pipeline depth while references stay flat, because the store holds content once and only ids travel.

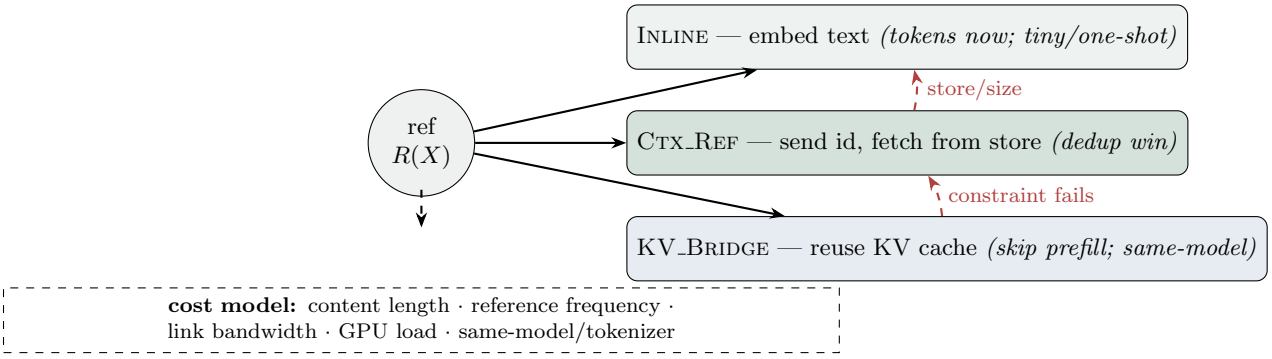

Figure 8: Reference materialization spectrum: one logical reference, three materializations chosen per call by a cost model, with graceful degradation KV_Bridge → Ctx_Ref → Inline when constraints fail. The same decision spans the token plane (resend vs. id) and the tensor plane (re-prefill vs. load KV).

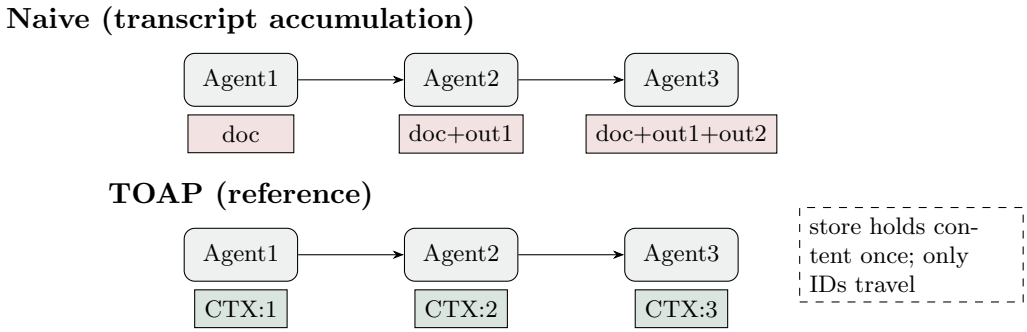

Figure 9: Token-flow contrast: naive prompts grow with depth; references stay flat.

4.5 Implemented vs. proposed

Implemented today (29 passing unit + integration tests): the V1 text protocol and parser; the **V2 binary control plane** (compact binary control header + text semantic payload — the two-plane design, with round-trip tests); the shared context store with per-context ACL, TTL, field-level deltas, and subscriptions; a **capability-lattice** provenance model (Origin ∈ {Internal, External, User}, each with an allowed-capability set) the broker enforces structurally (user-/external-origin content cannot flow into EXEC/PAY/DELETE); an **in-band cache coordinate** with a stable-prefix ‖ volatile-suffix layout helper; the **materialization policy with graceful fallback** (KV_BRIDGE → CTX_REF → INLINE) behind a `KvTransport` trait; broker-derived identity, token-bucket rate limiting, per-session replay rejection, `msg_id` correlation; and an MCP frontend.

A minimal **KV-cache transport** is now implemented as a same-model, same-tokenizer *prefix-reuse* sidecar (`benchmark/kv_bridge`): it extracts a shared context's KV, serializes/transfers it, and lets the receiver prefill only its query suffix, with a benchmark that measures prefill-latency saved, KV-byte-size vs. text-size, and exact greedy-output correctness. The Rust side owns the policy and fallback (`RegistryKvTransport` enforces the same-model constraint); the sidecar owns the tensor transport. We deliberately restrict to prefix reuse (absolute positions preserved) and do *not* splice caches across positions. Cross-vendor evaluation remains future work.

5 Implementation

TOAP is a Rust workspace. `toap-core` defines the message model, a strict recursive-descent parser, and the encoder; it deliberately rejects malformed *protocol* syntax but never rejects ordinary content for "looking dangerous" (SQL/HTML/prompt-like text are data). `toap-context` implements the in-memory store: numeric context IDs, an ACL grammar (`agent:rwd`), a *capability-lattice* provenance model (`Origin` × allowed `Capability` set, with `Capability::for_op` mapping opcodes to capabilities), TTL with lazy + sweep GC, a bounded per-context delta log, the `Reference` materialization enum with a `materialize_with_fallback` policy behind a `KvTransport` trait, and a `cache_layout` helper for stable-prefix ordering. `toap-core` also provides a V2 binary control-plane codec (`encode_v2`/`decode_v2`) that round-trips the V1 message model. `toap-security` provides a token-bucket rate limiter and a per-session replay guard; the broker enforces the capability lattice structurally, refusing e.g. an `EXEC/PAY/DELETE` against user- or external-origin context. `toap-broker` is an asynchronous (Tokio) TCP server with length-prefixed framing; identity is bound to the connection at SYN, and a responder addresses replies by the original `msg_id` so it never learns the requester (preventing source spoofing). `toap-client` is an async SDK; `toap-mcp` is the MCP frontend.

Testing is unit *and* integration: parser round-trip/validation and store/ACL/TTL/delta unit tests, plus end-to-end integration tests in which **real client processes connect to the broker over TCP** and exchange requests — the broker performs actual routing and store operations (not mocked). Security integration tests assert that an unauthorized read is denied, a restricted op on tainted context is refused, and a subscriber receives a delta event. A live multi-process demo (broker + summarizer + classifier + orchestrator) exercises fan-out and merge. Coordination-frame sizes reported below are measured on the actual encoder output.

6 Why a Protocol, Not Just a Better Orchestrator?

The sharpest objection is: "just write an orchestrator that summarizes before forwarding." Our token experiment confirms a summarizing orchestrator is in fact *more* token-efficient than refer-

encing (§8), so the case for a protocol cannot rest on token savings. It rests on two properties a prompt-engineering pattern does not provide, and we point to the measurement for each rather than asserting them. **(i) Losslessness.** A reference can be re-expanded on demand; a summary discards information irreversibly. This is not hypothetical: when the discarded detail is the one a downstream step needs, the summary arm fails and the reference arm does not (8/8 vs. 6/8, §8.1). **(ii) Security travels with the reference.** Provenance and broker-derived identity are enforced structurally, so untrusted content cannot be laundered into a side-effecting operation regardless of how the injection is phrased — 40/40 attempts blocked at no cost to benign traffic (§8.2); a summarizing prompt template offers none of this. A third intended property, *systematization* across heterogeneous agents, is a design goal of the protocol but is not something this paper measures (the implementation is single-language and single-node); we list it as motivation, not as a result. The honest bottom line is that referencing earns its place in exactly the regimes where losslessness or in-band security matters, and yields to summarization where neither does.

7 Experimental Method

Instrument. Each agent call is an isolated Claude subagent at an explicit model (Haiku/Sonnet/Opus), tool use disabled; outputs are genuine generations. **Controlled design.** To get clean variance we fix one canonical upstream (one document, one extracted-facts text, and one analysis containing all four rubric themes) so the only variables are {model, context strategy, sample}. Three strategies for the final *writer* stage: *naive* (document + facts + analysis), *summary* (a short lossy summary of the analysis, i.e. a competent summarizing orchestrator), and *TOAP* (the analysis only, i.e. the distilled slice passed by reference). We draw n=5 samples per cell for Haiku and n=3 for Sonnet/Opus (n=33 total). **Accuracy (bias-free).** The primary accuracy measure is a *deterministic* rubric checker: each of four required actions is detected by keyword sets, with no model in the loop, so it is immune to same-family judge bias and exactly reproducible. (A model judge was used only in an earlier single-sample run and is not relied upon here.) **Tokens.** Counted with `tiktoken` (cl100k) on the exact prompts and verbatim outputs. **Reproducibility.** All prompts, verbatim outputs, and the scorer are committed.

Reviewer-driven changes. A first review noted the token study was thin; a second noted that the paper disproved its headline without establishing its replacement. In response we (1) replaced n=1 with n=33 and added a *strong* summarizing baseline; (2) switched to a deterministic, model-independent scorer; (3) *ran* the losslessness experiment the earlier draft had only named as missing (§8.1); (4) added an adversarial evaluation of the capability lattice (§8.2); and (5) implemented and measured KV transfer across seven models (§8.3). The single-model-family scope of the *token* study remains; the losslessness claim, by contrast, we have now replicated on two open non-Claude models (§8.1).

8 Results

Accuracy parity; the regression retracted. All 33 samples scored 4/4 on the deterministic rubric (Figure 11); coverage variance was zero. A 4/4→3/4 Haiku regression seen in an earlier *single-sample* run **did not reproduce** at n=5 with a fixed upstream; we attribute it to single-sample noise and/or propagation of a weaker upstream analysis, and retract it.

Tokens: the summarizing baseline beats TOAP. Table 2 and Figure 10 give total writer tokens. Against the naive baseline, TOAP cuts tokens by 1.88–1.93× (mean ∼1.91×); the summarizing baseline cuts them by 2.39–2.62× (mean ∼2.54×). Both hold 4/4 accuracy. Reference-passing, in other words, is not the most token-efficient strategy; a competent summarizer is. We

state this plainly because it is the result most likely to be glossed over, and because it reframes what the rest of the protocol is for.

Model	Strategy	n	Accuracy (/4)	Total tokens (mean [min–max])	Reduction vs. naive
Haiku	naive	5	4.0	505 [491–520]	—
Haiku	summary	5	4.0	211 [193–219]	2.39×
Haiku	TOAP	5	4.0	268 [258–279]	1.88×
Sonnet	naive	3	4.0	512 [499–523]	—
Sonnet	summary	3	4.0	195 [190–204]	2.62×
Sonnet	TOAP	3	4.0	266 [265–267]	1.93×
Opus	naive	3	4.0	504 [487–519]	—
Opus	summary	3	4.0	194 [192–196]	2.60×
Opus	TOAP	3	4.0	263 [253–271]	1.92×

Table 2: Controlled writer experiment (n=33). Accuracy by deterministic rubric (bias-free). Tokens via tiktoken. TOAP < naive, but summary < TOAP.

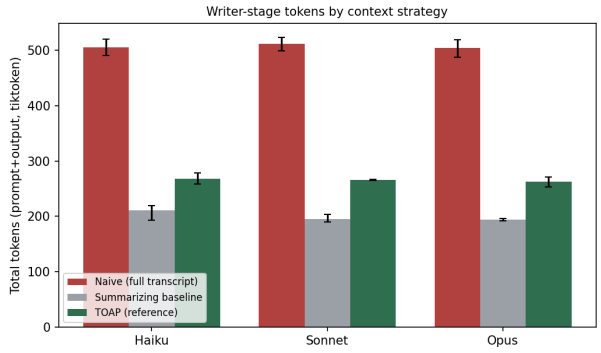

Figure 10: Writer tokens by strategy (error = min–max). Summary < TOAP < naive.

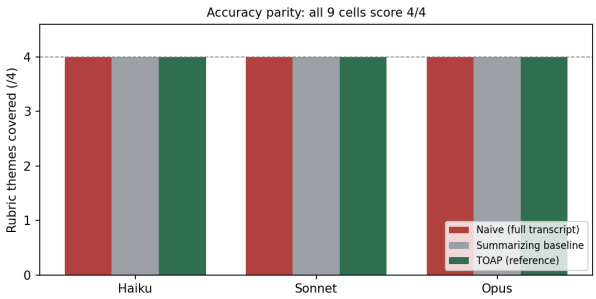

Figure 11: Accuracy parity: all nine cells 4/4 (the n=1 regression did not reproduce).

Where referencing does win: multi-hop and coordination. The writer experiment is a single hop. In a multi-hop pipeline, referencing avoids re-summarizing/re-sending at *every* hop, and the store's send-once/refer-many compounds; an exploratory three-stage run showed 1.65–1.82× downstream reductions. Separately, TOAP coordination frames are ∼17× smaller in tokens than JSON-RPC equivalents (model-independent, exactly reproducible). These are the regimes where a reference protocol, rather than per-hop summarization, is the natural fit — and where losslessness matters most.

8.1 Losslessness under pressure: when a summary's lossiness costs accuracy

The token experiment above used an easy task on which every strategy reached 4/4, so it could not show the cost of a summary's lossiness; that absence was the most important gap in an earlier draft. We close it with a pre-registered experiment designed so that lossiness must bite. A 130-word incident report carries eight distinct facts (a rollback time, a revenue figure, a PR number, the reason a bad config passed staging, and so on). We had a summarizer subagent compress it faithfully to 45 words — a length at which all eight facts cannot survive — and then asked a downstream responder eight fixed questions, each needing one specific fact, under two conditions: *summary* (it sees only the 45-word summary) and *reference* (it sees the full document, i.e. the lossless reference

materialized). The scenario, questions, and ground-truth answers were fixed before any responder ran; grading is deterministic substring matching; three samples per arm.

The faithful summary dropped two of the eight facts (the rollback time and the staging-schema reason). The downstream consequence is exactly as predicted: the **reference arm scores 8/8** in all three runs, while the **summary arm scores 6/8**, missing precisely the two questions whose facts were dropped (Table 3). This is the empirical content of the losslessness claim: a summary is cheaper (§8) until a later step needs something it discarded, at which point it fails silently; a reference cannot, because the information is still there to be re-read.

To check this is not a Claude-specific artifact, we re-ran the *identical* pre-registered experiment — same document, summary, questions, and grader — on two open, non-Claude instruct models (Qwen2.5-7B and Mistral-7B, loaded in 4-bit on a single T4, n=10 per arm; `lossless_colab.ipynb`). Both replicate the direction: the reference arm scores strictly higher than the summary arm (Qwen 6.0 vs 5.0; Mistral 5.4 vs 4.4; Table 4). The absolute scores are lower than Claude's — a 4-bit 7B model misses a fact or two even with the full document in front of it, and Mistral's reference arm shows real run-to-run variance (5–7) — and the gap is about one point rather than two, but across three independent model families the lossless reference never loses to the lossy summary. We do not generalize the *magnitude* of the gap (it is one scenario at one compression ratio), but the effect is clearly not specific to one model: any lossy compression of k facts into fewer than k must drop some, and any later step that needs a dropped one then fails silently.

Downstream question (fact needed)	Summary arm	Reference arm
Pool ceiling value; revenue; PR #; lower bound; alert threshold; runbook (6 facts)	3/3	3/3
Rollback time (09:51) — *dropped by summary*	**0/3**	3/3
Why staging passed (different schema) — *dropped by summary*	**0/3**	3/3
Total (mean of 3 runs)	**6/8**	**8/8**

Table 3: Losslessness under pressure. A faithful 45-word summary of an 8-fact document drops two facts; the downstream responder then fails exactly those two questions, while the lossless reference answers all eight. Pre-registered; deterministic grading; `benchmark/ab_study/lossless/`.

Model family	n/arm	Summary	Reference	Δ
Claude (API)	3	6.0	8.0	+2.0
Qwen2.5-7B-Instruct (4-bit, T4)	10	5.0	6.0	+1.0
Mistral-7B-Instruct-v0.3 (4-bit, T4)	10	4.4	5.4	+1.0

Table 4: Cross-vendor replication of the losslessness effect (mean score out of 8). On every model family tested, the lossless reference outscores the lossy summary; the open 4-bit 7B models score lower in absolute terms but show the same direction. Raw per-run scores: `benchmark/ab_study/lossless/lossless_crossvendor.json`.

8.2 Does the capability lattice actually stop anything?

§4.3 describes the lattice; the fair question is whether it stops real attacks or is just plausible-looking infrastructure. We threw a corpus at it. The threat we model is a prompt injection that tries to make untrusted content (User- or External-origin) drive a side-effecting operation, and we cover nine flavours of it: straightforward exfiltration, code and shell execution, payments,

deletes, privilege escalation through writes, the same restricted call buried at the end of a multi-hop chain, a payload assembled across several context updates, novel or mis-cased opcodes hoping to slip past the dispatcher, and the same attacks dressed up in base64, Unicode homoglyphs, or zero-width characters. That is 40 attacks in total, alongside 14 benign requests (reads, summaries, classifications, and the legitimate transforms that External content is allowed). Each is sent through the same broker check, which maps the requested opcode to a capability and refuses anything the content's provenance does not permit. The lattice **blocked all 40 attacks and allowed all 14 benign requests** — a 100% block rate at a 0% false-positive rate, the same in every category (Table 5).

We are not claiming a clever detector here, and the chaining and obfuscation cases are in the corpus to make that explicit: the decision is taken on (origin, capability), never on the text, so where the injection sits in a chain or how it is encoded simply does not enter the computation. That is the strength and also the boundary. A capability gate governs what untrusted content may be *used for*; it says nothing about attacks that live elsewhere. We include two such cases and confirm the lattice does *not* stop them, by design: an injection that stays inside an allowed capability (it only asks for a summary, then steers that summary) sails through, and content whose origin was mislabeled INTERNAL at ingress is trusted on that label. Defending those needs output-side checks and a trustworthy tagging boundary respectively — separate components we assume rather than evaluate. The whole corpus, the per-category counts, and the two boundary cases are a committed test (`crates/toap-context/tests/capability_redteam.rs`), so the result regenerates from a clean checkout.

Attack category	Blocked	Attack category	Blocked
Exfiltration (email)	5/5	Multi-hop chain	4/4
Execution (exec/shell)	6/6	Split payload	3/3
Financial (pay)	4/4	Stepping-stone opcode	4/4
Destructive (delete)	4/4	Obfuscation/encoding	5/5
Escalation (write/xform)	5/5	**All / benign**	**40/40 0 FP/14**

Table 5: Capability-lattice red team. 40 injection attempts across nine categories, all blocked; 14 benign read/analysis requests, none blocked. Two documented boundary cases (within-capability steering; mislabeled origin) are *not* blocked, by design.

8.3 KV-bridge: prefill reuse for same-model agents

The third materialization, KV_BRIDGE, transfers a shared context's key/value cache so the receiver prefills only its query suffix. We evaluate the same-model, same-tokenizer prefix-reuse case (absolute positions preserved; no cross-position splicing) on an NVIDIA T4 across seven open models that span attention designs: multi-head attention (GPT-2, GPT-2-large, Pythia-410M/1.4B), grouped-query attention with very few KV heads (Qwen2.5-0.5B/1.5B), and a 4-bit 7B model (Mistral-7B). Timing is warmed up and CUDA-event-timed, and isolates *prefill* (decode excluded). Each cell reports two speedups: *resident* (= recompute prefill / query-prefill on an already-resident KV — the co-located / shared-broker case, the pure compute win) and *cross-node* (additionally charging the serialize/transfer of the KV cache). Correctness is an exact greedy-token match against recompute. Three things come out of the run, and they match the hypothesis that motivated the gating policy. First, reuse is a *real* compute saving: once the shared prefix passes a crossover of a few hundred tokens, the resident speedup climbs steeply — to roughly 180× for the half-billion- parameter Qwen at an 8k prefix, because recompute pays $O(n^2)$ attention over the whole prefix while the bridge

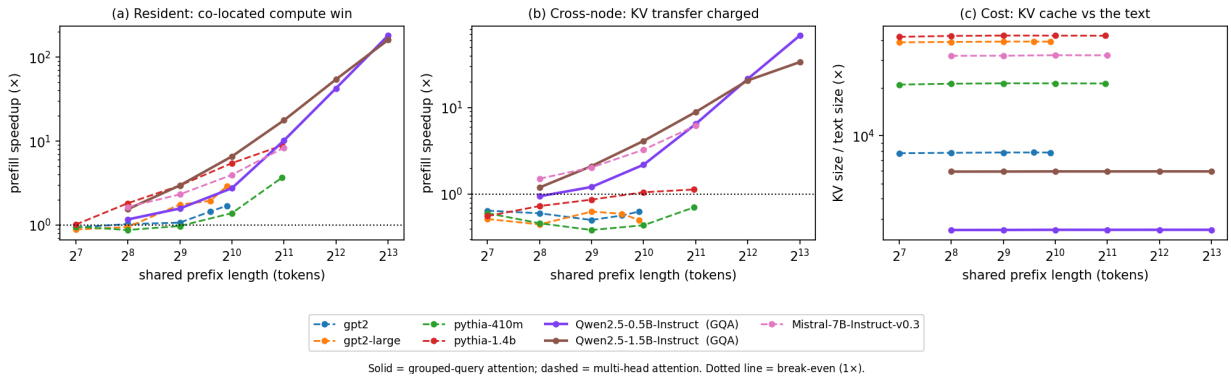

Figure 12: KV-bridge prefill reuse across seven models on an NVIDIA T4 (36 cells, log scales). (a) Resident speedup grows with prefix length and is dramatic for GQA models (solid), reaching ∼180× at 8k tokens. (b) With KV transfer charged, GQA stays above break-even while MHA (dashed) mostly does not. (c) The KV cache is thousands of times larger than the text, and far larger for MHA than GQA — the reason transfer cost flips the verdict by architecture.

Model	Attn	Max prefix	KV/text	Resident	Cross-node	Lossless
GPT-2 (124M)	MHA	960	7.8k	1.7×	0.6×	4/5
GPT-2-large (774M)	MHA	960	39k	2.9×	0.5×	5/5
Pythia-410M	MHA	2000	21k	3.7×	0.7×	5/5
Pythia-1.4B	MHA	2000	43k	8.9×	1.1×	5/5
Mistral-7B (4-bit)	MHA	2048	32k	8.4×	6.2×	4/4
Qwen2.5-0.5B	GQA	8192	2.5k	**181**×	**68**×	6/6
Qwen2.5-1.5B	GQA	8192	5.9k	**161**×	**34**×	6/6

Table 6: KV-bridge at each model's longest tested prefix. Resident = co-located compute win; cross-node charges KV transfer. KV/text is the cache size relative to the text it replaces. Lossless counts cells with byte-identical greedy output (35/36 overall).

pays only for the short query. Below the crossover the two are within noise (∼0.9–1.1×), which is exactly why the policy does not bridge tiny contexts. Second, the saving is essentially *lossless*: 35 of 36 cells produced byte-identical greedy output. The single exception (GPT-2 at a 512-token prefix) is a one-token divergence from float16 rounding — prefilling the whole sequence and prefilling the query on top of a reused cache accumulate rounding slightly differently — and would disappear in float32; we report it rather than hide it. Third, and most decisive for the architecture, the KV cache is *thousands* of times larger than the text it stands in for (Table 6), and the ratio depends sharply on attention design: grouped-query models (Qwen, two KV heads) carry an order of magnitude less cache than multi-head models (GPT-2-large, twenty heads). The consequence shows up in the cross-node column: once the transfer cost of moving that cache is charged, GQA models stay comfortably above break-even (Qwen reaches 68×) while most MHA models fall below it. In other words, KV-bridge is worth it exactly when the producer and consumer are co-located *or* the model uses grouped-query attention — which is precisely the condition `RegistryKvTransport` encodes, falling back to CTX_REF otherwise. The measurement grounds the policy rather than assuming it.

9 Discussion and Limitations

Pulling the threads together: transcript accumulation is wasteful, and both summarizing and referencing fix it without hurting accuracy on the tasks we tried. Summarizing is cheaper in tokens but throws information away, which costs accuracy the moment a later step needs a discarded fact; referencing costs a little more but keeps the information and carries provenance with it. KV transfer is a separate lever again, paying off in compute when the shared prefix is long and the model is friendly to it. None of these dominates, which is the whole point. We would also flag the retracted Haiku regression as a result in its own right: it looked real at n=1 and evaporated at n=5, a reminder that a single agent generation is not a measurement.

We would rather name the soft spots than have a reviewer find them. The biggest is breadth of models on the *token* side. The token experiment runs on Claude Haiku, Sonnet, and Opus, which share a tokenizer and an architecture family, so strictly speaking it is one family at three scales, not a cross-vendor study, and the token counts in particular are tokenizer-specific. The losslessness effect we have since replicated on two open, non-Claude models (Qwen2.5-7B and Mistral-7B, 4-bit on a T4, n=10 per arm; §8.1), and the KV-bridge experiment spans seven open models and three attention designs — but that open-model breadth has not yet reached the token-count comparison, which remains the single most valuable next run.

Sample size and scenario diversity are the next concern. The token comparison is one scenario with a fixed upstream and only three samples per Sonnet/Opus cell, so those per-cell ratios are indicative rather than tight. The losslessness experiment is a single pre-registered document; it cleanly shows that a faithful summary drops facts and that a task needing one then fails, but it cannot tell us how *often* real workloads land in that situation. We are careful to claim existence, not frequency. Establishing frequency would mean running the same protocol over several document types — a contract with exact obligations, a spec with exact thresholds, narrative text — at larger n, and reporting whichever way it comes out.

The security result is genuine but deliberately modest in what it proves. Forty injection attempts across nine categories are all blocked with no false positives, and the chaining and obfuscation cases are there to show that the decision ignores the text entirely. But that same property means the corpus mostly confirms the lattice is implemented as specified; the interesting attacks live where the lattice makes no promise, and we say so by including two boundary cases it does not stop — an injection that stays inside an allowed capability, and content mislabeled at ingress. A real security evaluation would push on exactly those: the tagging boundary, a compromised broker, and output-side manipulation that a capability gate cannot see.

Two scope notes finish the list. The KV-bridge numbers are microbenchmarks of a single prefill step on one T4, measured against full recompute rather than against provider prefix-caching, which is the comparison a deployment would actually care about and which we do not claim to beat; the "transfer" is a local serialize/deserialize rather than a network hop, and the one float16 divergence in thirty-six cells means "lossless" here is token-identical in practice, not bit-exact in theory. And the implementation is a single in-memory broker process. That matters for our third argument: systematization only really shows up at scale, across many machines and restarts, and that is precisely the setting we have not built. We therefore present systematization as a design goal rather than a measured result, and treat a persistent, distributed store as future work, not as something this paper has demonstrated.

10 Future Work

Three things would move this from a careful pilot to a result that travels. First, extend the cross-vendor replication to the *token* study: the losslessness effect already replicates on two open models (Qwen2.5-7B and Mistral-7B, §8.1), but the token-count comparison still runs only on the Claude family, and carrying it to a non-Claude tokenizer is what would retire the last single-family caveat. Second, breadth on the two thin experiments: several document types for losslessness at $n \geq 10$ each, and a security corpus that attacks the parts the lattice does not guarantee — ingress mislabeling and within-capability steering — rather than the part it does. Third, the engineering that the systematization claim is really about: a durable, distributed context store (a persistent backing store with restart survival is the honest first step), message signing and cross-reconnect replay nonces, and a head-to-head against provider prefix-caching for the KV path. None of these change the design; they are about earning the claims the design implies.

On novelty. We claim none for the mechanism — shared stores, references, and KV reuse all predate us. The contribution is the measured, reproducible accounting that ties them together: a summarizer beats referencing on tokens, a summary's lossiness costs accuracy when a dropped fact is needed, the capability lattice contains the attacks it was built for and openly not the ones it was not, and KV transfer's compute win and storage cost are mapped across seven models — with an architecture that makes the byte/token split and the materialization choice explicit rather than implicit.

11 Conclusion

Two measurements, on two cost planes, point the same way. On tokens, reference-minimization removes the waste of transcript accumulation at no measured accuracy cost — but it is not the cheapest option, a competent summarizing orchestrator saves more, so TOAP earns referencing's place through losslessness, in-band security, and systematization rather than raw token count. On compute, transferring a same-model KV cache is a real, near-lossless prefill speedup that grows to two orders of magnitude on long contexts, yet the cache is so much larger than the text that it only pays off co-located or under grouped-query attention. The common thread is that no single way of sharing context dominates: the right choice depends on length, architecture, co-location, and whether lossiness is acceptable — which is the case for making materialization an adaptive, cost-model-driven decision rather than a fixed mechanism. We release all artifacts and call for cross-vendor, large-n, harder-task replication — especially a task that stresses the lossless-versus-lossy distinction, and a comparison of KV-bridge against provider prefix-caching — before any general claim.

References

[1] L. Erman, F. Hayes-Roth, V. Lesser, D. Reddy. The Hearsay-II Speech-Understanding System (blackboard model). *ACM Computing Surveys* 12(2), 1980.

[2] Anthropic. Model Context Protocol specification, 2024–2026. https://modelcontextprotocol.io.

[3] Google / Linux Foundation. Agent2Agent (A2A) Protocol v0.3, 2025–2026. https://a2a-protocol.org.

[4] Z. Chang, J. Li, Z. Cao. A Token-Efficient Data Layer for Agents (ADOL). IETF Internet-Draft `draft-chang-agent-token-efficient-01`, Dec. 2025.

[5] Anthropic, OpenAI, Google. Prompt caching documentation, 2024–2026.

[6] Z. Wang et al. Talk Structurally, Act Hierarchically (TalkHier). arXiv:2502.11098, 2025.

[7] CodeAgents: pseudocode-codified multi-agent messages. arXiv:2507.03254, 2025.

[8] Google DeepMind. Defeating Prompt Injections by Design (CaMeL). arXiv:2503.18813, 2025.

[9] KVCOMM: Online Cross-context KV-cache Communication. arXiv:2510.12872, 2025.

A Reproducibility

Artifacts: `benchmark/ab_study/` (harness, scenarios, the n=33 run records `runs/writer_runs.json`, deterministic scorer `writer_experiment.py`, analysis `compute_writer.py`); `paper/figures/` (figure scripts); `paper/METHODS.md` and `paper/REVIEW_RESPONSE.md` (method log and the reviewer-fix trail). The Rust implementation is in `crates/` (29 unit + integration tests, incl. real-TCP end-to-end, V2 binary round-trip, capability-lattice flow, cache layout, and KV-bridge fallback). Token reductions recompute exactly from the committed records via `compute_writer.py`.