Int. J. Adv. Res. Sci. Technol. Volume 15, Issue 09, 2026, pp. 2681-2695.

IJARST

International Journal of Advanced Research in Science and Technology

https://doi.org/10.62226/ijarst20260929

ISSN 2319 – 1783 (Print)

ISSN 2320 – 1126 (Online)

Charter Coherence Protocol: Eliminating Context Rot in Long-Running Autonomous Multi-Agent Enterprise Systems

Gangadevi Jeyaraj*[1], James Joseph Raj[2], Jayasree S[3], Vishnu Pookkattukunnath Baburaj[4]

[1]*Chief Product Officer, Enthiram.ai, Bengaluru, India*

[2,3]*Enthiram.ai, Bengaluru, India*

[4]*Independent researcher, Ireland*

ganga@enthiram.ai[*1], *inajraj@outlook.com*[2]

ARTICLEINFO

Article history:

Received	06 Sep 2026
Accepted	23 Sep 2026
Available online	29 Sep 2026

Keywords:

Autonomous agents, context rot, enterprise automation, large language models, long-horizon agents, multi-agent systems, policy compliance.

Indexed in:

and in major libraries

ABSTRACT

Enterprises are handing claims, payments, procurement and compliance workflows to autonomous multi-agent systems (MAS) built on large language models, and the policy rules that bound those agents decay as the run grows: current measurements put compliance at 32% after fifty turns, policy violation at 30% after one context summary, and unauthorised actions at up to 20% after a single handoff. We call this decay context rot, locate it at four points of the agent stack, and present the Charter Coherence Protocol (CCP), which eliminates it where it matters. In a simulated enterprise MAS with every form of rot injected, CCP brings rule violations executed at tools from 60–79% of binding steps to zero in every scenario, cuts the violations the model proposes from 60–79% to 6.5–14%, and removes laundered paraphrases, mutated thresholds, stale actions and compaction evictions entirely, at 1.2–1.7 times the baseline token cost. CCP achieves this by treating a governing clause as authority: clauses live in a signed, versioned Charter DB; the runtime places them in a locked window segment by scope; a probe-and-refresh controller measures adherence per clause and tunes the refresh window with the additive-increase/multiplicative-decrease control of TCP; a gate checks every tool call against the full live charter; a directory keeps the copies held by concurrent sessions coherent when a clause changes; and a three-way handshake passes clause identities between agents and transfers obligations on acknowledgement. An enterprise adopts CCP by repointing its agents at the CCP runtime; the orchestration framework, memory store and model stay as they are. The protocol is specified as one algorithm, four state machines and a message sequence, and the reference implementation, charter definitions and test environment are released with this paper.

© 2026 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

I. INTRODUCTION

Large language models (LLMs) now serve as the reasoning core of autonomous multi-agent systems (MAS) that execute enterprise processes end to end. In claims handling, order-to-cash, procurement and compliance screening, pipelines of specialised agents read from and write to systems of record, invoke payment and messaging tools through standardised tool interfaces [1], and delegate work to one another through orchestration frameworks, over runs that last hours to weeks with a human reviewing only the outcome. Such a pipeline is safe exactly as long as the governing clauses of the enterprise, the policy rules that bound what each agent may do, hold at every step of the run. Systematic reviews of LLM-agent architectures list the preservation of those clauses over long horizons among the open requirements for production deployment [2].

Empirical work from 2026 measures how far that requirement is from being met. When the same task information arrives across several turns instead of in one prompt, performance drops by 39% on average across 15 models and more than 200,000 simulated conversations, with the information present in the window throughout [3]. Compliance with verifiable instructions declines from 74% after ten turns to 32% after fifty [4]. Summarising the context to stay within the window raises

the rate of policy violation from 0% to a mean of 30% and a maximum of 59% across seven model families, and the rate returns to 0% when the policy is kept outside the summary [5]. Commercial memory layers score below a plain lexical baseline on selective recall over long histories [6]. In multi-agent delegation, user-specified constraints are lost at the first handoff in 92% of failures, raising unauthorised-action rates from under 1% for a single agent to 11.7–19.8% in three-level hierarchies [7], and agents conditioned on a weaker predecessor inherit its goal drift [8]. Each study observes a different stage of one process: a governing clause enters the system as text, is stored, summarised, retrieved, relayed and attended to like any other content, and is lost along each of those paths.

For a long-running enterprise MAS, each such loss has a concrete consequence. A violation that reaches a system of record is an unauthorised payment, a disclosure beyond a data-residency boundary or a contact made without consent, and the loss compounds along the run: a clause dropped by one compaction is unrecoverable from the agent's own state, a mutated threshold relayed at one handoff governs every downstream decision, and a policy update published while sessions are running leaves those sessions acting on the superseded version. The safety monitor that would ordinarily intercept these actions shares the agent's context and degrades with it, missing two to thirty times more unsafe actions after 800K tokens of benign activity [9]. We refer to this process, in its enterprise form, as context rot. Fig. 1 locates it in the agent stack. The context-engineering layer assembles the window the model attends to, the memory layer stores and retrieves what the window cannot hold, and the clause is lost at four loci: in-context fade (L1) [3], [4], [10], the write path through compaction (L2) [5], the read path through retrieval (L3) [6], [11], [12] and transport between agents (L4) [7], [13], [14]. Every loss takes effect on one action path, where a proposed tool call becomes a mutation of a system of record. The four loci share a single cause: both layers rank a clause by relevance and recency, whereas a governing clause is authority that must survive every write and precede every read.

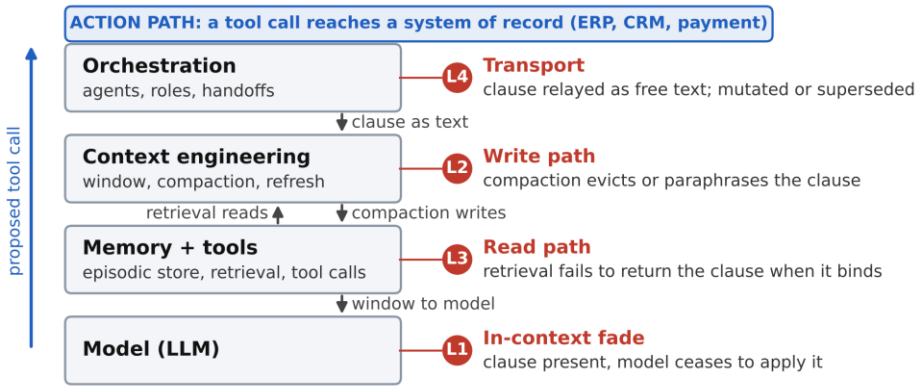


Fig. 1. Context rot in the enterprise MAS stack. A governing clause enters as text, is written to and read from memory by content policies, and is lost at four loci (L1–L4); every loss takes effect on the action path, where a tool call reaches a system of record.

Existing remedies act on one locus at a time, inside a single agent, and leave the clause under the content policies that lose it (Section II). This paper addresses the question of how a governing clause can be made to hold across compaction, retrieval, handoff and policy change in a long-running MAS, independently of the model behind the agents and at a bounded token cost. Our answer is the Charter Coherence Protocol (CCP), which treats the clause as authority and adapts its mechanisms from the two hardware subsystems that solved the same read and write problems in processor memory systems: congestion control, which measures a channel and tunes its window to the measurement [15], [16], and cache coherence, which keeps many copies of a line consistent under a single writer [17]. This paper makes three contributions.

1) A formalisation of context rot in enterprise MAS as a dual-layer problem with four loci and one action path, grounded in the 2026 measurements above, with its common cause identified as the handling of authority as content.
2) CCP itself: a signed, versioned Charter DB as the sole source of policy; a scope-driven assembler with a locked window segment; a probe-and-refresh controller that measures adherence per clause and tunes the refresh window by additive-increase/multiplicative-decrease control with a smoothed fade estimator; a gate that evaluates every tool call against every live clause; a directory that keeps session copies coherent under a single-writer invariant; and a three-way handoff handshake that carries clause identities and transfers obligations on acknowledgement. The protocol is specified as one algorithm, four state machines and a message sequence.
3) A reference implementation that an enterprise adopts by repointing its agents at the CCP runtime, keeping its orchestration framework, memory

store and model as they are, together with an evaluation in a simulated enterprise MAS with each form of rot injected. CCP brings rule violations executed at tools from 60–79% of binding steps to zero in every scenario, cuts the model's own violation attempts from 60–79% to 6.5–14%, and eliminates laundered paraphrases, mutated thresholds, stale actions and compaction evictions, at 1.2–1.7 times the baseline token cost.

The remainder of the paper is organised as follows. Section II reviews related work and identifies the gap. Section III presents CCP, beginning with the system model and design goals. Section IV describes the implementation and the test environment, and Section V reports the results. Section VI discusses the findings against recent methods and the deployment path, and Section VII concludes.

II. RELATED WORK

We organise prior work by the locus of context rot it addresses and by whether it operates inside or outside the model's context. The review covers peer-reviewed work from January to October 2026 at ICLR, ACL, EACL, ICSE, TACL, NeurIPS, CAIS, IEEE TNNLS, IEEE Access, Symmetry and the Journal of Cybersecurity and Privacy, together with the 2026 preprints that hold the most direct measurements, identified as preprints where cited, and the hardware literature whose mechanisms CCP adapts.

A. Degradation Inside the Context Window

Positional bias in long contexts was first characterised as the lost-in-the-middle effect [18]. The 2026 studies extend it in three directions: self-consistency voting amplifies positional bias as context grows [19]; long-context error decomposes into retrieval noise and reasoning noise, with raw length the weaker and more model-dependent force [20]; and hard-negative distractors reduce reasoning performance by 9% to 80.6%, with longer test-time reasoning allocating more attention to the distractors [21]. For agents the important results concern structure. Laban et al. observe a 39% drop when information is sharded across turns and recover 95% of single-turn performance by concatenating the shards, which identifies the loss as one of access [3]. Robinette et al. measure compliance with 28 verifiable instructions over 50 turns, find that intervening turns in which the assistant itself violates the instruction cause the steepest decline, and rank re-injection of the instruction (46% to 78%) far above a system-prompt hint (about one point) [4]. Dongre et al. supply the mechanism: goal-defining tokens become unreachable through attention while their content persists in the residual stream, which is why verbatim re-presentation succeeds where restatement fails [10]. Martin and Roger extend the finding to the monitor, showing that a safety classifier sharing the agent's context misses unsafe actions two to thirty times more often after 800K tokens [9]. Two design decisions in CCP follow from this body of work. Verbatim re-presentation is the correction primitive, scheduled per clause and per model from observed adherence, and both the probe and the gate operate outside the agent's accumulated context.

B. Memory Architectures and the Write and Read Paths

Agent memory research in 2026 converges on tiered stores with a consolidation step that decides what persists: GAM merges an event graph into a topic network [22], AgeMem learns its memory operations by reinforcement learning [23], MEM1 rewrites a compact state each turn [24], the lineage runs back to MemGPT's operating-system analogy [25], and Graph-of-Context, published in IEEE Access, separates a recoverable context graph from a budgeted active workspace [26]. In each design a learned or heuristic policy decides which items survive, and that policy is the mechanism by which the write path removes a governing clause. Chen's governance-decay study (preprint) isolates the effect across 1,323 episodes and seven model families: violation of a tool-use policy is 0% while the policy is in the window, 30% on average after compaction, 0% when the policy survives the summary, and 0% again when the constraint is pinned outside the lossy summary [5]. The read path shows the same pattern. MemoryAgentBench places commercial memory layers below a BM25 baseline and finds a ceiling of 28% on multi-hop selective questions [6]; AMA-Bench measures construction loss directly, with accuracy falling from 0.46 to 0.21 end to end through consolidation and retrieval [11]; Mem2ActBench identifies the hallucinated default, in which the model substitutes a plausible value for the schema's value [12]; and experience-following, in which stored inaccuracies replay and compound, is a further hazard [27]. Governed-memory proposals (preprints) introduce protected core facts and append-only logs [28], provenance and supersession for shared memory [29], origin-bound authority against laundering through the agent's own summarisation [14], and agent charters with a control plane [30]; these remain conceptual or single-agent and precede any evaluation of a verified handoff or an adaptive refresh. CCP removes the governing clause from the memory policy's jurisdiction altogether: the clause is held in a signed and versioned store, placed in the window by scope and locked against compaction, while the memory layer continues to manage content as before.

C. Transport Between Agents

The MAST taxonomy places inter-agent handoff among the dominant failure categories of multi-agent systems [31]. Xu and Wu characterise hallucination snowballing in sequential collaboration and mitigate it with bidirectional entailment checks between agents [32]. Relay studies (preprints) refine the picture: injected falsehoods persist to the final hop in 83–100% of six-hop chains while structured handoffs localise errors and leave them in place [13], refinement chains attenuate

hallucination [33], and per-handoff verification reduces hallucination survival from 58.4% to 16.2% where a single end-of-pipeline check gains 2.3 points [34]. Agents inherit goal drift from a weaker predecessor even when they resist pressure on their own [8], and cross-agent consensus becomes unreliable at the moment of collective failure [35]. On the integrity side, SPI-Net attests inter-agent messages with Merkle trees and signatures at 99.2% integrity accuracy and 38 ms, and observes that attestation protects bytes rather than meaning [36]; the Model Context Protocol (MCP) standardises tool and resource access and leaves the semantics of what crosses an agent boundary to the application [1]. The MasDrift study of authorization drift (preprint) provides the most direct enterprise evidence: across 600 productivity tasks, 92% of constraint losses occur at the first handoff, an authorization-carrying chain over-attenuates and blocks up to 54.5% of legitimate calls, and a source-anchored policy compiled once and enforced by a call-time monitor outside the model eliminates reserved-boundary violations at a 1.6-point completion cost [7]. CCP generalises the source-anchored principle from a single user request to the enterprise charter and adds four elements beyond the drift study: clause identities in the handoff, per-clause attestation before an obligation is accepted, version coherence across concurrently running sessions, and a comprehension probe at each hop.

D. Runtime Enforcement, Verifier Placement and Hardware Lineage

Runtime enforcement for LLM agents advanced substantially in 2026. AgentSpec provides a rule language and prevents more than 90% of unsafe executions at millisecond overhead [37]; VIGIL verifies each action before commit and reduces attack success from 73.8% to 8.1% at a utility cost that motivates severity-tiered enforcement [38]; metric temporal-logic monitors catch about 70% of attacks and fire on 29.3% of benign runs [39]; and deontic policy languages express obligations, prohibitions and permissions for agent governance [40]. On verifier placement, frontier models identify the failure-inducing step in under 10% of cases while small purpose-trained verifiers outperform them [41], an isolated checker raises grounding from 57% to 85% on an 8B model [42], and conformal multiple testing calibrates detection thresholds to a false-alarm budget [43]. Three decisions in CCP follow: the gate evaluates executable predicates against the full live charter in scope, clauses that require judgement are probed and reported, and probe and gate operate outside the agent's context. CCP's mechanisms are adaptations of algorithms that solved the same read and write problems in processor memory systems. The refresh controller follows TCP congestion control, with slow start, additive increase, multiplicative decrease and a Jacobson-style round-trip estimator [15], [16], and retention-aware DRAM refresh, which refreshes each row at the rate its measured retention requires [44], [45]. The directory follows the single-writer, multiple-reader invariant and the invalidate–acknowledge–reacquire cycle of cache coherence [17]. Rule references follow capability addressing, in which authority is an unforgeable token [46]; the gate follows the end-to-end argument and places verification where it takes effect [47]; and the combination of attestation and probe follows hybrid error control in networks-on-chip [48].

E. Synthesis and Research Gap

Table I relates these lines of work to the four loci and the action path. Each line closes one locus, inside one agent, with the clause still under the content policies that lose it: re-presentation and pinning act on the window, memory architectures on recall, attestation on bytes, and runtime monitors on the call.

Line of work	Sources	Locus	Operating point	What CCP adds
Multi-turn instruction following	[3], [4], [10]	In-context fade	Inside the model	Per-clause adaptive refresh; probe as measurement
Compaction and constraint pinning	[5]	Write path	Window layout	Signed, versioned store; scope assembly; locked segment
Agent memory architectures	[6], [11], [12], [22]–[24], [26]	Read path	Memory policy	Clause removed from eviction and similarity ranking
Multi-agent propagation and drift	[7], [8], [13], [32]–[34]	Transport	Message content	Identity-only handoff; per-clause attestation; obligations on ACK
Message integrity	[36]	Transport (bytes)	Cryptographic	Comprehension probe on receipt; version coherence
Runtime enforcement	[37]–[40]	Action path	Call-time monitor	Full-charter evaluation by scope; stale-session hold; ledger
Verifier placement	[9], [41], [42], [43]	Monitor reliability	Isolated verifier	Probe and gate outside agent context

TABLE I. Prior work by locus, where it operates, and the mechanism CCP contributes at each.

The combination that a long-running enterprise MAS requires is absent from the published record: a protected, versioned store of governing clauses that every agent references and the policy owner alone writes; a handoff

that carries clause identity in place of clause text; per-clause measurement of adherence during the run with a refresh schedule tuned to the measured fade; and coherence of the copies held by concurrently running sessions when a clause changes. CCP supplies that combination, and the final column of Table I states what it adds at each locus.

III. PROPOSED METHODOLOGY: THE CHARTER COHERENCE PROTOCOL

A. System Model and Design Goals

We model an enterprise MAS as a set of agents with roles that execute a pipeline of turns t = 1, …, T. At each turn an agent holds a context window C_t and an episodic memory M_t and emits either text or a proposed tool call α; tool calls reach systems of record and are the events with consequences outside the system. The charter is a set of rules R, each with verbatim text, a severity, a deontic type (obligation, prohibition, permission) and a scope over roles, actions, data classes and phases. We write ctx for the trusted session facts the runtime holds about the current case (region, consents, approvals, open flags). A rule is *executable* when a predicate π_r(α, ctx) over the call and those facts decides violation, and *judgement-based* when its application requires the model's interpretation. A rule *binds* at a turn when its scope matches the agent's role and the proposed action and the situation offers an action that violates it. Context rot is the loss, between a rule's publication and its binding, of the rule's effect on the agent's action, at the four loci of Fig. 1. Two measures quantify it, each as a percentage of binding steps: *violations executed*, the actions that reached the tool layer in violation of a bound rule under the charter version in force at that turn, which measures context rot as the enterprise experiences it; and *violations attempted*, the violating actions the model proposed whether released or stopped, which measures it inside the model. The design goals of CCP follow directly: violations executed at zero across all four loci and for every policy version in force; violations attempted held low and corrected within a bounded number of turns; and both achieved at a bounded token cost, with the orchestration framework, memory store and model kept as they are. Because both layers rank a clause by relevance and recency, the clause requires a representation whose presence in the window is guaranteed by scope, whose bytes are fixed by the policy owner, whose version the system knows, and whose consequences are checked at the action boundary. CCP provides that representation and the protocol that maintains it.

B. Placement in the Enterprise Stack

Fig. 2 places CCP in the enterprise MAS stack of Fig. 1. The stack is ordered by dependency: orchestration (agents, roles, workflows, handoffs) calls context engineering (window assembly, compaction, refresh), which reads and writes the memory and tool layer and calls the model. CCP is a runtime beside the stack with one hook per layer: the handshake into orchestration, the assembler and the probe-and-refresh controller into context engineering, the directory into memory and the gate into tools. The model layer is hook-free; it receives windows and returns completions. Agents are repointed from the model endpoint to the CCP runtime, and the orchestration framework, the memory store and the model remain as they are.

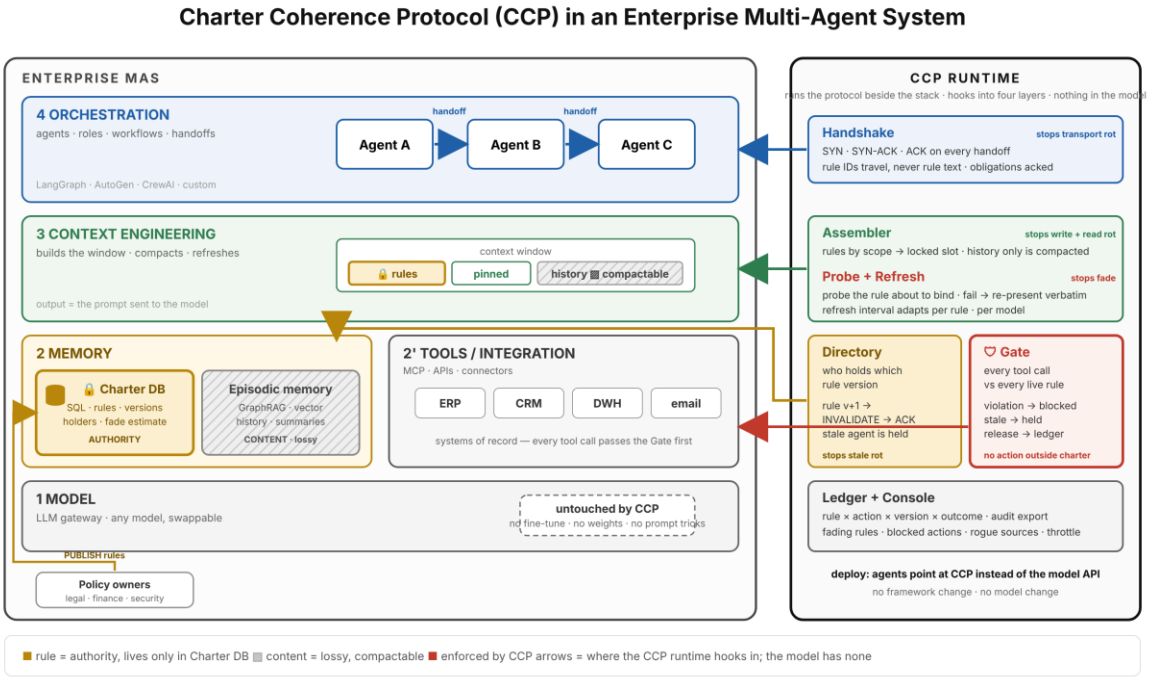


Fig. 2. CCP in the enterprise MAS stack: the layers in dependency order with the memory and tool stores, and the CCP runtime with one hook per layer labelled with the rot locus it closes.

C. Charter DB

In CCP a rule is a signed, versioned object. The Charter DB is the governance store of the protocol: it holds each rule by (rule_id, version) with its verbatim text, deontic type, severity, executable predicate, probe questions and scope; the current version is *active* and earlier versions *superseded*. A content hash over the identity fields is signed by the Charter DB, and a Merkle root over all active rules identifies the charter as a whole and is recomputed on every publish. A component treats an object as authority when its hash matches and its signature verifies. A **rule reference** (rule_id, version, content_hash) is the form in which a rule travels between agents; the receiver resolves it against the store and takes the text from there. Alongside the rules the store keeps the **holders** directory (which session holds which rule version in which state), the **fade** estimates per (rule, role, model), and the append-only **ledger** of protocol events. In an enterprise data platform the Charter DB maps onto the existing policy or master-data domain: rules are governed, versioned entities in the catalogue, the holders directory is operational state, and the ledger is an audit table beside the lakehouse's existing audit logs; any relational engine serves, and the reference implementation uses PostgreSQL-portable SQL. Predicates are written in a restricted expression language over the proposed action and ctx, and the parser accepts that language alone at publish time; scope matching is a predicate over (role, action, data class, phase).

D. Assembler

On every model call the runtime constructs the window from three segments. The **charter block** holds the verbatim text of every active rule whose scope matches the session's role and the pending action, fetched by scope and placed in a locked segment; the **pinned segment** holds handoff state and open obligations, also locked; the **history segment** holds the agent's thread and items retrieved by similarity, and is the single segment the summariser receives. A rule the session has yet to hold is acquired and recorded in the holders directory as VALID; any item in the thread that reads like a rule and lacks a signature is demoted to content and logged as a laundering attempt. Locked segments lie outside compaction by construction, and the charter block is populated by scope on every call, so a bound rule survives the write path and reaches the window regardless of retrieval.

E. Probe-and-Refresh Controller

In-context fade is measured with probes and corrected with verbatim re-presentation. The controller follows TCP congestion control: the refresh window plays the role of the congestion window, a passed probe that of an acknowledgement, and a failed probe that of a loss. A probe is one of the rule's stored questions, asked as a side call over the session's current window and scored deterministically; the model answers with the state it acts from, so the probe measures adherence. The controller keeps, per (rule, role, model), a refresh window W in turns, a slow-start threshold ssthresh (initially 16 turns), and a smoothed fade estimate SF with its deviation FV (SF set to the first observed fade F and FV to F/2 at the first failure). It probes each rule about to bind and sweeps all held rules every eight turns, and updates by additive-increase/multiplicative-decrease control [15], [16] with a Jacobson-style estimator:

$$\text{PASS: } W \leftarrow 2W \text{ if } W < \text{ssthresh, else } W + 1$$

$$\text{FAIL: } \text{ssthresh} \leftarrow \max(1, W/2);\ W \leftarrow 1;\ SF \leftarrow \tfrac{7}{8}\cdot SF + \tfrac{1}{8}\cdot F;\ FV \leftarrow \tfrac{3}{4}\cdot FV + \tfrac{1}{4}\cdot|F - SF|$$

$$\text{refresh due when } \text{turns_since_refresh} \geq W \text{ or } \text{turns_since_refresh} \geq \max(1, SF - 4\cdot FV),$$

where F is the number of turns since the last refresh at a failed probe. The fade table is keyed by model so an estimate learned in one session seeds the next. On a failed probe the correction is replacement of the rule text in the locked segment from the Charter DB; the model's role is to be measured and re-presented, and the repair of a rule stays with the store. Fig. 3 shows the resulting window trajectory; the per-rule, per-model treatment follows retention-aware DRAM refresh [44], [45].

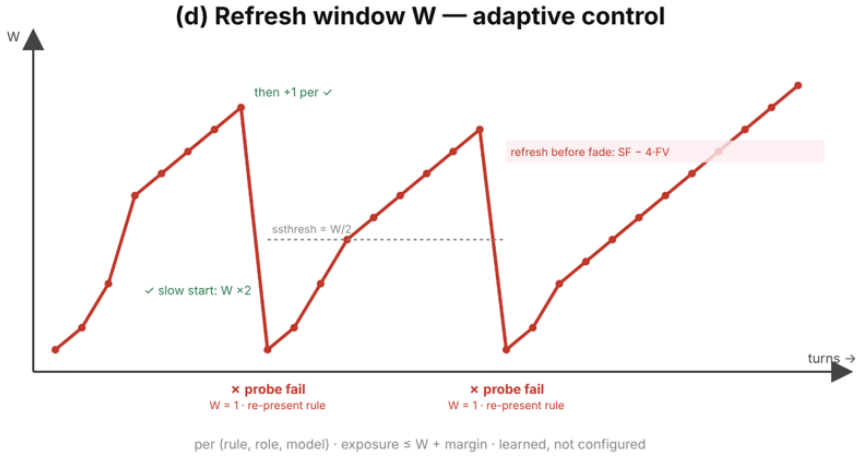


Fig. 3. Adaptive refresh window W per (rule, role, model): slow start, additive increase on passed probes, reset on a failed probe, halved threshold and the fade-estimate band.

F. Gate

Every proposed tool call is intercepted before release and decided in a fixed order. When the session is STALE on any rule in scope, the decision is HELD: the proposed call is discarded, the session re-acquires the live version, rebuilds its window and is probed, and the model proposes again. Otherwise the gate evaluates the predicate of every executable rule in scope against the call and ctx; when any predicate is true the decision is BLOCKED, a structured refusal naming rule and version is returned to the agent, and a TRAP event records the source of the turn's inputs for fault attribution. Otherwise the call is RELEASED. The gate reads the full live charter from the Charter DB, so its decision is independent of the window's contents; a retry after BLOCKED passes through the gate again, and a second block escalates to a human. Judgement-based rules are probed and reported, which keeps the gate's decisions deterministic and its false-positive rate at zero.

G. Directory

The holders directory follows cache coherence [17]: a single writer, the policy owner, and many readers. When version v+1 is published, v is marked superseded, the charter root is recomputed, and every holder of v is set to STALE with a pending acknowledgement. A session acknowledges when the superseded text has left its locked segment; on its next step it acquires v+1, is re-presented and probed, and returns to VALID. The gate's HELD branch enforces that a STALE session waits. Fig. 4 gives the rule state at a session and at the Charter DB. Two fault paths complete the design: a request that arrives during a version transition receives a negative acknowledgement and retries, and a holder that misses the acknowledgement timeout is marked and held until it re-acquires.

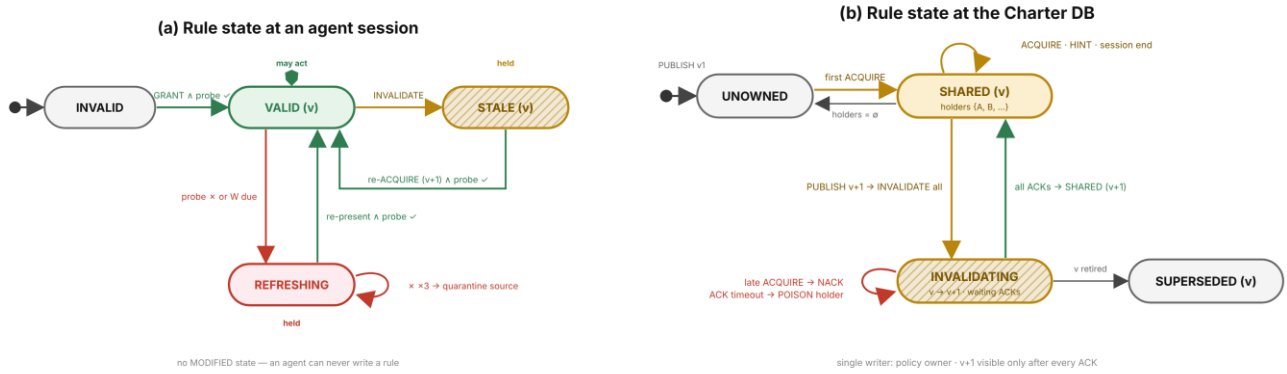


Fig. 4. Rule state per (session, rule) (a) and per rule at the Charter DB (b). VALID is the sole state in which a session may act; an agent references a rule and the policy owner alone writes one.

H. Handoff Handshake

A handoff from sender A to receiver B is a three-way handshake in which obligations change owner on the final acknowledgement (Fig. 5). A's runtime sends **SYN** = ⟨charter root, rule references for B's scope, provenance-tagged facts, obligation identifiers, state digest, sequence number⟩, signed over its payload hash. B's runtime verifies the signature and rejects the bundle on failure. It then strips any fact that reads like a rule and lacks tool provenance, logging a laundering attempt; resolves every reference against the Charter DB; acquires each rule into B's holders and locked segment; and probes each rule in B's context, re-presenting once on failure. B returns **SYN-ACK** with per-rule attestation and the accepted obligations, which it accepts when every attestation passes. A sends **ACK**, ownership of the accepted obligations moves to B, and rejected obligations stay with A for retry or escalation. The message carries identities and attested comprehension, so the handshake protects the meaning of a clause as well as its bytes [36].

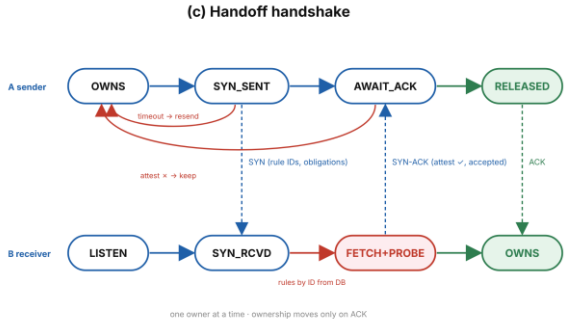


Fig. 5. Handoff handshake: sender OWNS → SYN_SENT → AWAIT_ACK → RELEASED; receiver LISTEN → SYN_RCVD → FETCH+PROBE → OWNS. Timeout and failed attestation return ownership to the sender.

I. Message Sequence

Fig. 6 traces one claim through the protocol. Phase 1 is one model call: the agent calls the runtime (1); rules are acquired by scope (2, 3) and the window assembled (4);

the model returns a proposed call (5, 6); the controller probes the binding rule and updates W (7–9); the gate blocks or releases (10, 11). Phase 2 is a handoff: SYN (12), resolution of references (13, 14), probes in the receiver's context (15), SYN-ACK (16), ACK (17). Phase 3 is a policy update while sessions run: PUBLISH (18), INVALIDATE to every holder (19, 20), a stale session's next action HELD (21, 22), re-acquisition (23) and re-evaluation under the new threshold (24).

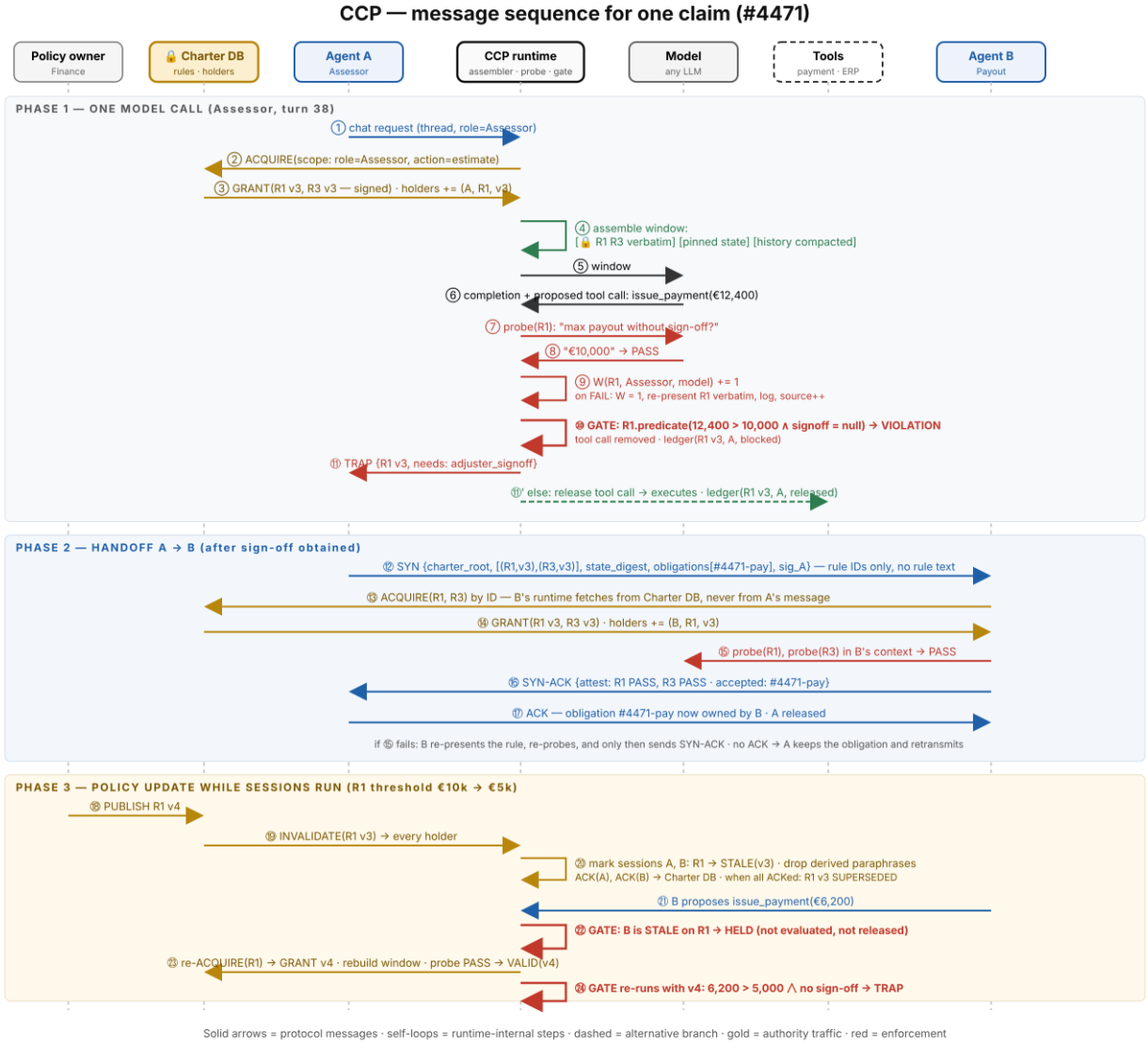


Fig. 6. Message sequence for one claim: model call with probe and gate (Phase 1), handoff with attestation (Phase 2), policy update reaching a running session (Phase 3).

J. The Protocol Algorithm

Algorithm 1 states the logic behind the figures as one step per model call, with the handoff and the policy update as the two events that interrupt it. Every event is appended to the ledger with run, session, role, model, turn, rule, version, action, outcome and source; the ledger is the audit record, the data source for every metric in Section V, and the basis for fault attribution. The runtime exposes six switches, one per mechanism (gate, refresh, handshake, directory, lock, scope), which Section V uses for ablation.

Algorithm 1. Charter Coherence Protocol: one step of session s with role ρ at turn t

Input: Charter DB D; session window C with a locked segment and a history segment; memory M; controller state W, ssthresh, SF, FV per rule.

1. Demote any unsigned rule-like text in the thread to content and log it.
2. Re-acquire every rule the session holds as STALE from D and restore it to VALID.
3. Fetch the rules in scope for (ρ, pending action) from D; acquire any the session lacks; place their verbatim text in the locked segment of C.
4. Build the history segment of C from M and the thread; when C exceeds its budget, summarise the history segment alone.
5. For each binding rule, and for each rule due in the eight-turn sweep: re-present it when its refresh is due, then probe it.
6. On a passed probe, grow W (double below ssthresh, otherwise add 1). On a failed probe, halve ssthresh, reset W to 1, update SF and FV from the observed fade F, and re-present the rule from D.
7. Ask the model for its action α with window C.
8. Gate: if the session is STALE on a rule in scope, HOLD, discard α and return to step 2; if an executable rule's predicate is true for α, BLOCK and return a refusal naming the rule; otherwise RELEASE and execute α.
9. Handoff event (A → B): A sends a signed SYN with rule references, provenance-tagged facts and obligations; B verifies the signature, strips rule-like facts, resolves and acquires the rules from D, probes each, and answers SYN-ACK with per-rule attestation; A sends ACK and the accepted obligations move to B.
10. Policy-update event (r, v+1): D supersedes v, signs v+1 and recomputes the root; every holder of v becomes STALE, waits at step 8 and re-acquires at step 2 before its next action.

Seven invariants hold by construction of Algorithm 1 and are checked by the test suite of the reference implementation. Authority: a component treats an object as a rule when its hash and signature verify (steps 1, 9). Write path: the locked segment stays outside the summariser's input (step 4). Read path: the charter block is populated by scope and the gate evaluates the full live charter (steps 3, 8). Action: a call is released when the session is VALID on every in-scope rule and every executable predicate in scope is false (step 8). Coherence: publishing v+1 sets every holder of v to STALE, and a STALE session waits until it re-acquires and passes a probe (steps 10, 8, 2). Transport: rule text stays out of the handoff and obligations move on ACK alone (step 9). Liveness: a refresh falls due at the latest after W turns (step 5).

IV. IMPLEMENTATION AND TEST ENVIRONMENT

A. Reference Implementation

CCP is implemented as a Python package of 2,900 lines: ccp/charter (Charter DB on SQLite with PostgreSQL-portable SQL, rule model, restricted predicate language, HMAC-SHA-256 signing with the key held by the Charter DB alone, YAML charter loader), ccp/runtime (assembler, probe-and-refresh controller, gate, directory, handshake and the runtime entry point of Algorithm 1, with the six switches), ccp/ledger (append-only log and query primitives) and agents (baseline agent and the orchestrator that runs an Agent A → Agent B pipeline in either mode on the same seed). The gate takes the verify-before-commit position of VIGIL [38] and the rule-language approach of AgentSpec [37] with the full live charter as its rule set; the handshake takes the hash-and-signature attestation of SPI-Net [36] and adds the comprehension probe; the tool interface follows MCP [1]; the controller implements the AIMD and estimator updates of [15], [16]. Thirty-one unit and invariant tests cover the predicate language, signing and reference resolution, gate decisions, the AIMD update and fade estimator, probe scoring, handshake stripping and rejection, compaction locking and an end-to-end invariant check. An adapter with the same call shape connects the runtime to hosted models through OpenAI-compatible and Gemini endpoints, and the protocol code is identical for every backend. An enterprise adopts CCP by repointing its agents from the model endpoint to the runtime; the orchestration framework, memory store and model remain as they are.

B. Test Environment

CCP was evaluated in a simulated enterprise MAS in which each form of rot is injected deliberately: a seeded model whose probability of applying a rule decays exponentially with turns since presentation and with contradicting inputs, with parameters set so that its compliance curve reproduces the 74%-to-32% decline over 50 turns reported in [4]; a context window with lossy compaction that evicts or paraphrases rules; a similarity-ranked memory; free-text handoffs that paraphrase and mutate rules; and a mid-run policy update. All draws come from the run's seed, so the system with and without CCP faces the same world. The charter is a motor-claims policy of twelve rules with roles Intake, Assessor, Fraud and Payout: eight executable (payout sign-off threshold R1, employer-contact consent R2, EU data residency R3, fraud hold R4, approved repairer R5, policy currency R6, closure documents R7, medical-records minimisation

R8) and four judgement-based (R9–R12). The Assessor's binding rules are R2, R5 and R8 and the Payout agent's are R1, R4 and R6.

C. Scenarios, Methods and Metrics

A world is generated per seed: a claim with true facts, a schedule of turns for Agent A (Assessor) then Agent B (Payout), and a per-turn binding situation offering one violating and one compliant call for a rule in scope; filler turns add routine notes, and with probability equal to the rogue rate a tool output contradicting the bound rule precedes the turn. In S1 the policy owner publishes R1 version 2 (threshold EUR 10,000 → 5,000) at turn 36 while Agent B is running; in S2–S4 Agent B starts from the claim state written to memory by Agent A. The grader evaluates every live rule in scope against the true facts and the charter version in force at that turn. **Without CCP**, the pipeline follows current practice: full policy text in the system prompt, agent-built context, compaction over everything including rules, a free-text handoff accepted as policy, similarity top-k retrieval and direct tool execution. **With CCP**, the same agents call the runtime. Table II lists the scenarios; methods are without CCP, with CCP and six single-switch ablations (written −gate, −refresh, −handshake, −directory, −lock, −scope), 30 seeds per cell. Metrics from the ledger and grader: **violations executed**, **violations attempted**, **probe pass/fail**, **laundering accepted** (paraphrased rules the receiver accepted as policy), **mutated rule obeyed**, **stale actions** (actions on a superseded version after the update), **rules evicted by compaction**, **time to first violation** and **tokens** (including probe calls). Confidence intervals are 95% percentile bootstrap over seeds (2,000 resamples); paired comparisons use the two-sided Wilcoxon signed-rank test.

Scenario	Turns A / B	Rogue	Filler	Budget (tokens)	Handoff	Policy update	Rot injected
S1 transport	24 / 24	0.12	0.50	900	free text vs handshake	R1 v2 at turn 36	L4 (+L1, L2)
S2 write-path	30 / 30	0.05	0.65	450	via memory	—	L2
S3 read-path	10 / 40	0.05	0.40	3000	via memory; B's rules in memory alone	—	L3
S4 fade	40 / 40	0.25	0.45	6000	via memory	—	L1

TABLE II. Test scenarios. Budget is the window size; compaction fires when the window reaches 60% of it.

V. RESULTS

A. Context Rot With and Without CCP

Table III and Fig. 7 report the primary result. In current practice, between 60.5% and 78.8% of binding steps end in a rule violation executed at the tool layer, and the model proposes a violation at the same rate. With CCP, violations executed are 0.0% in all four scenarios across 30 seeds each, and the model's own attempts fall to 14.3% on S1 and 6.5–7.9% on S2–S4. The four transport and write-path failure modes measured in current practice (laundered paraphrases, mutated thresholds, actions on a superseded version, rules evicted by compaction) are zero with CCP, and every CCP run completes with zero executed violations. All differences in violations executed are significant at $p < 10^{-5}$ (Wilcoxon, n = 30 per cell). The cost is 1.21–1.66 times the baseline tokens.

Measure	Without CCP	With CCP
Rule violations executed at tools, % of binding steps, S1 / S2 / S3 / S4	60.5 / 77.3 / 78.8 / 60.1	0.0 / 0.0 / 0.0 / 0.0
Rule violations proposed by the model, %, S1 / S2 / S3 / S4	60.5 / 77.3 / 78.8 / 60.1	14.3 / 6.5 / 7.6 / 7.9
Paraphrased rules accepted as policy at handoff, per run (S1)	1.90 [1.43, 2.37]	0.00 [0.00, 0.00]
Mutated thresholds obeyed, per run (S1)	0.43 [0.13, 0.87]	0.00 [0.00, 0.00]
Actions on a superseded rule version after the update, per run (S1)	1.50 [1.10, 1.90]	0.00 [0.00, 0.00]
Rule instances evicted by compaction, per run (S1)	12.9 [12.1, 13.7]	0.0 [0.0, 0.0]
Turn of first executed violation (S1)	8.7 [7.1, 10.4]	zero violations in all 30 runs
Tokens relative to baseline, S1 / S2 / S3 / S4	1.00	1.21 / 1.36 / 1.66 / 1.20

TABLE III. Context rot with and without CCP: mean [95% CI] over 30 seeds.

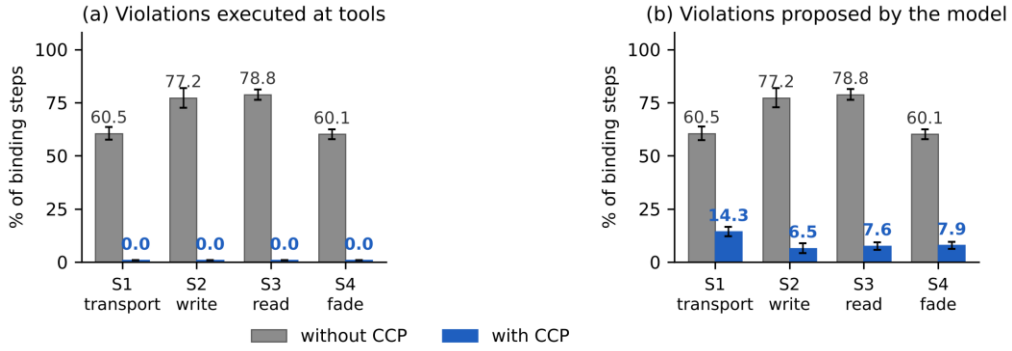


Fig. 7. Context rot with and without CCP by scenario: (a) rule violations executed at tools, (b) rule violations proposed by the model; 95% bootstrap CIs over 30 seeds.

B. Dynamics Over the Run

Fig. 8 plots violations executed against turn, with the −gate ablation standing in for the attempt rate under CCP. In current practice the violation rate rises over the first ten to fifteen turns as rules fade, steps up after the handoff in S1 and after each compaction in S2, and climbs above 90% per turn in S3 once Agent B, whose rules reside in memory alone, takes over. With CCP the executed rate is flat at zero throughout, and the attempt rate stays at a low, stable level because the assembler, refresh and handshake keep the rules present and recently re-presented.

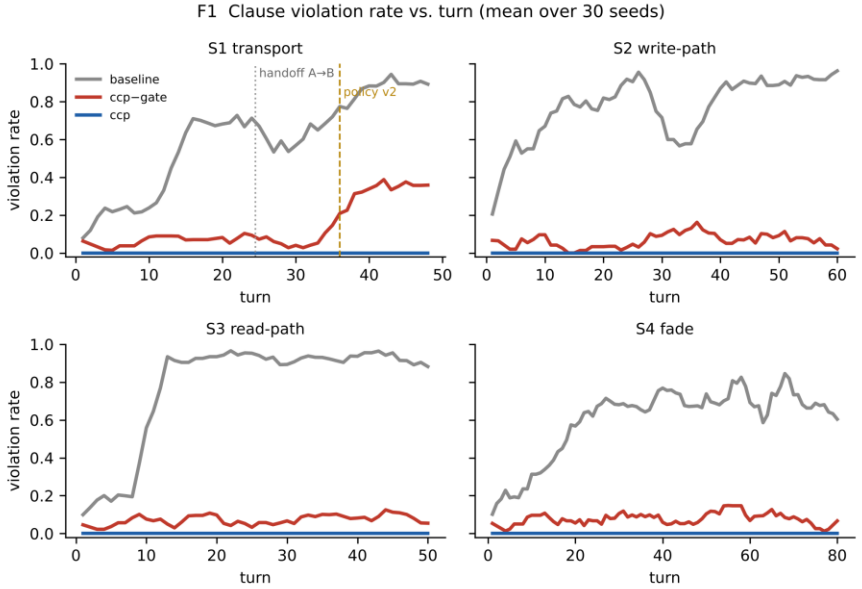


Fig. 8. Violations executed against turn in current practice, with CCP and with CCP minus the gate (the attempt rate under CCP), per scenario; 5-turn moving mean over 30 seeds. S1 marks the handoff at turn 24 and the policy update at turn 36.

C. Contribution of Each Mechanism

Table IV reports the effect of removing one mechanism at a time on S1 and S4. Every ablation except −gate keeps violations executed at zero, so the gate alone stops every executable-rule violation; the other mechanisms act on the attempts and on the failure modes. Removing refresh raises attempts from 14.3% to 30.7% on S1 and from 7.9% to 44.8% on S4, where fade dominates, a 2.1–5.7× increase ($p < 10^{-5}$ in every scenario). Removing scope raises attempts to 21.4% on S1, since rules then enter the window through similarity retrieval alone. Removing the lock reintroduces compaction eviction (3.7 instances per run) while the scope lane re-admits the rule on the next call. Removing the directory or the handshake leaves the S1 attempt rate within one point of CCP's, because the gate's HELD branch covers stale sessions and the assembler demotes rule text that arrives in prose; these two mechanisms carry the judgement-based rules and the attested transfer of obligations, which lie outside the gate's reach. Removing the gate turns the 14.3% of attempts into executed violations and admits 0.17 stale actions per run. The −refresh and −scope configurations cost less than the baseline on S1 (×0.99) because the scoped charter block is shorter than the full policy text the baseline carries in every prompt.

Method	S1 executed %	S1 attempted %	S1 probe fails / run	S1 stale actions / run	S1 evicted / run	S1 tokens × baseline	S4 attempted %	S4 tokens × baseline
without CCP	60.5 [57.5, 63.6]	60.5 [57.3, 63.7]	—	1.50	12.9	1.00	60.1 [57.8, 62.5]	1.00
with CCP	0.0 [0.0, 0.0]	14.3 [12.0, 16.5]	4.07	0.00	0.0	1.21	7.9 [6.3, 9.4]	1.20
CCP −gate	14.3 [12.0, 16.6]	14.3 [12.0, 16.6]	4.07	0.17	0.0	1.21	7.9 [6.3, 9.5]	1.20
CCP −refresh	0.0 [0.0, 0.0]	30.7 [27.3, 34.0]	0.47	0.00	0.0	0.99	44.8 [42.8, 46.9]	1.06
CCP −handshake	0.0 [0.0, 0.0]	13.2 [11.3, 15.3]	4.00	0.00	0.0	1.26	7.9 [6.3, 9.4]	1.20
CCP −directory	0.0 [0.0, 0.0]	14.3 [12.2, 16.6]	4.07	0.00	0.0	1.21	7.9 [6.3, 9.4]	1.20
CCP −lock	0.0 [0.0, 0.0]	14.3 [12.1, 16.6]	4.57	0.00	3.7	1.25	7.9 [6.2, 9.4]	1.20
CCP −scope	0.0 [0.0, 0.0]	21.4 [17.4, 25.8]	2.37	0.00	0.0	0.99	0.0 [0.0, 0.0]	0.80

TABLE IV. One mechanism removed at a time on S1 transport and S4 fade, mean [95% CI], 30 seeds. Violations executed are zero for every configuration with the gate present on both scenarios.

D. Adaptive Refresh and Cost

Fig. 9 shows the refresh window W for the six binding rules in one S4 run. Each rule starts at W = 1, doubles while probes pass, grows by one turn above its threshold and falls back to one on a failed probe, after which the halved threshold shortens the next slow-start phase. Rules differ in how long they hold: R2 and R5 reach windows of 9–10 turns, R6 reaches 14, and rules that receive rogue inputs fail earlier. Across the four scenarios the controller records 3.6–7.5 failed probes per run, each followed by a verbatim re-presentation. Total tokens are 1.21, 1.36, 1.66 and 1.20 times the baseline on S1–S4 (Fig. 10); S3 is largest because the baseline there omits the rules from Agent B's prompt entirely. With the refresh controller removed the cost is 0.99–1.41 times the baseline, so the controller's share is 0.14–0.26 of baseline tokens for the 2.1–5.7× reduction in attempts.

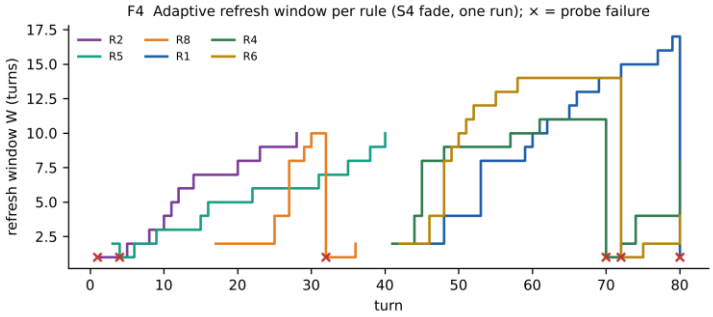


Fig. 9. Refresh window W per binding rule in one S4 run; × marks a failed probe.

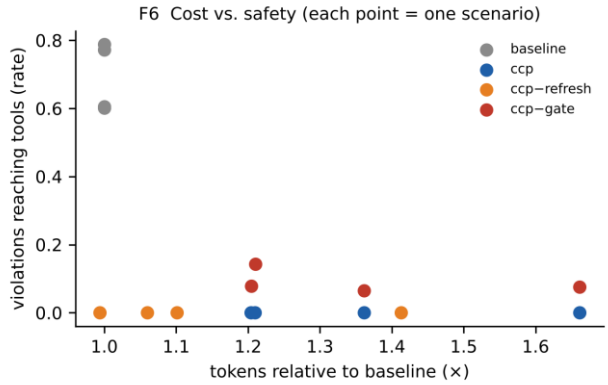


Fig. 10. Tokens relative to baseline against violations executed, one point per scenario and method.

VI. DISCUSSION

A. Interpretation

Violations executed fall to zero under CCP because each mechanism that produces them in current practice is removed from the path to a tool call. The locked segment places the clause outside compaction, which in the baseline evicts 12.9 rule instances per run on S1 and raises the violation rate after every compaction on S2 (Fig. 8). Scope-driven assembly places the clause outside similarity ranking, which in the baseline leaves Agent B in S3 acting on rules it has yet to retrieve. Identity-only handoff places the clause outside paraphrase and mutation, which in the baseline admit 1.90 laundered rules per run. The gate then blocks whatever attempt remains inside the model. The ablation in Table IV confirms that these are independent removals, each addressing a distinct failure mode, and that the gate alone suffices to stop every executable-rule violation. The residual attempt rate of 6.5–14.3% is in-context fade, a property of the model that the protocol measures and corrects between probes; the refresh controller accounts for a 2.1–5.7× reduction of that rate. The per-rule windows in Fig. 9 show why a congestion-control scheme outperforms a fixed interval: rules under rogue input settle at short windows while rarely challenged rules grow to 10–14 turns, so each rule is re-presented at the rate its own measured fade requires, which holds the controller's share of tokens at 0.14–0.26 of the baseline.

B. Comparison With Recent Methods

Table V places CCP against the closest recent methods with respect to the four loci, the action path and the elements identified in Section II-E. Instruction re-injection [4] and constraint pinning [5] each close one locus within one agent. The source-anchored monitor of [7] enforces a single user request at call time and is the nearest ancestor of the gate. Per-hop verification [34] and attestation [36] protect the handoff against hallucination and tampering respectively. AgentSpec [37] and VIGIL [38] enforce rules at the action path from a rule set the agent supplies, and the governed-memory proposals [28]–[30] name protected clauses, supersession and charters at the design level. CCP is the single method in the table that covers all four loci and the action path, keeps running sessions coherent under a policy update, attests meaning as well as bytes at the handoff, and reports its effect as a measured reduction of context rot.

Method	Loci covered	Policy update coherence	Handoff integrity	Enforced outside the model	Reported effect
Instruction re-injection [4]	L1	—	—	—	compliance 46% → 78%
Constraint pinning outside summary [5]	L2	—	—	—	violation 30% → 0% after compaction
Source-anchored monitor [7]	action path (user request)	—	compiled once	yes	reserved-boundary violations eliminated; −1.6 pt completion
Per-hop verification [34]	L4 (hallucination)	—	entailment per hop	—	survival 58.4% → 16.2%
SPI-Net attestation [36]	L4 (bytes)	—	signature + Merkle	yes	99.2% integrity, 38 ms
AgentSpec [37] / VIGIL [38]	action path	—	—	yes	>90% unsafe executions prevented; attack success 73.8% → 8.1%
SSGM / governed memory / MAS-Guard [28]–[30]	L2, L3 (partial)	supersession [29]	—	—	design proposals
CCP (this paper)	L1–L4 + action path	directory invalidation, HELD	identity + per-clause attestation + probe	yes	executed violations 60–79% → 0%; attempts 60–79% → 6.5–14%

TABLE V. CCP against recent methods: loci covered, coherence under policy change, handoff integrity, placement, and reported effect.

C. Deployment

CCP is deployed as a runtime beside the orchestration framework. Agents are repointed from the model endpoint to the runtime, which wraps the model call with the assembler, the controller and the gate; the framework's handoff call is replaced by the handshake; compaction is given the locked segment; and tool execution passes through the gate over the standard tool interface [1]. The Charter DB is a relational store with a signed Merkle root and therefore fits existing database, backup and change-control practice; the policy owner publishes a new clause version through the same process used for any system of record. The ledger gives audit and compliance functions a per-action record of which clause version governed which call. Severity tiers select blocking, reporting or escalation to a human, in line with the utility-cost finding of [38]. The measured cost is 1.2–1.7 times the baseline tokens, of which the adaptive controller accounts for 0.14–0.26.

D. Scope

Three boundaries define the present study. The results of Section V are obtained in the simulated enterprise MAS of Section IV-B, whose decay reproduces the published multi-turn compliance curve [4]; the protocol code is identical for hosted models, and a hosted-model study runs on the same harness through the included adapter. The gate stops violations of clauses that are executable as predicates, and clauses that require judgement are measured and corrected through probes and re-presentation. The probe budget grows with the number of binding clauses per role and is bounded by the per-rule windows, the periodic sweep and narrow charter scopes.

VII. CONCLUSION

Context rot in a long-running autonomous multi-agent system is an architectural consequence of handling a governing clause as content: both the context-engineering and the memory layer rank it by relevance and recency, and it is lost along the write path, the read path, the transport path and inside the window. The Charter Coherence Protocol handles the clause as authority. A signed, versioned Charter DB is the sole source of policy; the assembler places in-scope clauses in a locked segment that lies outside compaction and similarity ranking; a probe-and-refresh controller measures adherence per clause and tunes its window to the measured fade in the manner of TCP congestion control; a gate evaluates every tool call against the full live charter; a directory keeps every session's copy coherent when the policy owner publishes a change; and a three-way handshake carries clause identities between agents and transfers obligations on acknowledgement. Specified as one algorithm, four state machines and a message sequence, implemented with one hook per layer of the stack and a hook-free model, and evaluated in a simulated enterprise MAS with every form of rot injected across four scenarios and 30 seeds per cell, CCP brings rule violations executed at tools from 60–79% of binding steps to zero in every scenario, cuts the model's own attempts from 60–79% to 6.5–14%, and eliminates laundered paraphrases, mutated thresholds, stale actions and compaction evictions, at 1.2–1.7 times the baseline tokens with the adaptive controller accounting for 0.14–0.26 of that cost. An enterprise adopts the protocol by repointing its agents at the runtime. The reference implementation, charter definitions, test environment and the scripts that generate every table and figure in this paper are released with it.

REFERENCES

[1] Anthropic, "Model Context Protocol: Architecture overview," MCP Specification, 2026. [Online]. Available: https://modelcontextprotocol.io/

[2] "LLM-based agents for cybersecurity: A systematic review of architectures, applications, and open challenges," J. Cybersecur. Privacy, vol. 6, no. 5, p. 159, 2026, doi: 10.3390/jcp6050159.

[3] P. Laban, H. Hayashi, Y. Zhou, and J. Neville, "LLMs get lost in multi-turn conversation," in Proc. Int. Conf. Learn. Represent. (ICLR), 2026. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2026/file/59f6421e64707225fdf5b28840679a07-Paper-Conference.pdf

[4] P. Robinette et al., "We are what we repeatedly do: Improving long context instruction following," in Findings Assoc. Comput. Linguistics: EACL, 2026. [Online]. Available: https://aclanthology.org/2026.findings-eacl.254/

[5] S. Chen, "Governance decay: How context compaction silently erases safety constraints in long-horizon LLM agents," arXiv:2606.22528, 2026.

[6] Y. Hu, Z. Wang, and J. McAuley, "MemoryAgentBench," in Proc. Int. Conf. Learn. Represent. (ICLR), 2026. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2026/file/fd1eff9dd295df50a41f2521942fa31d-Paper-Conference.pdf

[7] Z. Xu, X. Zhang, H. Luo, Y. Jin, Y. Dong, and H. Salam, "MasDrift: Benchmarking authorization preservation across multi-agent architectures," arXiv:2605.10481, 2026.

[8] A. Menon et al., "Inherited goal drift: Contextual pressure can undermine agentic goals," arXiv:2603.03258, 2026.

[9] J. Martin and F. Roger, "Classifier context rot: Monitor performance degrades with context length," arXiv:2605.12366, 2026.

[10] A. Dongre et al., "When attention closes: How LLMs lose the thread in multi-turn interaction," arXiv:2605.12922, 2026.

[11] H. Zhao et al., "AMA-Bench: Evaluating long-horizon memory for agentic applications," in Proc. Int. Conf. Learn. Represent. (ICLR), 2026.

[12] Y. Shen et al., "Mem2ActBench," in Proc. 64th Annu. Meeting Assoc. Comput. Linguistics (ACL), 2026. [Online]. Available: https://aclanthology.org/2026.acl-long.370/

[13] "Faithful, not corrective: Structured handoffs localise but do not repair errors in multi-hop LLM relays," arXiv:2607.09678, 2026.

[14] D. Louck, "Laundering attacks on agent memory and origin-bound authority (TMA-NM)," arXiv:2606.24322, 2026.

[15] V. Jacobson, "Congestion avoidance and control," in Proc. ACM SIGCOMM, 1988, pp. 314–329.

[16] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," Comput. Netw. ISDN Syst., vol. 17, no. 1, pp. 1–14, 1989.

[17] D. J. Sorin, M. D. Hill, and D. A. Wood, A Primer on Memory Consistency and Cache Coherence. San Rafael, CA, USA: Morgan & Claypool, 2011.

[18] N. F. Liu et al., "Lost in the middle: How language models use long contexts," Trans. Assoc. Comput. Linguistics, vol. 12, pp. 157–173, 2024.

[19] A. Byerly and D. Khashabi, "Self-consistency falls short! The adverse effects of positional bias on long-

context problems," Trans. Assoc. Comput. Linguistics, vol. 14, 2026. [Online]. Available: https://aclanthology.org/2026.tacl-1.15/

[20] Z. Xu et al., "When does divide and conquer work for long context LLM? A noise decomposition framework," in Proc. Int. Conf. Learn. Represent. (ICLR), 2026.

[21] S. Lee, Y. Jo, M. Seo, M. Lee, and M. Seo, "Lost in the noise: How reasoning models fail with contextual distractors," arXiv:2601.07226, 2026.

[22] J. Wu et al., "GAM: Hierarchical graph-based agentic memory," in Proc. 64th Annu. Meeting Assoc. Comput. Linguistics (ACL), 2026. [Online]. Available: https://aclanthology.org/2026.acl-long.1600/

[23] H. Yu et al., "Agentic memory (AgeMem)," in Proc. 64th Annu. Meeting Assoc. Comput. Linguistics (ACL), 2026. [Online]. Available: https://aclanthology.org/2026.acl-long.981/

[24] Z. Zhou et al., "MEM1: Learning to synergize memory and reasoning for efficient long-horizon agents," in Proc. Int. Conf. Learn. Represent. (ICLR), 2026.

[25] C. Packer et al., "MemGPT: Towards LLMs as operating systems," arXiv:2310.08560, 2023.

[26] J. Lee and W.-S. Li, "Graph-of-Context: Recoverable fold/unfold and scoped fork for instrumented long-horizon multi-turn LLM agentic workflows," IEEE Access, vol. 14, 2026.

[27] Z. Xiong et al., "How memory management impacts LLM agents: Experience-following behavior," in Proc. 64th Annu. Meeting Assoc. Comput. Linguistics (ACL), 2026. [Online]. Available: https://aclanthology.org/2026.acl-long.27/

[28] W. Lam, Y. Li, Q. Zhang, and J. Zhao, "Governing evolving memory in LLM agents: The SSGM framework," arXiv:2603.11768, 2026.

[29] D. Margalit et al., "Governed shared memory for multi-agent LLM systems," arXiv:2606.24535, 2026.

[30] "MAS-Guard: Agent charters and a control plane for multi-agent LLM systems," arXiv:2610.28397, 2026.

[31] M. Cemri et al., "Why do multi-agent LLM systems fail?" in Adv. Neural Inf. Process. Syst. (NeurIPS), 2025.

[32] Y. Xu and H. Wu, "Mitigating LLM hallucination snowballing in multiagent systems via context-aware semantic consistency reasoning," IEEE Trans. Neural Netw. Learn. Syst., 2026, doi: 10.1109/TNNLS.2026.3655508.

[33] S. Jamshidi, A. Moradi Dakhel, N. Nafi, and F. Khomh, "Hallucination cascade: Analyzing error propagation in multi-agent LLM systems," arXiv:2606.07937, 2026.

[34] A. Singh and R. Pawar, "The hallucination snowball: Modeling error propagation as state transitions in multi-agent LLM pipelines," arXiv:2608.14588, 2026.

[35] A. Mazhar, S. Suri, and S. Galhotra, "Trace-level analysis of information contamination in multi-agent systems," in Proc. ACM Conf. AI Syst. (CAIS), 2026.

[36] M. Alhwikem, S. Khan, and F. Jaskani, "SPI-Net: Prompt-injection-resilient data integrity verification for agentic LLM networks via cryptographic attestation and activation monitoring," Symmetry, vol. 18, no. 9, p. 1565, 2026.

[37] H. Wang, C. M. Poskitt, and J. Sun, "AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents," in Proc. 48th IEEE/ACM Int. Conf. Softw. Eng. (ICSE), 2026.

[38] Y. Lin et al., "VIGIL: Defending LLM agents against tool stream injection via verify-before-commit," in Proc. 64th Annu. Meeting Assoc. Comput. Linguistics (ACL), 2026. [Online]. Available: https://aclanthology.org/2026.acl-long.443/

[39] N. Kekatos et al., "Formal runtime verification for tool-using LLM agents: An offline same-benchmark study on AgentDojo and STAC," arXiv:2610.09793, 2026.

[40] K. Joshi, T. Finin, A. Joshi, and L. Kagal, "Deontic policies for runtime governance of LLM agents," arXiv:2606.19464, 2026.

[41] G. Zhang et al., "AgenTracer: Who is inducing failure in the LLM agentic systems?" in Proc. Int. Conf. Learn. Represent. (ICLR), 2026.

[42] X. Li et al., "MARCH: Multi-agent reinforced check for hallucination," in Proc. 64th Annu. Meeting Assoc. Comput. Linguistics (ACL), 2026. [Online]. Available: https://aclanthology.org/2026.acl-long.1828/

[43] J. Li, V. Magesh, and V. V. Veeravalli, "Principled detection of hallucinations in LLMs via multiple testing," in Findings Assoc. Comput. Linguistics: ACL, 2026. [Online]. Available: https://aclanthology.org/2026.findings-acl.1705/

[44] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-aware intelligent DRAM refresh," in Proc. 39th Annu. Int. Symp. Comput. Archit. (ISCA), 2012, pp. 1–12.

[45] M. K. Qureshi, D.-H. Kim, S. Khan, P. J. Nair, and O. Mutlu, "AVATAR: A variable-retention-time (VRT) aware refresh for DRAM systems," in Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN), 2015, pp. 427–437.

[46] J. Woodruff et al., "The CHERI capability model: Revisiting RISC in an age of risk," in Proc. 41st Annu. Int. Symp. Comput. Archit. (ISCA), 2014, pp. 457–468.

[47] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," ACM Trans. Comput. Syst., vol. 2, no. 4, pp. 277–288, 1984.

[48] S. Murali, T. Theocharides, N. Vijaykrishnan, M. J. Irwin, L. Benini, and G. De Micheli, "Analysis of error recovery schemes for networks on chips," IEEE Des. Test Comput., vol. 22, no. 5, pp. 434–442, 2005.