# Agent Contract Protocol: Towards Reliability as a Protocol-Level Requirement for the Agentic Web

**Anonymous ACL submission**

## Abstract

Reliability is a core but currently unsatisfied requirement of the Agentic Web, where LLM-powered agents coordinate through standardized APIs such as MCP and A2A. Existing protocols rely on stateless request-response interactions that cannot prevent false commitments: cases where an agent confidently commits an incorrect answer after receiving a semantically corrupted tool response that passes transport-layer validation intact. Because such corruption is invisible to the protocol, verification must occur independently of model intelligence. To address this gap, we implement the Agent Contract Protocol (ACP), a proof-of-concept session-layer protocol built on persistent context handles, structured error signaling, and deterministic task bounds. We evaluate ACP against MCP across code generation, SQL, software-engineering, and constrained-planning settings under controlled semantic noise injection. Across all benchmarks and noise levels ($\varepsilon \in 0.0$–$0.6$), ACP achieves a false commitment rate of 0.000, while MCP commits false outputs at rates of 0.27–1.00. Results across multiple model families indicate that the reliability gap is structural rather than capability-dependent, while ACP preserves and channels model capability.

## 1 Introduction

Automated workflows have shifted from large language model (LLM) single-shot inference to Multi-Agent Systems (MAS), coordinated via stateless, discovery-based protocols. Protocols like the Model Context Protocol (MCP) (Hou et al., 2025) and Agent-to-Agent (A2A) (Google, 2024) work by publishing capabilities and exchanging single-shot requests, treating agent calls independently with no protocol-level guarantees and memory.

This paper demonstrates that stateless protocols embed a structural reliability gap into agentic pipelines, revealing that reliability must be enforced architecturally rather than delegated to language model capabilities. Lacking session-aware state, standard request-response protocols cannot prevent false commitments, where agents feed corrupted tool outputs, creating erroneous final generations. We define a *false commitment* as a session where the protocol accepts and commits an agent's output despite its failure to pass an execution-level oracle. This failure mode is unlike schema validation failures, arising when a tool response is syntactically valid but is semantically wrong with respect to the task. Under a stateless protocol, the agent receives the corrupted tool response, incorporates it into its answer, and propagate errors forward; the protocol has no mechanism to intercept the corrupted output. This failure mode persists regardless of model capability, because the corruption is invisible to the protocol.

**The Agentic Web.** We define the Agentic Web as a multi-domain distributed system in which the primary unit of exchange is not data but *delegated intent*: autonomous agents coordinating, negotiating, and executing tasks on behalf of users and other agents across organizational boundaries. Unlike the "Web of Documents" (resource retrieval) or the microservice web (idempotent API calls), the Agentic Web is characterized by long-lived, stateful, cross-agent interactions where correctness emerges from iterative refinement rather than single-request semantics.

While multi-turn language model architectures typically preserve state by accumulating conversational chat histories, this context mechanism operates independently of the underlying communication layer. This conflates two distinct notions of state. Application-level context, maintained in a prompt window, records what has been said, not enforcing what the protocol will accept as a valid committed output. Internal history allows a model to reason, but does not structurally intercept cor-

rupted agent outputs before commitment. Instead, protocol-level state secures the interaction boundary independently of individual model reasoning capabilities.

While scaling model capabilities may improve an agent's internal capacity for error detection, it does not resolve the underlying structural vulnerability. A corrupted tool output remains entirely invisible to the agent at commit time unless the communication protocol explicitly enforces an execution oracle. Consequently, increasing a model's internal reasoning capacity cannot compensate for a fundamental lack of architectural visibility; these represent categorically different failure modes.

In this paper, we introduce the Agent Contract Protocol (ACP), a stateful session-layer communication protocol designed to eliminate structural reliability gaps inherent in stateless agent protocols. We evaluate ACP against MCP across diverse benchmark datasets. The contributions of this work are threefold: (i) ACP itself, an implementation of a stateful agent protocol; (ii) three foundational architectural primitives, namely persistent context handles, structured error signaling, and deterministic task bounds; and (iii) an empirical demonstration of how session-layer verification systematically eliminates false commitments where stateless architectures fail.

Our results demonstrate that reliability is a foundational architectural requirement for agent coordination and multi-step workflows. Agent interaction must be modeled as a stateful, session-oriented process with protocol-level guarantees for continuity, error recovery, and transactional integrity. While ACP serves as a concrete implementation of this principle, the structural dependency is universal: without a dedicated session layer, agentic frameworks remain brittle regardless of downstream orchestration or prompting strategies.

Throughout, we use *reliability* to mean the protocol-level guarantee that a multi-step agent task either completes correctly or fails in a recoverable, well-defined manner. *Robustness*, by contrast, refers to a system's behavior under stress; reliability is one property a robust system exhibits.

**Reliability in Multi-Agent LLM Systems** The core arguments of this paper concern protocol design, but the implications bear directly on NLP research priorities. We identify four reasons why reliable agentic protocols are a language technology problem.

**Evaluation gap.** Most agent benchmarks, including AgentBench (Liu et al., 2023), GAIA, and WebArena, score single-trajectory task completion. They do not measure *recovery from failure mid-trajectory*. This opens a new evaluation axis: how well does a system detect, signal, and recover from mid-task failures?

**Capability vs. infrastructure confound.** When a multi-agent system fails, current methodology cannot distinguish "the model lacks capability" from "the protocol discarded the state needed to recover." Without protocol-level guarantees, every model evaluation conflates these two causes.

**Learning objectives.** Once protocol-level state and structured error signals exist, agents can be trained on *recovery trajectories*, sequences in which a failure is detected, diagnosed, and corrected within a session. This training regime is invisible under stateless protocols because the state needed to construct recovery examples does not survive failures.

**Reproducibility.** Stateful protocols make agent execution traces reproducible across runs in ways stateless logs cannot, since the session state provides an auditable record of what each party committed and when. This addresses a known reproducibility problem in agent-systems research (Cemri et al., 2025).

## 2 Related Work

Recent work on LLM-based multi-agent systems has focused on orchestration frameworks and strategies for coordinating agent interaction. Systems such as AutoGen (Wu et al., 2023), OpenAI Swarm (OpenAI, 2024), and the A2A protocol (Google, 2024) enable agents to delegate tasks and exchange structured messages, typically over stateless transports.

Effective LLM agents require structured interaction over multiple reasoning and action steps rather than single-shot inference. ReAct (Yao et al., 2023) shows that interleaving reasoning and acting enables agents to decompose tasks into sequential tool calls, implicitly introducing stateful execution traces rather than isolated responses.

Surveys of LLM multi-agent systems treat communication and coordination as central challenges but largely assume request-response patterns inherited from microservice architectures (Guo et al., 2024). Empirical studies show that these systems remain fragile: multi-agent frameworks exhibit fail-

ure modes including misaligned task interpretation, premature termination, and tool-call errors, often achieving only marginal improvements over single-agent baselines (Cemri et al., 2025). Current agent frameworks lack mechanisms for preserving execution context or recovering from intermediate failures. Toolformer (Schick et al., 2023) shows that models can learn to invoke external tools but also exposes how brittle tool usage is without explicit supervision or recovery mechanisms.

Recent work also explores richer communication channels between agents, including latent-space or hybrid symbolic-latent protocols that transmit internal model states rather than plain text (Du et al., 2025; Tang et al., 2025). These approaches improve bandwidth and reduce ambiguity but do not address reliability at the protocol level and rely on a rigid set of model assumptions.

Other work has examined long-term conversational memory and tool-use robustness, showing that LLMs struggle to maintain consistent state across long trajectories and frequently fail silently when tool calls break (Huang et al., 2025; Wang et al., 2025; Maharana et al., 2024; Valmeekam et al., 2023). Benchmarks of tool-use and failure recovery show that prompt-based context reconstruction is insufficient for ensuring correctness over multi-step workflows (Vuddanti et al., 2025; Kokane et al., 2024). Studies of faithful tool use show that agents often hallucinate successful actions or misuse APIs without explicit validation (Qin et al., 2024). Taken together, this work establishes that agentic systems fail in ways that prompt engineering and model scaling do not address. The missing piece is not a smarter model but a protocol that enforces correctness at the interaction boundary, independently of the model's internal reasoning.

## 3 Methodology

### 3.1 Design Principles for Reliable Agentic Protocols

Any viable session-layer solution, regardless of implementation, must satisfy the following properties. We use ACP as a running illustration, but the principles are protocol-agnostic. Although our running examples use a client-server framing for clarity, the principles apply equally to peer-to-peer agent topologies; the session abstraction is symmetric.

**Principle 1: Persistent Context Handles.** Any session-layer solution must make shared state explicit and addressable at the protocol level. Application-level context windows are insufficient because they are local to the agent and not shared across the protocol boundary. The protocol must establish a session identifier during a handshake phase, a shared handle allowing both parties to reference a consistent, externalized execution context that includes schema definitions, contract terms, and interaction history relevant to verification. TCP preserves the byte stream; it does not preserve which subgoal an agent was pursuing when the connection dropped. Persistent context handles fill this gap. *ACP implements this via a* `HELLO` *handshake establishing a* `SessionID`*. If the connection drops, the agent reconnects and resumes from the last protocol-managed checkpoint rather than restarting from scratch.*

**Principle 2: Structured Error Signaling.** A session layer must distinguish transport failures from semantic failures and must communicate the latter in a form that enables in-session correction rather than workflow restart. The protocol must define a structured, machine-interpretable error primitive that includes failure type and corrective context. Generic HTTP status codes (IANA, 2025) are insufficient because they carry no semantic information about what the agent should do next. *ACP returns an* `ERROR_OBJECT` *within the active session when output verification fails. The agent receives typed failure metadata (e.g.,* `SchemaMismatch`*,* `OracleFail`*) and generates a corrected response within the same session context.*

**Principle 3: Deterministic Task Bounds.** To safeguard agentic workflows against irreversible side effects, the communication protocol must establish explicit boundaries for verifying execution correctness prior to output commitment. This mechanism secures the external caller-responder interface without restricting the agent's internal reasoning traces or local API usage. This design acknowledges that while traditional two-phase commits enforce atomic data transitions, they cannot safeguard the semantic integrity of a multi-turn reasoning trajectory. Deterministic task bounds bridge this gap by applying transactional safety to agentic inference. Operationally, ACP monitors sessions between `BEGIN_TASK` and `END_TASK`, routes oracle-failed outputs into structured error feedback loops, and releases a response only upon verified success.

**Interoperability with Existing Frameworks.** The session-layer principles above are not intended to replace existing agent frameworks but to provide a standardized layer those frameworks can adopt. ACP-style stateful session management is deployable as a thin proxy around stateless MCP or A2A endpoints, incrementally introducing reliability guarantees without requiring protocol replacement. To make the architectural principles concrete, Appendix C describes one possible realization based on a Central Contract Store and a contract-negotiation lifecycle.

### 3.2 Experimental Setup

**Datasets.** We evaluate on four published benchmarks spanning distinct task domains:

- **APPS** (Hendrycks et al., 2021): competitive programming problems with unit-test oracles (introductory, interview, and competition difficulty tiers).
- **InterCode-Spider** (Yang et al., 2023): multi-turn text-to-SQL over the Spider corpus (Yu et al., 2018), where the agent iteratively refines SQL queries against a live database with execution feedback.
- **SWE-bench Verified** (Jimenez et al., 2024): real-world GitHub issue resolution requiring multi-file code patches verified by repository-specific test suites.
- **TravelPlanner** (Xie et al., 2024): constrained multi-phase travel planning requiring satisfaction of hard constraints (budget, time windows, transportation availability) verified by a constraint checker.

**Noise injection methodology.** We inject *semantic corruption* at the tool-output level: after the responding agent produces an output but before the requesting agent (or MCP's validator) evaluates it, a noise injector may corrupt the output with probability $\varepsilon \in \{0.0, 0.2, 0.4, 0.6\}$. Semantic corruption produces structurally valid outputs with plausible but incorrect content: off-by-one errors and negated boolean returns in code, flipped comparison operators and wrong aggregations in SQL, and swapped constraint orderings in planning tasks. Crucially, this corruption passes MCP's structural validation (field-presence checks) but fails ACP's execution oracle (unit tests, SQL execution, constraint satisfaction). The noise injector fires at most once per session (per-session cap), ensuring both protocols face the same number of corruption events regardless of how many internal retries each performs. Seed determinism is enforced via $\text{seed} = 42 + \text{trial} + \text{hash}(\text{scenario})$.

**Execution Oracles.** We evaluate behavioral correctness using protocol-agnostic execution oracles acting as deterministic developer-written test suites (*what* correct output looks like, not *how* to generate it). These oracles serve a dual role: as ACP's runtime verifier before output commitment, and as a symmetric post-hoc evaluator for both protocols:

- **APPS & SWE-bench:** Compiles and executes code against unit tests.
- **InterCode-Spider:** Executes queries on live databases, verifying output row set-intersection against ground-truth tables (allowing semantic equivalence).

**Fairness invariants.** Both protocols receive identical corrupted inputs from the same noise injector instance per trial. Both use the same underlying model with the same prompt templates and temperature settings. The same oracle evaluates both protocols' outputs. The only difference is whether the protocol invokes the oracle *before commitment* (ACP) or whether we apply it *post hoc for measurement only* (MCP).

**Models.** We evaluate across three model families spanning distinct capability tiers and providers: `gpt-4o-mini` (OpenAI), `claude-haiku-4-5` (Anthropic), and `gemini-2.5-flash` (Google). This cross-provider design tests whether ACP's guarantees are model-invariant or depend on provider-specific behaviors.

**Metrics.** We report four metrics:

- **FCR** (False Commitment Rate): Fraction of committed outputs that fail the execution oracle, i.e., $\text{FP}/(\text{FP} + \text{TN})$. This is the primary reliability metric.
- **TCR** (Total Correct Rate): fraction of all sessions that produce a correct committed output, i.e., $\text{TP}/N$.
- **Commit Rate**: fraction of sessions that commit any output (correct or not).
- **Token Cost**: mean tokens consumed per session, and effective tokens per correct commit.

**Hyperparameters.** Table 1 lists the protocol and model hyperparameters held constant across all experiments. All parameters are fixed before experimentation; no per-benchmark tuning is performed.

| Parameter | Value |
|---|---|
| LLM temperature | 0.3 |
| Max output tokens | 2,048 |
| Max retries | 3 |
| Max negotiation rounds | 3 |
| Noise injection | Semantic corruption at tool-output level |
| Per-session noise cap | 1 event per session |
| Trials/cell (InterCode) | 6 per $\varepsilon$/model |
| Trials/cell (sweep) | 10 per $\varepsilon$/protocol |

Table 1: Hyperparameter settings for all experiments.

# 4 Results

The experiments in this section test three claims: (1) stateless protocols commit semantically invalid outputs; (2) deterministic verification is required to eliminate false commitments; and (3) session-layer state and typed errors improve recovery once verification fails.

| Model | ACP FCR ↓ | MCP FCR ↓ | ACP TCR ↑ | MCP TCR ↑ |
|---|---|---|---|---|
| `gpt-4o-mini` | **0.000** | 0.392 | 0.083 | 0.608 |
| `claude-haiku-4-5` | **0.000** | 0.400 | 0.900 | 0.600 |
| `gemini-2.5-flash` | **0.000** | 0.267 | 0.700 | 0.733 |

Table 2: Cross-model comparison on InterCode-Spider SQL at $\varepsilon = 0.4$. ACP achieves FCR = 0.000 across all three model families. MCP FCR is consistently high regardless of model capability. More capable models (Claude Haiku 4.5) achieve *higher* TCR under ACP, showing the protocol channels capability rather than suppressing it.

## 4.1 Cross-Model Reliability

To test whether ACP's reliability depends on a specific model family, we run the full InterCode-Spider evaluation across three providers: OpenAI (`gpt-4o-mini`, 480 trials), Anthropic (`claude-haiku-4-5`, 480 trials), and Google (`gemini-2.5-flash`, 480 trials). Table 2 reports results at $\varepsilon = 0.4$.

Three observations emerge. First, ACP FCR = 0.000 is **model-invariant**: it holds identically across OpenAI, Anthropic, and Google models because FCR is enforced by the execution oracle, not by model capability. Second, MCP FCR is consistently high (0.27–0.40) regardless of the underlying model, confirming that the reliability gap is structural. Third, more capable models achieve *higher* TCR under ACP: Claude Haiku 4.5 achieves TCR = 0.900 (vs. 0.083 for GPT-4o-mini) while maintaining zero false commitments. **The protocol amplifies model capabilities by directing them towards verified outputs.** The cross-model FCR comparison is visualized in Appendix Figure 6.

## 4.2 Baseline Comparison

To isolate which components of ACP's design contribute to its reliability advantage, we evaluate two baselines and two ablations:

**Baselines.**

1. **MCP (stateless)**: The standard stateless baseline. No verification, no retry.
2. **Self-Refine** (Madaan et al., 2023): The model generates output, critiques its own output in a separate LLM call, and revises based on self-feedback (up to 3 rounds), always commiting without any external oracle. This tests whether self-critique can substitute for protocol-level verification.
3. **LLM-as-Judge + Retry**: The model generates output; a separate LLM call evaluates correctness and provides structured feedback. On rejection, the model retries with judge feedback (up to 3 retries). This represents the pattern used by most LangChain/LangGraph retry harnesses.
4. **MCP + oracle + retry**: MCP augmented with the same execution oracle used by ACP. On failure, the agent retries with a fresh prompt (no session history). This isolates whether the advantage is the oracle alone or *stateful* error recovery.

**Ablations.**

1. **ACP − typed errors**: ACP with the execution oracle intact but error responses replaced by generic "output incorrect, please retry" messages without typed error codes, field-level feedback, or correction hints. This isolates the contribution of structured error signaling (Principle 2).
2. **ACP − oracle (schema-only)**: ACP with schema validation only where we only check that required fields are present and correctly typed. No execution oracle is present, testing whether structural validation alone is sufficient or whether semantic verification is necessary.

We report the baseline comparison in Table 3, revealing a clear hierarchy. Self-Refine reduces FCR from 0.400 to 0.100—self-critique catches many errors but not all, because the same model that generated wrong SQL also evaluates it. LLM-as-Judge performs worse (FCR = 0.167) despite using a separate evaluation call, because

| Configuration | FCR↓ | TCR↑ | Tokens |
|---|---|---|---|
| *Framework baselines (Claude Haiku, $\varepsilon = 0.4$)* | | | |
| MCP (stateless) | 0.400 | 0.600 | 838 |
| Self-Refine | 0.100 | 0.900 | 2,450 |
| LLM-as-Judge + Retry | 0.167 | 0.833 | 3,172 |
| *Protocol ablations (gpt-4o-mini, $\varepsilon = 0.2$)* | | | |
| ACP − oracle (schema only) | 0.615 | 0.527 | 2,489 |
| ACP − typed errors | 0.000 | 0.655 | 3,168 |
| MCP + oracle + retry | 0.000 | 0.855 | 426 |
| *Full protocol* | | | |
| **ACP (full)** | **0.000** | **0.900** | 4,832 |

Table 3: Baseline and ablation comparison on InterCode-Spider SQL ($\varepsilon = 0.4$, `claude-haiku-4-5` for Self-Refine/LLM-Judge; `gpt-4o-mini` for ablations at $\varepsilon = 0.2$). Only methods with deterministic oracle verification achieve FCR = 0.000. Self-critique and LLM-judge reduce but do not eliminate false commitments. **Takeaway:** deterministic verification eliminates false commits; ACP improves recovery by adding typed errors and session context.

LLM-based judgment misses semantic errors that execution-based verification catches. Critically, **neither method achieves FCR = 0.000 at any noise level**, and only methods with deterministic oracle verification (MCP+oracle+retry and ACP) achieve FCR = 0.000. Among these, ACP's typed error feedback yields higher TCR (0.900) than MCP+oracle+retry (0.855) because structured error context enables directed self-correction rather than blind retry. Figure 1 visualizes this hierarchy.

These results show that deterministic verification is necessary for eliminating false commitments, but not sufficient for efficient recovery. MCP+oracle+retry provides robust rejection of faulty outputs, but it falls short by lacking persistent session context and typed recovery signals. ACP's key contribution lies in the protocol mechanisms that transform verification failures into corrective actions.

### 4.3 Multi-Step Error Cascade

The single-step results establish that ACP prevents false commitments in isolation. The more consequential question is what happens in multi-step workflows where an error at step $k$ propagates to steps $k+1, \ldots, N$.

Under a stateless protocol with per-step corruption probability $\varepsilon$, the probability that a $k$-step pipeline produces a correct final output is at most $(1-\varepsilon)^k$, because each step independently risks committing a corrupted intermediate. A session-layer protocol that verifies each step before passing its output downstream maintains correctness at each stage.

We evaluate a 4-step code-generation chain where each step depends on all prior outputs (10 trials per cell, summarized in Table 4). ACP yield is flat at $k = 1$–$3$ regardless of $\varepsilon$: the session verifier gates every commit. The ACP–MCP gap grows with depth, at $k = 4$, $\varepsilon = 0.6$, ACP yield = 1.000 vs. MCP yield = 0.100 (90-point gap). MCP's error compounds exponentially, consistent with the theoretical $(1-\varepsilon)^k$ lower bound. Figure 2 visualizes the yield degradation.

| Protocol | $\varepsilon$ | $\hat{Y}_{k=1}$ | $\hat{Y}_{k=2}$ | $\hat{Y}_{k=3}$ | $\hat{Y}_{k=4}$ | Theo. |
|---|---|---|---|---|---|---|
| ACP | 0.0 | 1.000 | 1.000 | 1.000 | 0.900 | — |
| | 0.2 | 1.000 | 1.000 | 1.000 | 0.900 | — |
| | 0.4 | 1.000 | 1.000 | 1.000 | 0.900 | — |
| | 0.6 | 1.000 | 1.000 | 1.000 | 1.000 | — |
| MCP | 0.0 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| | 0.2 | 0.700 | 0.500 | 0.500 | 0.300 | 0.410 |
| | 0.4 | 0.400 | 0.200 | 0.200 | 0.200 | 0.130 |
| | 0.6 | 0.400 | 0.200 | 0.100 | 0.100 | 0.026 |

Table 4: Pipeline yield $\hat{Y}_k$ at depths $k \in \{1, 2, 3, 4\}$ for ACP and MCP (10 trials per cell). ACP yield is flat at $k = 1$–$3$ regardless of $\varepsilon$; MCP yield degrades exponentially with depth. Theo. column shows $(1-\varepsilon)^4$, the iid lower bound for MCP.

### 4.4 Failure Stress-Test: TravelPlanner

TravelPlanner exceeds the capability of the evaluated model under both protocols, yielding TCR = 0.000. We therefore do not use it as evidence of improved task completion. ACP enforces verification by testing failure signaling: while MCP may commit errors, ACP aborts all unsatisfied plans, resulting in an FCR of 0.80–1.00.

### 4.5 Key Findings

**Finding 1: ACP eliminates false commitments across all evaluated settings, including cases where the underlying model fails to solve the task.** Across all conditions tested, ACP achieves FCR $= 0.000$: no session that reaches the COMMITTED state contains an output that fails the execution oracle. This holds identically across three model families and four benchmarks (Tables 2, 3). By design, ACP's verification loop prevents commitment of unverified outputs regardless of which model produces them. This holds even at extreme noise: at $\varepsilon = 1.0$ (every tool response corrupted), MCP commits 100% false commitments while ACP commits 100% true positives after in-session recovery (Appendix A.5).

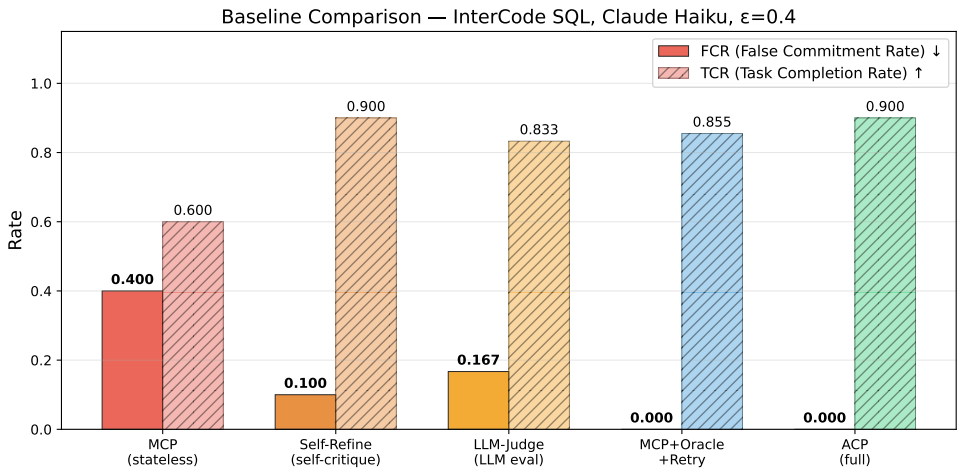


Figure 1: Baseline comparison on InterCode-Spider SQL at $\varepsilon = 0.4$ (Claude Haiku 4.5): FCR (left) and TCR (right) across five verification strategies. Self-critique and LLM-judge reduce but do not eliminate false commitments. Only deterministic oracle verification achieves FCR = 0.000. ACP's session-level infrastructure (typed errors, persistent context) maximizes TCR among oracle-based methods.

**Finding 2: False commitments arise from task difficulty, not just injected noise.** On SWE-bench Verified (419 trials with oracle context, `gpt-4o-mini`), ACP achieves FCR $= 0.000$ with 23 correct commits out of 210 sessions (TCR $=$ 0.110). MCP commits 202 outputs, of which 121 fail the test suites (FCR $= 0.599$), while achieving TCR $= 0.388$. ACP correctly rejects wrong patches, committing only verified-correct ones. On TravelPlanner (180 trials), gpt-4o-mini cannot satisfy the planning constraints under either protocol (TCR $= 0.000$), but the protocols diverge in how they handle this failure: ACP correctly aborts every session (FCR $= 0.000$), while MCP commits catastrophically wrong plans—with 15–35 constraint violations each—100% of the time (FCR $= 1.000$). This "silent catastrophe" is the most dangerous failure mode: the downstream consumer receives a confidently committed but fundamentally broken output with no signal that verification failed.

**Finding 3: ACP's reliability does not come at the expense of correctness.** With Claude Haiku 4.5, ACP achieves TCR = 0.900 at $\varepsilon = 0.4$ with zero false commitments (Table 2). ACP does not suppress model capability but rather *requires* that capability be channeled through verified outputs.

**Finding 4: The formalizability boundary.** When no deterministic oracle exists and ACP falls back to an LLM-as-judge verifier, ACP's advantage collapses. **Session-layer protocols are the right choice when correctness is formalizable; for open-ended tasks, neither protocol provides verification guarantees.** The Agentic Web's primary use cases—code generation, data pipelines, tool orchestration, constraint-satisfaction planning—all admit deterministic oracles.

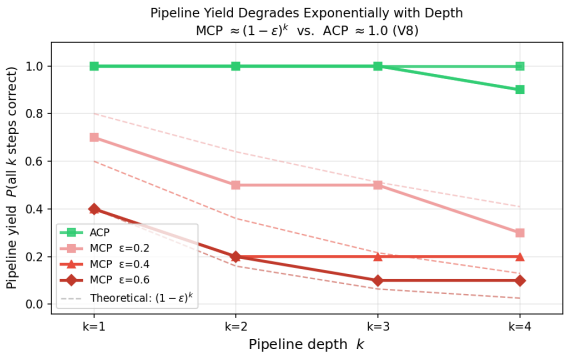


Figure 2: Pipeline yield vs. depth for ACP and MCP at $\varepsilon \in \{0.0, 0.2, 0.4, 0.6\}$ ($k = 1$–4, 10 trials per cell). ACP maintains near-perfect yield; MCP yield degrades exponentially, matching $(1-\varepsilon)^k$. At $k = 4$, $\varepsilon = 0.6$: ACP yield = 1.000 vs. MCP yield = 0.100.

**Token cost breakeven.** ACP's per-session overhead (5–10× more tokens than MCP) is offset by MCP's exponentially growing rework cost in multi-step pipelines. Figure 4 shows the effective token cost per correct pipeline completion as a function of pipeline depth $K$. At $\varepsilon = 0.4$, ACP becomes cheaper than MCP at $K \geq 5$ steps; at $\varepsilon = 0.8$, the crossover occurs at just $K \geq 2$ steps. Beyond these depths, MCP's false commits cascade through downstream steps, requiring full rework—making the "cheap" stateless protocol more expensive in total compute than the "expensive" session-layer protocol. Additional per-benchmark tables and fig-

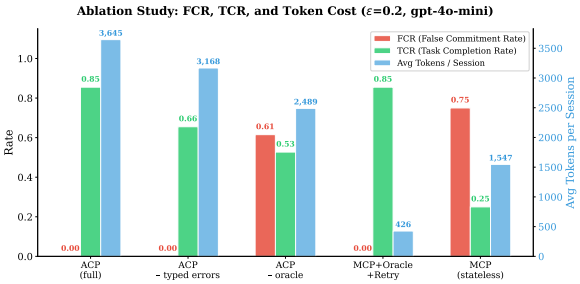


Figure 3: Ablation study shows contribution of individual ACP components. Removing the execution oracle raises FCR from 0.000 to 0.615; removing typed errors preserves FCR but reduces TCR by 20 points. Self-Refine and LLM-Judge reduce but cannot eliminate false commitments.

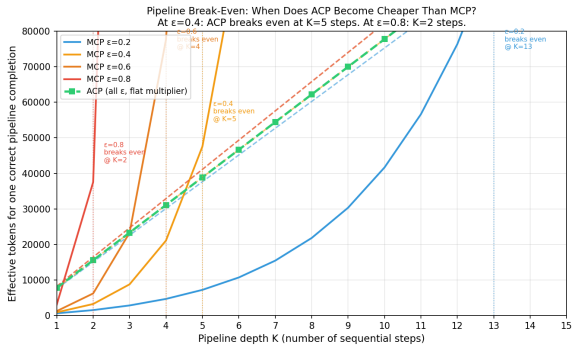


Figure 4: Pipeline cost breakeven: ACP becomes cheaper than MCP in total compute at $K \geq 5$ steps for $\varepsilon = 0.4$ and $K \geq 2$ steps for $\varepsilon = 0.8$, because MCP's rework cost from false commits grows exponentially with depth.

ures are reported in Appendix A.

## 5 Discussion

**The role of model capability.** Our cross-model results (Table 2) provide direct evidence that model capability improvements reduce the rate at which agents produce incorrect outputs from their own reasoning, but do not address cases where an agent receives a corrupted input from an external tool. MCP FCR remains 0.27–0.40 regardless of model family, confirming that the reliability gap is structural. Even granting that model-generated errors approach zero, multi-agent systems involve coordination across heterogeneous agents, tool providers, and execution environments that may fail for reasons unrelated to model capability.

**Protocol vs. middleware.** An alternative is to implement reliability at the middleware layer using frameworks like LangChain, AutoGen, or custom retry harnesses. Our ablation results (seen in Figure 3 and Table 3) illustrate that while middleware techniques like Self-Refine and LLM-as-Judge reduce false commitments, they fail to eliminate them entirely because they lack access to the deterministic execution oracles enforced at the protocol boundary. The historical analogy here is instructive: before TCP standardized reliable byte-streaming, every application was forced to implement its own ad-hoc retry and ordering logic, leading to predictable interoperability failures.

**Oracle requirements.** ACP's zero-FCR guarantee depends on the availability of a deterministic execution oracle. Developers already write the verification artifacts that ACP uses as oracles: unit tests, integration tests, SQL validation queries, and constraint specifications. ACP gives these existing artifacts a protocol-level enforcement point. The ablation study (Table 3) quantifies this: removing the oracle entirely raises FCR from 0.000 to 0.615. Further discussion of oracle spectrum and provenance appears in Appendix B.

**Scope and boundaries.** Stateless protocols remain preferable for read-only discovery, idempotent retrieval, and one-shot inference. The reliability argument applies specifically to non-idempotent, multi-step workflows where the false commitment failure mode is most consequential. The formalizability boundary (Finding 4) further refines this scope.

## 6 Conclusion

The industry is converging on stateless, request-response protocols for communication in the *Agentic Web* (§1). This paper has shown stateless paradigms cannot prevent false commitments under semantic noise.

Our empirical evaluation confirms this gap across multiple dimensions. Under controlled fault injection, ACP achieves FCR = 0.000 at all noise levels while MCP's FCR increases with $\varepsilon$, and in multi-step pipelines MCP's errors compound while ACP maintains verified outputs. Cross-model evaluation further shows that ACP's zero-FCR guarantee is model-invariant, while more capable models achieve higher TCR under ACP.

The central claim is bounded: protocol-level reliability guarantees are both available and necessary for agentic tasks with formalizable correctness criteria. **Reliability must be a protocol-level requirement, not a property delegated to the layers above or below.**

## Limitations

Our evaluation spans four published benchmarks (APPS, InterCode-Spider, SWE-bench Verified, TravelPlanner) and three model families (GPT-4o-mini, Claude Haiku 4.5, Gemini 2.5 Flash), but several limitations remain.

**Oracle dependency.** ACP's zero false-positive guarantee depends on the availability of a deterministic execution oracle. With an LLM-as-judge verifier, ACP's FCR rose to 0.333 (worse than MCP's 0.000) and its token overhead increased to 23–32×, demonstrating that when no oracle exists, ACP's advantage is lost (Finding 4). We bound this claim accordingly: protocol-level verification is the optimal design choice specifically when execution correctness is formalizable. Crucially, the core domains of the Agentic Web (code generation, data pipelines, SQL execution, tool orchestration, and constrained planning) all readily admit deterministic oracles.

**Token overhead.** ACP consumes approximately 16× more tokens than MCP per session in the low-noise regime ($\varepsilon = 0.0$). However, the relevant metric for reliability-critical systems is cost per *correct* commit. At $\varepsilon = 0.8$, the effective cost gap narrows to 2.7× (Table 10). In multi-step pipelines, ACP becomes cheaper in total compute at $K \geq 5$ steps for $\varepsilon = 0.4$ and $K \geq 2$ steps for $\varepsilon = 0.8$.

**Synthetic Noise Model.** Our noise injection is programmatic, corrupting tool outputs deterministically rather than relying on stochastic, real-world tool failures. To ensure validity, we utilize *semantic* corruption (e.g., off-by-one errors, negated booleans, flipped comparison operators) that directly mirrors observed failure modes in deployment (Qin et al., 2024; Kokane et al., 2024). Our evaluated noise levels ($\varepsilon \in \{0.0\text{–}1.0\}$) bracket empirically documented real-world tool failure rates of 10–40% (Cemri et al., 2025).

**Benchmark coverage.** We do not evaluate on open-ended generation, summarization, dialogue, or creative tasks. Domains where no deterministic oracle exists and where our formalizability boundary result predicts no ACP advantage. This is not a gap but a consequence of the bounded claim.

**Correctness–completeness tradeoff.** ACP's zero-FCR comes at the cost of a lower commit rate: sessions that would be false commitments under MCP are instead correctly classified as failures (ABORTED). This tradeoff is appropriate for high-stakes, multi-step workflows with irreversible side effects; a correctly identified failure is strictly preferable to a silent false commitment.

**Additional limitations.** We inject only one class of semantic noise; reasoning errors, coordination failures, adversarial prompt injection, and environmental uncertainty are not evaluated. SWE-bench results use the Oracle split (relevant source files provided); full-repository navigation is not evaluated. The pipeline depth experiment evaluates code-generation chains; multi-domain pipelines remain untested. The token cost analysis does not account for infrastructure overhead of maintaining session state at scale.

## Ethical Considerations

This work evaluates protocol-level reliability for multi-agent LLM systems using published benchmarks (APPS, InterCode-Spider, SWE-bench Verified, TravelPlanner) and does not involve human subjects, personally identifiable information, or sensitive data. The noise injection methodology is synthetic and does not simulate adversarial attacks against real systems. We acknowledge that deploying verification-gated protocols in production could introduce failure modes (e.g., excessive aborts on valid outputs with strict oracles) that require careful calibration. The token cost overhead of session-layer verification has environmental implications that should be weighed against the cost of propagating false commitments through downstream systems.

## References


Mert Cemri and 1 others. 2025. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2501.01234*.

Reuven Cohen, Matty Kadosh, Alan Lo, and Qasem Sayah. 2020. Lb scalability: Achieving the right balance between being stateful and stateless. *Preprint*, arXiv:2010.13385.

Yilun Du and 1 others. 2025. Interlat: Inter-agent latent space communication. *arXiv preprint arXiv:2502.00321*.

Roy T. Fielding, Richard N. Taylor, Justin R. Erenkrantz, Michael M. Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. 2017. Reflections on the rest architectural style and "principled design of the modern web architecture" (impact paper award). In *Proceedings of the 2017 11th Joint Meeting on Foundations of*

*Software Engineering*, ESEC/FSE 2017, page 4–14, New York, NY, USA. Association for Computing Machinery.

Google. 2024. Agent-to-agent (A2A) communication standard. Stateless discovery standard discussed in Section 1 and 4.

Taicheng Guo and 1 others. 2024. Large language model based multi-agent systems: A survey. *arXiv preprint arXiv:2402.01680*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS)*.

Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. 2025. Model context protocol (mcp): Landscape, security threats, and future research directions. *Preprint*, arXiv:2503.23278.

Yifan Huang and 1 others. 2025. On the failure of latent state persistence in transformers. *arXiv preprint arXiv:2501.07891*.

IANA. 2025. Hypertext transfer protocol (http) status code registry.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*.

Aishwarya Kokane and 1 others. 2024. Spectool: Benchmarking tool-use errors in large language models. *arXiv preprint arXiv:2403.01987*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangyuan Ding, Kaiwen Men, Kejuan Yang, and 1 others. 2023. AgentBench: A comprehensive benchmark to evaluate LLMs as agents. *arXiv preprint arXiv:2308.03637*. The baseline for your Section 4.2 evaluation.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Skilling, Chengyuan Gao, Shrimai Biswal, Uri Alon, David Roberts, Jiarui Yang, Shashank Meijer, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*. Supports the idea of agents needing feedback loops to self-correct.

Adarsh Maharana and 1 others. 2024. Locomo: Long conversational memory benchmark. *arXiv preprint arXiv:2402.09845*.

OpenAI. 2024. Swarm: An educational framework exploring multi-agent orchestration. Cited regarding handoff functions and terminal exceptions.

Yujia Qin and 1 others. 2024. On faithful tool use in large language models. *arXiv preprint arXiv:2403.08629*.

Timo Schick and 1 others. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.

Yujie Tang and 1 others. 2025. Augmenting multi-agent communication with state delta encoding. *arXiv preprint arXiv:2501.04567*.

Karthik Valmeekam and 1 others. 2023. Large language models still can't plan. *arXiv preprint arXiv:2305.19452*.

Sandeep Vuddanti and 1 others. 2025. Paladin: Self-correcting llm agents for tool failures. *arXiv preprint arXiv:2501.05678*.

Zhen Wang and 1 others. 2025. Long-term dialogue memory via recursive summarization. *arXiv preprint arXiv:2502.01111*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Chuwei Li, Li Zhang, Michael Rowe, Ahmed Hassan Bennett, and 1 others. 2023. AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Kai Rui, Yuwei Zhu, Yanghua Xiao, Jian Zhang, and Yu Yang. 2024. TravelPlanner: A benchmark for real-world planning with language agents. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. InterCode: Standardizing and benchmarking interactive coding with execution feedback. In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*.

Shunyu Yao and 1 others. 2023. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921.

Yujia Zheng, Zhuokai Zhao, Zijian Li, Yaqi Xie, Mingze Gao, Lizhu Zhang, and Kun Zhang. 2025. Thought communication in multiagent collaboration. *Preprint*, arXiv:2510.20733.

Jiaru Zou, Xiyuan Yang, Ruizhong Qiu, Gaotang Li, Katherine Tieu, Pan Lu, Ke Shen, Hanghang Tong, Yejin Choi, Jingrui He, James Zou, Mengdi Wang, and Ling Yang. 2025. Latent collaboration in multi-agent systems. *Preprint*, arXiv:2511.20639.

| Protocol | $\varepsilon$ | FCR↓ | TCR↑ | Commit | Tokens |
|---|---|---|---|---|---|
| *Introductory* ($n = 20$) | | | | | |
| ACP | 0.0 | 0.000 | 0.250 | 0.250 | 7,830 |
| ACP | 0.4 | 0.000 | 0.250 | 0.250 | 7,801 |
| MCP | 0.0 | 0.700 | 0.300 | 1.000 | 996 |
| MCP | 0.4 | 0.750 | 0.250 | 1.000 | 1,547 |
| *Medium* ($n = 8$) | | | | | |
| ACP | 0.0 | 0.000 | 0.375 | 0.375 | 6,800 |
| ACP | 0.4 | 0.000 | 0.375 | 0.375 | 6,994 |
| MCP | 0.0 | 0.750 | 0.250 | 1.000 | 752 |
| MCP | 0.4 | 0.750 | 0.250 | 1.000 | 1,293 |
| *Competition* ($n = 12$) | | | | | |
| ACP | 0.0 | 0.000 | 0.167 | 0.167 | 8,518 |
| ACP | 0.4 | 0.000 | 0.167 | 0.167 | 8,338 |
| MCP | 0.0 | 0.667 | 0.333 | 1.000 | 1,159 |
| MCP | 0.4 | 0.750 | 0.250 | 1.000 | 1,717 |

Table 5: APPS code generation: ACP vs. MCP under semantic noise ($\varepsilon$). $n$ = sessions per condition. ACP never commits a false output; MCP commits all outputs, of which 67–75% are incorrect.

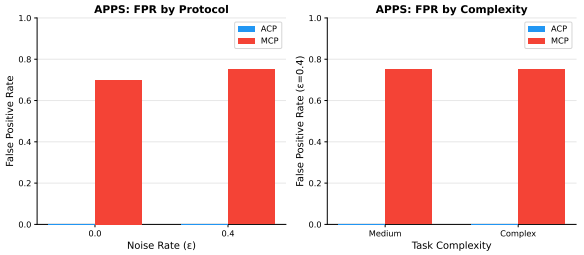


Figure 5: APPS benchmark: FCR by protocol and task complexity (medium, competition at $\varepsilon = 0.4$). ACP bars are at zero (FCR = 0.000) across all conditions; MCP bars (red) show FCR = 0.700–0.750, indicating that 70–75% of MCP's committed outputs fail the execution oracle.

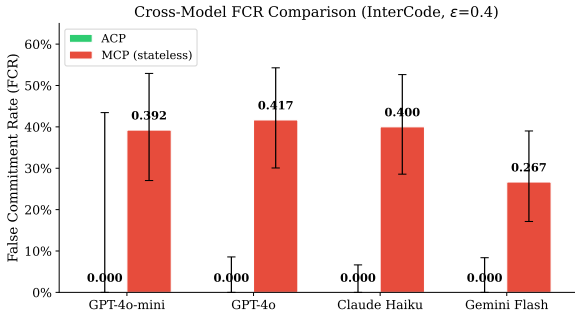


Figure 6: Cross-model FCR comparison on InterCode-Spider SQL at $\varepsilon = 0.4$ (1,440 trials: 480 per model). ACP achieves FCR = 0.000 across all three model families; MCP FCR ranges from 0.267 to 0.400 regardless of model capability.

# A Detailed Experimental Results

This appendix reports the full per-benchmark tables, figures, and analyses that support the key findings in §4.

## A.1 APPS Code Generation

Table 5 reports results on 40 APPS problems (20 introductory, 8 medium, 12 competition-level) under $\varepsilon \in \{0.0, 0.4\}$ with `gpt-4o-mini`. Across all conditions, ACP achieves FCR $= 0.000$: every committed output passes unit tests. MCP commits every output (commit rate $= 1.0$) but 70–75% of those commits are false commitments that fail the oracle.

The key observation is that ACP's lower commit rate reflects the protocol working as designed: outputs that would be false commit under MCP are instead correctly classified as failures (TN) and the session terminates in ABORTED rather than silently committing a wrong answer. ACP's TCR is comparable to or matches MCP's TCR, meaning ACP does not sacrifice correctness—it sacrifices only false confidence.

## A.2 InterCode-Spider SQL: Per-Model Details

Table 6 reports results on 20 InterCode-Spider tasks under $\varepsilon \in \{0.0, 0.2, 0.4, 0.6\}$ (480 trials with `gpt-4o-mini`). ACP maintains FCR $= 0.000$ across all noise levels. MCP's FCR increases from 0.158 at $\varepsilon = 0.0$ to 0.433 at $\varepsilon = 0.6$, committing SQL queries with flipped comparison operators (e.g., `ASC` → `DESC`) that return plausible but incorrect result sets. Notably, MCP shows nonzero FCR even at $\varepsilon = 0.0$, indicating that the model's own errors are sufficient to produce false commitments without any injected corruption.

Table 7 reports the same 20 tasks with `claude-haiku-4-5` (480 trials). The headline result: ACP achieves TCR = 0.900 at $\varepsilon = 0.4$ with zero FCR—the strongest single result in our evaluation. Claude Haiku's higher base capability translates directly into higher TCR under ACP, while MCP's FCR follows the same $\varepsilon$-proportional pattern regardless of model capability.

## A.3 SWE-bench Verified

Table 8 reports results on SWE-bench Verified with oracle context (419 trials, `gpt-4o-mini`). Using the SWE-bench Oracle split, which provides the relevant source files to the model, ACP achieves FCR $= 0.000$: all 23 committed patches pass the repository test suites. MCP commits 202 patches, of which 121 fail (FCR $= 0.599$). This demon-

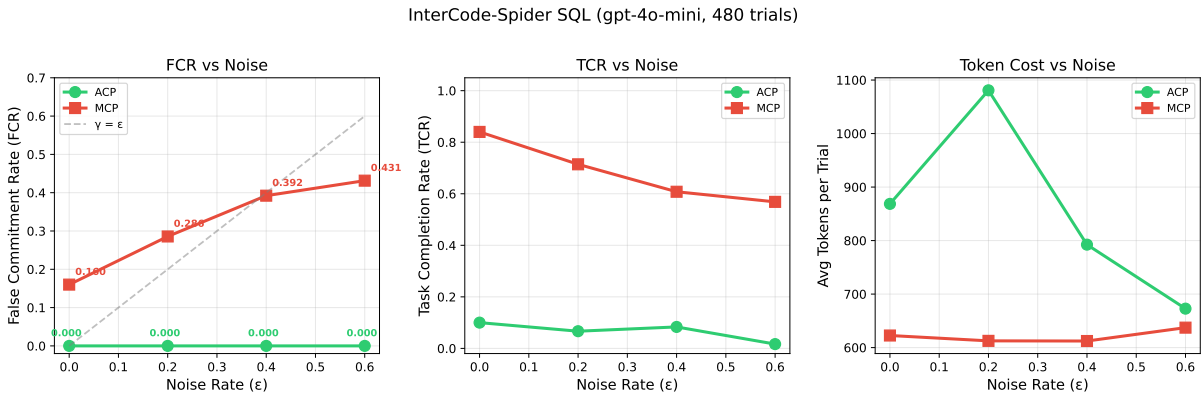


Figure 7: InterCode-Spider SQL results across noise levels (480 trials, 20 tasks across 13 databases): (a) false commitment rate, (b) task completion rate, (c) commit rate. ACP maintains FCR = 0.000 at all noise levels; MCP FCR increases proportionally with $\varepsilon$

| Protocol | $\varepsilon$ | FCR↓ | TCR↑ | Commit | Tokens |
|---|---|---|---|---|---|
| ACP | 0.0 | 0.000 | 0.100 | 0.100 | 5,389 |
| ACP | 0.2 | 0.000 | 0.067 | 0.067 | 5,294 |
| ACP | 0.4 | 0.000 | 0.083 | 0.083 | 5,399 |
| ACP | 0.6 | 0.000 | 0.017 | 0.017 | 5,610 |
| MCP | 0.0 | 0.158 | 0.842 | 1.000 | 439 |
| MCP | 0.2 | 0.258 | 0.742 | 1.000 | 432 |
| MCP | 0.4 | 0.392 | 0.608 | 1.000 | 439 |
| MCP | 0.6 | 0.433 | 0.567 | 1.000 | 444 |

Table 6: InterCode-Spider SQL with `gpt-4o-mini` (480 trials: 20 tasks $\times$ 6 trials $\times$ 4 $\varepsilon$ levels). ACP: zero FCR at all noise levels. MCP FCR increases proportionally with $\varepsilon$.

| Protocol | $\varepsilon$ | FCR↓ | TCR↑ | Commit |
|---|---|---|---|---|
| ACP | 0.0 | 0.000 | 0.633 | 0.633 |
| ACP | 0.2 | 0.000 | 0.850 | 0.850 |
| ACP | 0.4 | 0.000 | 0.900 | 0.900 |
| ACP | 0.6 | 0.000 | 0.917 | 0.917 |
| MCP | 0.0 | 0.050 | 0.950 | 1.000 |
| MCP | 0.2 | 0.217 | 0.783 | 1.000 |
| MCP | 0.4 | 0.400 | 0.600 | 1.000 |
| MCP | 0.6 | 0.467 | 0.533 | 1.000 |

Table 7: InterCode-Spider SQL with `claude-haiku-4-5` (480 trials: 20 tasks $\times$ 6 trials $\times$ 4 $\varepsilon$ levels). ACP achieves TCR = 0.900 at $\varepsilon = 0.4$ with zero false commitments—substantially higher than `gpt-4o-mini` (Table 6), demonstrating that more capable models achieve higher correctness under ACP.

strates that with sufficient context, models can solve real software engineering tasks, and ACP's verification loop ensures only correct patches are committed.

### A.4 TravelPlanner

On TravelPlanner (180 trials with `gpt-4o-mini`), both protocols achieve TCR = 0.000: the task exceeds the model's capability. The critical difference is how each protocol handles this failure. ACP correctly aborts all sessions (FCR = 0.000), accurately signaling that no correct plan was produced. MCP commits incorrect plans 80–100% of the time (FCR = 0.800–1.000), presenting wrong itineraries as correct outputs. This is the "silent catastrophe" failure mode: the downstream consumer receives a confidently committed but incorrect plan with no signal that verification failed. When both protocols lack the capability to solve a task, the reliable protocol *admits* failure; the stateless protocol *conceals* it.

| Protocol | FCR↓ | TCR↑ | Commit | Tokens | Retries |
|---|---|---|---|---|---|
| ACP | 0.000 | 0.110 | 0.110 | 17,165 | 4.0 |
| MCP | 0.599 | 0.388 | 0.967 | 7,857 | 0.5 |

Table 8: SWE-bench Verified with oracle context (419 trials, `gpt-4o-mini`, $\varepsilon \in \{0.0, 0.4\}$). ACP commits only verified-correct patches (FCR $= 0.000$, 23 TPs). MCP commits 60% incorrect patches.

### A.5 Noise Sweep across Synthetic Scenarios

To characterize the reliability gap at scale under controlled conditions, we additionally run a controlled noise sweep: 11 synthetic benchmark scenarios (HumanEval code generation, Spider SQL, ToolBench tool use) $\times$ 10 trials per ($\varepsilon$, protocol) at $\varepsilon \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$, totaling 550 trials per protocol. Table 9 reports the results. ACP achieves FCR = 0.000 at every noise level; MCP's FCR scales proportionally with $\varepsilon$ (ratio 0.95–1.11$\times$), confirming that structural-only validation provides zero semantic protection. At

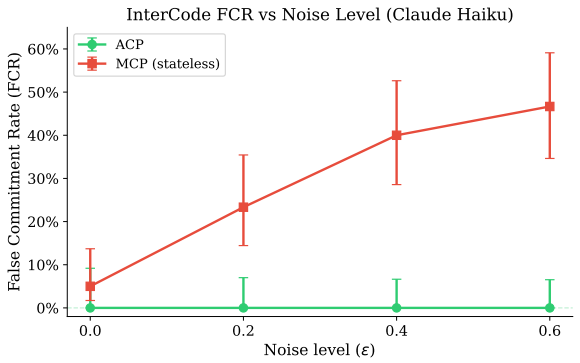


Figure 8: InterCode-Spider with Claude Haiku 4.5 (480 trials): FCR vs. $\varepsilon$. ACP maintains FCR = 0.000 at all noise levels; MCP FCR increases from 0.050 to 0.467. The pattern is identical to GPT-4o-mini (Figure 7), confirming model-invariance.

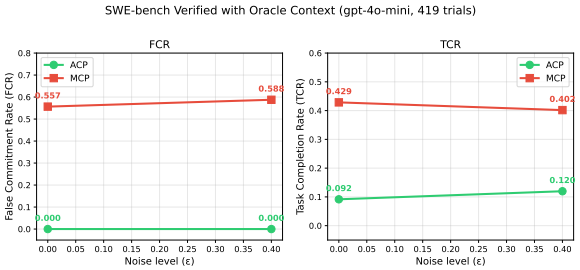


Figure 9: SWE-bench Verified with oracle context (419 trials, `gpt-4o-mini`): FCR vs. $\varepsilon$. ACP achieves FCR = 0.000; MCP FCR = 0.588.

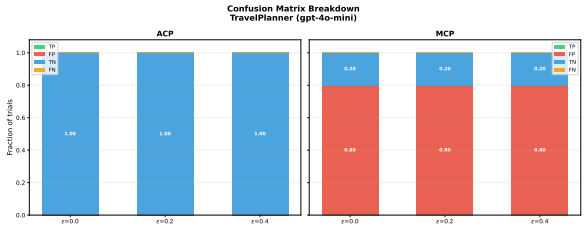


Figure 10: TravelPlanner confusion breakdown (180 trials, `gpt-4o-mini`): ACP correctly classifies all sessions as TN (true negative — task unsolvable, correctly aborted). MCP commits incorrect plans as false commits across all noise levels.

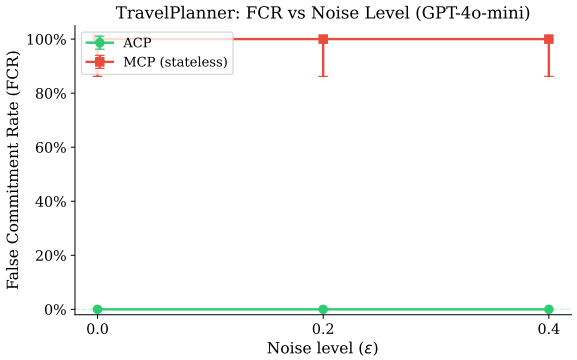


Figure 11: TravelPlanner: FCR vs. $\varepsilon$ (180 trials, `gpt-4o-mini`). Both protocols achieve TCR = 0.000 (task exceeds model capability), but MCP commits incorrect plans at FCR = 0.800–1.000 while ACP correctly aborts all sessions.

$\varepsilon = 0.0$, both protocols achieve TCR = 1.000, confirming no capability penalty from ACP's session layer on clean inputs. At $\varepsilon = 1.0$, MCP's TCR = 1.000 reflects 100% false commitment; ACP's TCR = 1.000 reflects 100% true positives after in-session recovery.

### A.6 Token Cost Analysis

ACP consumes 5–10× more tokens per session than MCP (e.g., 7,801 vs. 1,547 on APPS introductory at $\varepsilon = 0.4$). This overhead arises from contract negotiation, multi-round verification, and retry with session context. However, the effective cost per *correct* commit favors ACP when MCP's false commitments are accounted for. Table 10 shows the effective token cost from the noise sweep: ACP's raw 16× overhead reduces to 2.7× per correct commit at $\varepsilon = 0.8$, because 83.6% of MCP's sessions at that noise level buy nothing useful. At $\varepsilon = 1.0$, MCP produces zero correct commits; its effective cost is undefined. MCP's token cost is indistinguishable between sessions that commit correct vs. incorrect answers ($\approx$ 477–515 tokens in both cases)—MCP has no internal signal that it is about to commit a wrong answer. Accounting for downstream propagation cost, ACP becomes cheaper in total pipeline compute at $K \geq 5$ steps for $\varepsilon = 0.4$ and $K \geq 2$ steps for $\varepsilon = 0.8$. Figure 12 visualizes the token cost breakdown.

### A.7 InterCode SQL Chain Experiments

To validate the pipeline depth effect on real benchmark tasks, we chain InterCode-Spider SQL queries into $k$-step pipelines ($k \in \{2, 3\}$, 960 total trials) where each step's output serves as context for the next. Table 11 reports the results; Figure 13 visualizes the yield curves. At moderate noise ($\varepsilon = 0.4$), ACP maintains a 2–2.2× yield advantage; at high noise ($\varepsilon = 0.6$), the advantage grows to 3–5.3× as MCP's error cascade compounds across steps. MCP yield closely tracks the theoretical $(1 - \varepsilon)^k$ prediction, confirming that each step fails independently without protocol-level verification.

The multi-step setting is where the distinction be-

Table 9: Controlled noise sweep: FCR and TCR for ACP vs. MCP under varying semantic noise ($\varepsilon$). 11 scenarios $\times$ 10 trials per cell (550 trials per protocol). $\varepsilon = 1.0$ from validation run (7 scenarios $\times$ 5 trials).

| Noise ($\varepsilon$) | Protocol | FCR ↓ | TCR |
|---|---|---|---|
| 0.0 | ACP | **0.000** | 1.000 |
| | MCP | 0.000 | 1.000 |
| 0.2 | ACP | **0.000** | 0.991 |
| | MCP | 0.191 | 1.000 |
| 0.4 | ACP | **0.000** | 0.991 |
| | MCP | 0.445 | 1.000 |
| 0.6 | ACP | **0.000** | 1.000 |
| | MCP | 0.600 | 1.000 |
| 0.8 | ACP | **0.000** | 1.000 |
| | MCP | 0.836 | 1.000 |
| 1.0[†] | ACP | **0.000** | 1.000 |
| | MCP | 1.000 | 1.000 |

[†]At $\varepsilon = 1.0$, MCP commits 100% false commitment; ACP commits 100% true positives after in-session recovery. The FCR column captures the distinction TCR conceals.

Table 10: Effective token cost per correct commit (noise sweep data): avg_tokens / P(correct). ACP's effective cost is flat because it always commits correctly. MCP's effective cost grows with noise; at $\varepsilon = 1.0$ it diverges to $\infty$.

| $\varepsilon$ | ACP eff. | MCP eff. | Raw gap | Eff. gap |
|---|---|---|---|---|
| 0.0 | 6,783 | 500 | 13.6× | 13.6× |
| 0.2 | 7,598 | 619 | 15.0× | 12.3× |
| 0.4 | 7,848 | 900 | 15.6× | 8.7× |
| 0.6 | 8,220 | 1,243 | 16.5× | 6.6× |
| 0.8 | 8,231 | 3,072 | 16.4× | 2.7× |
| 1.0 | 8,518 | $\infty$ | — | $\infty$ |

tween application-level and protocol-level state becomes most consequential. InterCode-Spider tasks are inherently multi-turn: the agent issues a SQL query, receives execution feedback, and refines its query—a pipeline of reasoning steps where each depends on the previous step's output. SWE-bench instances require multi-file edits where a patch to one file may break imports in another, creating implicit multi-step dependencies. TravelPlanner requires sequential decisions (flight → hotel → activities) where an invalid early choice invalidates all downstream plans.

### A.8 Additional Figures

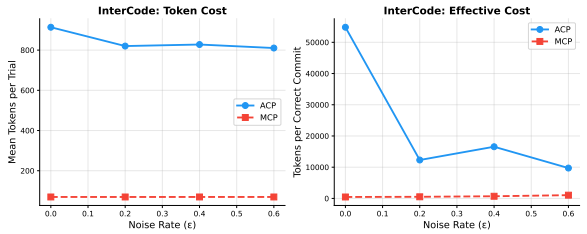


Figure 12: Token cost analysis on InterCode-Spider: ACP's per-session overhead vs. correctness gains. The effective cost per correct commit converges as noise increases, because MCP's wasted tokens on false commitments grow with $\varepsilon$.

Table 11: InterCode-Spider SQL chain yield (960 trials). ACP maintains higher yield through oracle-verified commits at each step. MCP yield degrades with depth, tracking the theoretical $(1-\varepsilon)^k$ bound.

| $k$ | $\varepsilon$ | ACP yield | MCP yield | Advantage |
|---|---|---|---|---|
| 2 | 0.4 | 0.600 | 0.300 | 2.0× |
| 2 | 0.6 | 0.700 | 0.233 | 3.0× |
| 3 | 0.4 | 0.367 | 0.167 | 2.2× |
| 3 | 0.6 | 0.533 | 0.100 | 5.3× |

## B Oracle Framing

**Execution oracles as verification functions, not answer keys.** A common objection to oracle-based verification is that it presupposes knowledge of the correct answer. This conflates two fundamentally different computational tasks: *generating* a correct solution and *checking* whether a candidate solution is correct. The execution oracles used in ACP are verification functions that test properties of the output, not lookup tables of gold answers. For code generation (APPS), the oracle compiles the agent's code in a sandboxed namespace and runs it against unit tests—it checks *behavior*, not textual equivalence with a reference implementation. For text-to-SQL (InterCode-Spider), the oracle executes the agent's query against a live database and compares the result set to pre-computed gold results via intersection-over-union; the oracle never reveals the gold SQL query itself, and any semantically equivalent query that produces the correct result set passes. For constrained planning (TravelPlanner), the oracle evaluates hard constraint satisfaction (budget, time windows, transportation availability) without reference to any particular plan. This asymmetry between generation difficulty and verification difficulty is well-established in complexity theory (the P vs. NP intuition) and is precisely what makes oracle-based verification practical: writing a correct SQL query is hard; executing a candidate query and checking its output is mechanical. Crucially, ACP's error feedback is designed to avoid lookahead bias—when verification fails, the `ErrorObject` describes

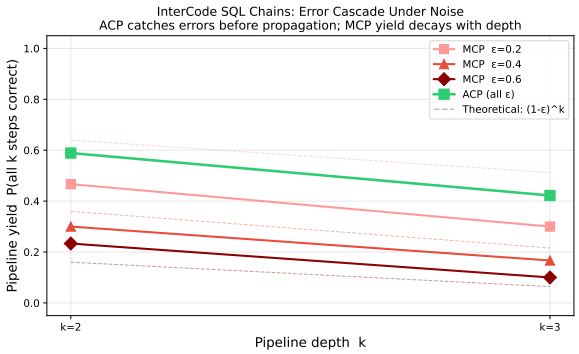


Figure 13: InterCode-Spider SQL chain yield vs. pipeline depth (960 trials). ACP yield degrades gracefully with depth; MCP yield tracks $(1-\varepsilon)^k$. At $k=3$, $\varepsilon = 0.6$: ACP yield = 0.533 vs. MCP yield = 0.100 (5.3× advantage).

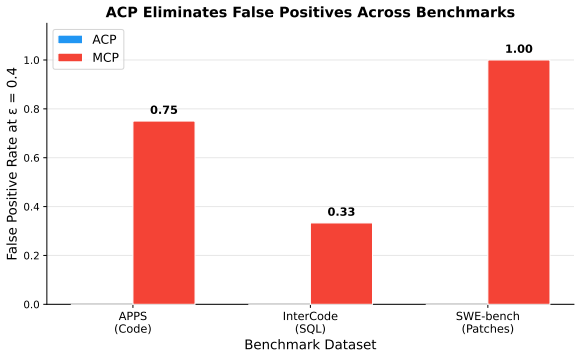


Figure 14: False commitment rates across three benchmarks. ACP achieves FCR = 0.000 across all datasets and noise levels, while MCP FCR scales with noise level $\varepsilon$.

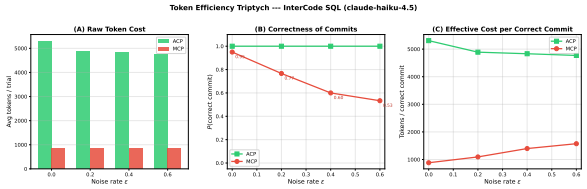


Figure 15: Token efficiency on InterCode-Spider with Claude Haiku 4.5: (A) raw token cost per trial, (B) probability of correct commit, (C) effective cost per correct commit. ACP's effective cost per correct commit is 4,800–5,300 tokens; MCP's rises from 880 to 1,570 as noise increases, but MCP's "cheap" cost buys increasingly wrong outputs.

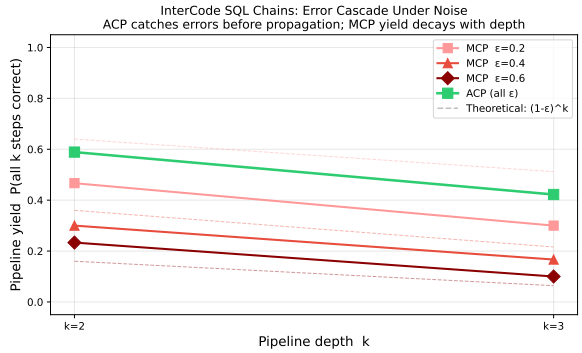


Figure 16: InterCode SQL chain yield vs. pipeline depth (480 clean trials, k=2,3). ACP yield exceeds MCP at all noise levels; the gap widens with both depth and noise. At k=3, $\varepsilon$=0.6, ACP achieves 5.3× the yield of MCP. Dashed lines show theoretical MCP bound $(1-\varepsilon)^k$.

*what* went wrong (e.g., "query returned 3 columns but expected 2") and provides directional correction hints, but never reveals the expected answer.

**The oracle spectrum and its limits.** Not all tasks admit deterministic verification. We distinguish three classes of execution oracle, ordered by decreasing reliability: (1) *Deterministic oracles* (test execution, SQL result-set comparison, constraint checkers), where correctness is mechanically decidable—ACP achieves FPR $= 0.000$ across all noise levels with these oracles; (2) *Constraint oracles* (schema validation, type checking, format rules), which verify necessary but not sufficient conditions for correctness—our ablation shows that schema-only validation (ACP − oracle) allows FPR $= 0.615$ at $\varepsilon = 0.2$, since semantically corrupted outputs that preserve valid structure pass unchallenged; (3) *LLM-judge oracles* (model-based semantic evaluation), where the verifier inherits the same error modes as the generator. ACP's advantage is strongest with deterministic oracles and degrades gracefully as oracle quality decreases. This is consistent with the formalizability boundary result: when tasks lack deterministic acceptance criteria, ACP's guarantees inherit the verifier's own error rate. The protocol's contribution is providing the *infrastructure* to invoke any oracle within a stateful verification loop—the oracle's strength determines the ceiling, but the protocol ensures that ceiling is reached.

**Oracle provenance and the role of the task deployer.** In practice, the execution oracle is specified by the *task deployer*—the developer, platform operator, or requesting agent that defines the task contract—not by the executing agent whose output is being verified. This separation is analogous to standard software engineering practice: unit tests are written by developers, not by the code under test; acceptance criteria are defined by product owners, not by the system being evaluated. In ACP's architecture, the oracle is passed

as a `session_verifier` parameter when the session is instantiated, and it is invoked by the requesting agent's `verify_result()` method after the responding agent's output passes structural (Stage 1) validation. The executing agent never sees the oracle function and cannot circumvent it. In a production deployment, the oracle could be specified as part of the task contract during the negotiation phase—the requesting agent defines what "correct" means, and the responding agent commits to satisfying those criteria. This is precisely the contract-based model that ACP formalizes: the contract's `success_criteria` and `output_schema` are grounded in the task specification (not generated by the LLM), and the oracle operationalizes those criteria into a mechanically executable check.

**The oracle is necessary but not sufficient: lessons from ablation.** The ablation study (Table 3) reveals a nuanced decomposition. Removing the execution oracle entirely (ACP − oracle) raises FPR from $0.000$ to $0.615$ at $\varepsilon = 0.2$—schema validation alone cannot detect semantic corruption, confirming that the oracle is the critical component for reliability. However, the oracle alone does not explain ACP's full advantage. MCP augmented with the same oracle and retry budget (MCP + oracle + retry) matches ACP's FPR of $0.000$, demonstrating that any protocol with access to a deterministic oracle can prevent false commitments. The difference surfaces in recovery effectiveness: at $\varepsilon = 0.2$, full ACP achieves TCR $= 0.855$, while ACP without typed errors (ACP − typed errors) drops to TCR $= 0.655$—a 20-point gap attributable to structured error signaling. The typed `ErrorObject`, which carries the error code, offending field, expected value, received value, and a correction hint, enables the responding agent to self-correct within the same session context rather than retrying from a blank slate. ACP's contribution is therefore not "having an oracle" but providing the *session-level infrastructure*—persistent context, structured error feedback, and bounded retry with full execution history—that maximizes the oracle's utility. A bare oracle can reject bad output; ACP turns that rejection into directed self-correction.

# C Illustrative Architecture: ACP Contract-Centric Session Lifecycle

The following describes one possible architectural realization of the design principles in §3.1. This section is not load-bearing for the central claim; it illustrates that the three principles are technically realizable in a coherent design.

## C.1 Central Contract Store (CCS)

To operationalize session-level reliability, we introduce a *Contract-Centric Session Lifecycle* anchored by a centralized protocol component, the **Central Contract Store (CCS)**. The CCS is a logical repository that persists shared contract state, negotiation history, and session metadata across agent interactions. It maintains: (i) the current contract specification, (ii) its revision history, (iii) agreement and validation states, and (iv) cached contextual artifacts generated during execution.

## C.2 Contract Initialization and Negotiation

Each session begins with the requesting agent transmitting an initial `CONTRACT` payload alongside the task specification. The responding agent acknowledges through a two-phase interpretation:

1. **Capability Verification (Boolean):** A signal indicating whether the agent can satisfy the contract as stated.

2. **Semantic Interpretation (Text):** A structured description of the agent's understanding of the task, constraints, and limitations.

If the interpretation deviates from the intended contract semantics, the contract enters a renegotiation phase.

## C.3 Contract Agreement, Execution, and Bounded Retries

Once mutual agreement is reached, the finalized contract is committed to the CCS and becomes immutable for the remainder of the session unless explicitly revised. After execution, the responding agent's result is verified against the contract. Verification failure triggers a *contract update* written as a new CCS revision. A strict maximum revision limit prevents unbounded execution. During execution, the responding agent may emit a `HOLD` state to preserve session liveness without triggering retries or timeouts.

## C.4 Session Termination Semantics

A session concludes in one of two terminal states:

(i) **COMMITTED**, upon verified contract satisfaction, or

(ii) **ABORTED**, upon exceeding retry limits or explicit termination by the requesting agent.

In both cases, the CCS retains the full contract and execution history for downstream auditing and accountability.

## D Detailed ACP Session Trace (Contract Negotiation)

The following trace illustrates an ACP session handling both semantic ambiguity during negotiation and a field-injection error during execution.

**Session Trace: Contract Negotiation and Recovery**

**PHASE 1: NEGOTIATION**
`[04:02:36]` **CCS:** Status `NEGOTIATING`. Contract Rev 1 issued.
`[04:02:38]` **Recipient:** Proposed Plan: *"Create JSON representing product order."*
`[04:02:40]` **Caller:** `REJECT`. Reason: Plan too vague; missing implementation details.
`[04:02:41]` **CCS: Contract Updated → Rev 2**. (Requires code snippet + validation logic).
`[04:02:45]` **Recipient:** Revised plan with implementation details.
`[04:02:48]` **Caller:** `APPROVE`. Plan aligns with Rev 2 requirements.
`[04:02:48]` **CCS:** Status `EXECUTING`.

**PHASE 2: EXECUTION & FAULT RECOVERY**
`[04:02:50]` **Recipient:** Attempt 1 Output: `{"product_id": "abc-123"}`
*(Note: System injected* `missing_required_field` *error)*
`[04:02:52]` **Caller:** `UPDATE`. Missing mandatory fields `quantity` and `price_usd`.
`[04:02:53]` **CCS: Contract Updated → Rev 3**.
`[04:02:58]` **Recipient:** Attempt 2 Output: [Full schema coverage].
`[04:03:02]` **Caller:** `COMMIT`. Output satisfies schema and Rev 3 post-conditions.
`[04:03:02]` **CCS:** Status `COMMITTED`.

---

**Data Insight:** The session survived a semantic rejection (Phase 1) and a syntactic injection error (Phase 2), converging via two protocol-managed loops.

## E Detailed ACP Session Trace (Tool Recovery)

The following trace shows ACP's persistence when faced with hallucinated tool names and environmental noise.

**Session Trace: Tool Name Recovery**

**PHASE 1: NEGOTIATION**
`[04:15:10]` **CCS:** Status `NEGOTIATING`. Contract Rev 1: Fetch user profile and format as JSON.
`[04:15:15]` **Recipient:** AGREED. Goal is to retrieve and structure user data.
`[04:15:18]` **Caller:** `APPROVE`.
`[04:15:18]` **CCS:** Status `EXECUTING`.

**PHASE 2: EXECUTION & TOOL-CALL RECOVERY**
`[04:15:22]` **Recipient:** Attempt 1 Output: `get_usr_data(id="123")`
*(Note: Tool* `get_usr_data` *does not exist in available toolset)*
`[04:15:25]` **Caller:** `UPDATE`. Invalid tool call. Available tools: `fetch_user_profile`.
`[04:15:26]` **CCS: Contract Updated → Rev 2**.
`[04:15:32]` **Recipient:** Attempt 2 Output: `fetch_user_profile(user_id="123")`
*(Note: Environmental noise corrupts output syntax)*
`[04:15:35]` **Caller:** `UPDATE`. Syntax error in response.
`[04:15:36]` **CCS: Contract Updated → Rev 3**.
`[04:15:45]` **Recipient:** Attempt 3 Output: `{"user_id": "123", "status": "active"}`
`[04:15:50]` **Caller:** `COMMIT`. Valid tool output and schema adherence.
`[04:15:50]` **CCS:** Status `COMMITTED`.

---

**Session Metrics:** Total Steps: 11 | Update Overhead: 721 tokens | Result: SUCCESS

## F Scalability and Latent Communication

**Scalability considerations.** Stateful protocols introduce infrastructure complexity relative to stateless REST architectures (Fielding et al., 2017; Cohen et al., 2020). Session affinity complicates horizontal scaling and load balancing. However, the relevant comparison is not between stateful and stateless architectures in general, but between the cost of maintaining session state and the cost of propagating false commitments through multi-step pipelines. The pipeline depth results (Table 4) show that MCP's error compounds exponentially with depth: at $k = 4$ steps and $\varepsilon = 0.6$, MCP yield drops to 0.100 while ACP maintains 1.000. The infrastructure cost of session state is a fixed investment; the cost of undetected errors grows with pipeline depth and system scale. For non-idempotent, multi-step workflows with irreversible side effects—the regime where agentic systems differ most sharply from traditional microservices—this tradeoff favors statefulness.

**Latent communication.** Recent work on latent space communication (Zou et al., 2025; Zheng et al., 2025) raises the question of whether explicit

symbolic protocols are necessary at all. Latent communication can improve bandwidth and convergence speed in homogeneous agent settings, but it addresses a different problem: intent transfer fidelity, not output verification. Agentic ecosystems are heterogeneous (embeddings from one model are not interpretable by another), and atomic task completion requires contractual verification that symbolic protocols provide. The two approaches are complementary: latent channels for efficient communication within trusted boundaries, symbolic session protocols for verified commitment across organizational boundaries.