# TRAJECTORY-LEVEL SPECULATION WITH SEMANTIC VERIFICATION FOR TOOL-USING AGENTS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Large language model agents solve tasks through long sequences of tool calls, and the prevailing design has a single strong model generate every step, although many steps are routine enough for a much smaller model. Existing ways of sharing this work keep the strong model as the generator: routers decide which model produces each step, and speculative methods check a small model's drafts one step at a time, usually against the strong model's own action. We introduce SPECAGENT, a collaboration protocol that turns the strong model from the author of each step into a reviewer of steps drafted by a weak model. The weak model executes read-only tool calls as it drafts, so that every step is grounded in real observations, and holds the first action that would modify the environment. The strong model then verifies the drafted segment in a single forward pass, commits the longest prefix it judges semantically valid, and repairs the first rejected step. Because verification replaces autoregressive decoding with parallel prefill, the strong model decodes only short verdicts and occasional repairs rather than every step. On $\tau^2$-Bench, SPECAGENT comes within one point of the strong model's success rate while roughly halving memory traffic and wall time, and it lowers both on AutomationBench. In an edge–cloud deployment on MCP-Atlas with three cloud models, it closes 53–75% of the weak–strong quality gap while the strong model generates at most half as many tokens. These results suggest that a strong model need not author every step of an agent: reviewing grounded drafts retains most of its quality while cutting both latency and memory traffic.

## 1 INTRODUCTION

Language-model agents now solve tasks through long sequences of tool calls and user turns, from tens of turns in customer service to hundreds in everyday-life assistance (Table 1). The prevailing design has one strong model generate every step, so serving cost grows with the length of the trajectory and is dominated by the strong model's token-by-token decoding. Yet many steps are routine, such as looking up an order, reading a record, or searching again, and a much smaller model could often produce them.

Table 1: Representative trajectory length per task for frontier agents, counted as messages from the agent, the user, and tools.

| **Real-World MCP Services** (MCP-Atlas) | **Transactional Interaction** ($\tau^2$-Bench) | **Business Workflows** (AutomationBench) | **Life Assistance** (VibeLifeBench) |
|---|---|---|---|
| 20–30 turns | 30–80 turns | 40–60 turns | ∼200 turns |

Existing ways of sharing this work keep the strong model as the generator. Routers and collaborative agents decide, step by step, which model should act (Chen et al., 2023b; Aggarwal et al., 2024; Ding et al., 2024; Ong et al., 2025; Gao et al., 2026), so every step is still written by one model from scratch. Speculative agents carry the draft-and-verify idea of speculative decoding (Leviathan et al., 2023) over to actions (Hua et al., 2024; Ye et al., 2026), but usually keep a draft only when it matches the strong model's own action, so the strong model must still generate each reference step and its work is overlapped rather than removed. Recent variants replace matching with a step-level critic



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

(Huang et al., 2025; Zhong et al., 2026), but they check one predicted step at a time in search-style tasks whose actions do not change the environment. Exact matching is also too strict for agents: different queries, lookup orders, or argument choices can all lead to a valid solution, and progress that differs from the strong model's choice is discarded. We therefore ask: *can a strong model review several steps drafted by a weak model, and keep what is valid, instead of generating every step itself?*

We answer it with SPECAGENT, in which a small *Weak* model drafts the trajectory and a large *Strong* model reviews it, separating acting from committing (Figure 1). Three properties of agents shape its design. First, a draft is meaningful only if it is grounded, because later steps depend on what earlier tool calls return; the Weak model therefore executes read-only calls as it drafts. Second, some actions cannot be undone, so the Weak model holds the first state-changing action, such as changing an order, and the Strong model reviews the segment before it runs, commits the longest prefix it judges semantically valid, and repairs the first rejected step. Third, review must cost less than generation even though the reviewer reads the full context: the Strong model reads a segment in one parallel prefill pass, whereas generating it would require decoding every token.

We compare against *Strong-only* and *Weak-only*, the single-model agents in which one model generates every step, and evaluate complete quality–resource operating points, counting both models and keeping compute, memory traffic, and wall time as separate axes. On $\tau^2$-Bench, SPECAGENT comes within one point of Strong-only success while reducing estimated memory traffic to 43–57% and wall time to 41–54% of Strong-only; on AutomationBench it recovers 44% of the Weak–Strong quality gap, again with lower memory traffic and wall time. In an edge–cloud deployment on MCP-Atlas with three cloud Strong models, it closes 53–75% of the gap while the Strong model generates at most half as many tokens. The saving comes from decoding rather than arithmetic: estimated FLOPs rise by 30–38%, so SPECAGENT suits memory-bandwidth-bound serving.

Our contributions are as follows.

- **Framework.** SPECAGENT, a weak–strong protocol in which the Weak model drafts with real read-only tool calls and holds the first state-changing action, and the Strong model commits the longest prefix it judges semantically valid and repairs the first rejected step, together with a cost model that predicts when replacing Strong decoding by prefill saves memory traffic and time.
- **Evaluation.** Experiments on $\tau^2$-Bench, AutomationBench, and an edge–cloud deployment on MCP-Atlas with three cloud Strong models, comparing complete quality–resource operating points against routing and speculative baselines.
- **Analysis.** An accounting of where the savings arise: measured Strong traffic and time follow the cost model, the saving comes from decoding, not computation, and it shrinks as acceptance falls.

## 2 RELATED WORK

**Speculative execution.** Speculative decoding lets a small model draft tokens that a large model verifies in parallel while preserving its output distribution (Leviathan et al., 2023; Chen et al., 2023a); it is complementary to SPECAGENT, accelerating each model's own decoding. The draft-then-verify pattern has since been applied to reasoning and search (Wang et al., 2025a;b; Fu et al., 2025) and to agents, where a fast model predicts plans, actions, or tool calls (Hua et al., 2024; Guan et al., 2025; Ye et al., 2026; Nichols et al., 2025; Hooper et al., 2026; Ji et al., 2026), sometimes with relaxed or semantic matching (Huang et al., 2025; Zhong et al., 2026). These methods target latency and keep a draft when it agrees with the large model's own step. SPECAGENT instead verifies multi-step, grounded segments in one pass and checks only before actions that change the environment.

**Routing and weak–strong collaboration.** Cascades and routers pick one model per query, including edge–cloud deployments (Chen et al., 2023b; Aggarwal et al., 2024; Ding et al., 2024; Ong et al., 2025); agent-level methods hand control between models during a trajectory (Gao et al., 2026; Hemadri et al., 2026; Wei et al., 2026). In both, the model in control generates each step; in SPECAGENT, the Strong model decides how much drafted progress to keep and generates only repairs.

**Verification versus generation.** Checking a candidate is often easier than producing one, as verifiers and process reward models show for reasoning (Cobbe et al., 2021; Lightman et al., 2024). SPECAGENT applies this asymmetry to agent trajectories, with the Strong model as the verifier,

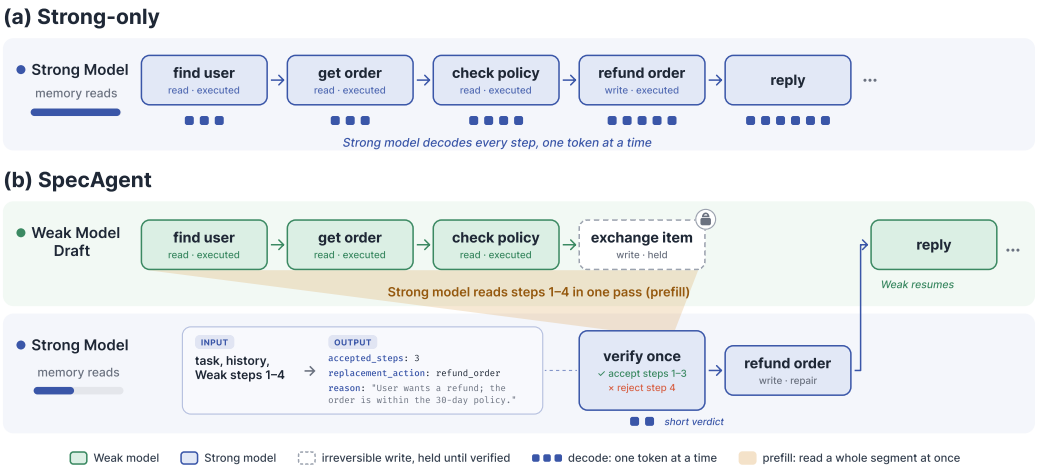


Figure 1: (a) A Strong-only agent generates every step by decoding. (b) SPECAGENT lets the Weak model draft ahead, running reads at once and holding the first write. The Strong model reviews the draft in a single prefill pass, keeps the valid steps, and replaces the rejected write; the Weak model then resumes. Bars indicate relative Strong-model memory reads.

and exploits its cost counterpart: prefill processes many tokens per weight read, whereas decoding rereads them for every token (Pope et al., 2023; Kwon et al., 2023; Zheng et al., 2024).

## 3 SPECAGENT

SPECAGENT distinguishes steps by whether their effects can be undone. A read leaves the state untouched, so the Weak model can execute it immediately, and a wrong read costs no more than being discarded; a state-changing action is where an error becomes permanent, and it is the only point at which SPECAGENT stops for the Strong model. This lets the Weak model speculate over whole stretches of the trajectory while keeping execution safe. When the Strong model steps in, it judges whether the drafted progress is valid rather than whether it matches what it would have done, so it reviews the trajectory instead of writing it (Figure 1).

### 3.1 PROBLEM FORMULATION

We model a tool-using task as a partially observable Markov decision process. The environment has a hidden state $s$, such as a database or a workspace, that the agent cannot see directly. At each step the agent issues an action $a_t$, typically a tool call, and receives an observation $o_t$. A policy therefore conditions on the task $u$ and the history

$$H_t = \big(u, (a_1, o_1), \ldots, (a_t, o_t)\big). \tag{1}$$

Two policies share this interface: a Weak model $\pi_W$ that is cheap to run and a Strong model $\pi_S$ whose quality we want to preserve.

Actions fall into two classes. A *read*, such as a search or a lookup, returns an observation and leaves the state unchanged. A *state-changing* action, such as modifying a record or submitting a final result, alters the state. The distinction determines what can be undone, and it is the principle behind SPECAGENT. An error that stays in the history is recoverable: a wrong read is removed by deleting it from the history, and because reads leave the task state unchanged, the environment is as it was (Appendix D). An error that reaches the environment is not, since no edit to the history withdraws a changed record or a submitted result. Speculation can therefore proceed freely while execution remains reversible, and verification is needed exactly at the boundary where an error would become irreversible: earlier checks spend the Strong model on steps that could still be undone, and later ones come too late.

### 3.2 Speculative Execution Protocol

**Draft.** From $H_t$, the Weak model drafts a chunk of at most $K$ steps,

$$\hat{\tau} = \big((\hat{a}_1, \hat{o}_1), \ldots, (\hat{a}_{n-1}, \hat{o}_{n-1}), (\hat{a}_n, \bot)\big). \tag{2}$$

Reads execute immediately, so each step is chosen after seeing real observations rather than imagined ones. The chunk ends at the first state-changing action $\hat{a}_n$, which is held rather than executed, since every later step would depend on its outcome. A chunk that reaches $K$ steps with only reads is committed without review, as the reversibility principle allows.

**Algorithm 1** SpecAgent: draft, verify, repair

**Require:** Task $u$, Weak policy $\pi_W$, Strong verifier $V_S$, Strong policy $\pi_S$, chunk size $K$, adapter $E$

$H \leftarrow (u)$
**while** the task is not complete **do**
  $\hat{\tau} \leftarrow [\,]$
  **loop**
    $\hat{a} \leftarrow \pi_W(H \oplus \hat{\tau})$
    **if** $E$.StateChanging($\hat{a}$) **then**
      Append $(\hat{a}, \bot)$ to $\hat{\tau}$; **break**
    **end if**
    $\hat{o} \leftarrow E$.Execute($\hat{a}$); append $(\hat{a}, \hat{o})$ to $\hat{\tau}$
    **if** $|\hat{\tau}| = K$ **then**
      $H \leftarrow H \oplus \hat{\tau}$; $\hat{\tau} \leftarrow [\,]$
    **end if**
  **end loop**
  $(j, r, \tilde{a}) \leftarrow V_S(H, \hat{\tau})$
  **if** $j = |\hat{\tau}|$ **then**
    $H \leftarrow H \oplus \mathrm{Prefix}_{j-1}(\hat{\tau})$
    Execute the held action; append its result to $H$
  **else**
    $H \leftarrow H \oplus \mathrm{Prefix}_j(\hat{\tau})$
    **if** $E$.Valid($\tilde{a}$) **then**
      $c \leftarrow \tilde{a}$
    **else**
      $c \leftarrow \pi_S(H, r)$
    **end if**
    Execute $c$; append $(c, o_c)$ to $H$
  **end if**
**end while**

**Verify.** The Strong model reads the history and the chunk in a single pass and returns a verdict

$$(j, r, \tilde{a}) = V_S(H_t, \hat{\tau}), \qquad j \in \{0, \ldots, n\}, \tag{3}$$

where $j$ is the length of the longest acceptable prefix, $r$ explains the first rejected step, and $\tilde{a}$ optionally replaces it. The verdict is a prefix because each drafted step conditions on the observations of the steps before it: once a step is wrong, later steps rest on a mistaken premise and cannot be trusted even if they look valid. Acceptance is *semantic*: the verifier asks whether each step is a valid move toward the task, not whether it is the step the Strong model would have written. Tool-using tasks admit many valid trajectories, so semantic review retains progress that exact matching would discard. If $j = n$, the whole chunk is committed and the held action is executed.

**Repair.** A rejection should cost the Strong model as little generation as possible. Many rejected steps are a single tool call with a wrong argument or target, which the verifier can state while explaining the rejection; the verdict then carries a replacement $\tilde{a}$, and the repair adds no Strong call. Some repairs cannot be written that way, for instance when the fix spans several calls or the proposed replacement fails the adapter's validity checks; only then does the Strong model take one corrective turn from the history and the reason $r$. The fused path keeps the common case cheap, and the fallback keeps every rejection repairable. A repair is the Strong model's own step, so it is committed without a second review. The rejected suffix, including the held action, is discarded, so the next draft starts from a history of committed steps only. Algorithm 1 gives the full loop.

### 3.3 Efficiency Analysis

Autoregressive decoding generates one token per forward pass, and each pass streams the model's active weights from GPU memory, so decoding is bound by memory bandwidth rather than arithmetic (Pope et al., 2023). A prefill pass processes many tokens against a single weight read. We make this the basis of a simple cost model.

**Assumption 1** (Decode-bound serving)**.** *While a Strong replica decodes with a batch small enough to remain memory-bound, the time and memory traffic of a Strong call are approximately proportional to its number of weight passes: one per decoded token and one for the prefill.*

**Assumption 2** (Per-task acceptance)**.** *As a first-order model, each task $u$ has an average per-step acceptance rate $p_u$, related to the task's difficulty, the Weak model's competence, and the Strong verifier's preferences; the steps of a reviewed chunk of length $n$ are accepted independently.*

Let $d$ be the weight passes of a Strong-only step, $v$ those of a verdict, and $\rho$ the probability that a rejection needs a separate repair turn because no valid replacement was fused into the verdict. The comparison is intuitive. In Strong-only, every committed step costs a full decoded step. In SPECAGENT, the Strong model instead decodes one short verdict per reviewed chunk, shared by all the steps the chunk commits, and a repair only when a step is rejected. Its cost per committed step should therefore shrink as verdicts get shorter, as more of each chunk is accepted, and as fewer rejections need a separate repair turn. Proposition 1 makes this precise.

**Proposition 1** (Strong decoding)**.** *Under Assumptions 1 and 2, on task $u$ the Strong model's time and memory traffic per committed step, relative to Strong-only, are*

$$R_u = \frac{1 - p_u}{1 - p_u^n} \cdot \frac{v}{d} + \rho\,(1 - p_u). \tag{4}$$

*If task $u$ commits $T_u$ steps in both systems, the ratio over a task distribution is the step-weighted mean $R = \mathrm{E}_u[T_u R_u]/\mathrm{E}_u[T_u]$.*

$R_u$ falls as $p_u$ rises, from $v/d+\rho$ at $p_u = 0$ to $v/(nd)$ as $p_u \to 1$, so the saving concentrates on tasks where the Weak model's drafts suit the verifier; read-only chunks committed without review only lower $R$ further. Saving Strong time does not by itself shorten the task: every committed step must first be drafted, so the Weak model's decoding replaces part of the Strong model's, and SPECAGENT is faster only if a Weak step is cheap relative to a Strong one.

**Proposition 2** (Latency)**.** *Under the same assumptions, let $\kappa$ be the time of a Weak drafting step relative to a Strong-only step, and ignore tool latency. On task $u$ the end-to-end time per committed step, relative to Strong-only, is*

$$L_u = R_u + \kappa\,\frac{n(1 - p_u)}{1 - p_u^n}, \tag{5}$$

*and the ratio over tasks is again the step-weighted mean of $L_u$.*

The second term counts drafted steps per committed step and approaches one as $p_u \to 1$. Proofs are in Appendix C, and Section 5 examines where the savings arise.

## 4 EXPERIMENTS

We evaluate SPECAGENT on agent tasks that mix reads, state-changing actions, and user-visible replies. The experiments ask three questions. **Q1:** how much of the Weak–Strong quality gap does semantic prefix verification recover? **Q2:** which resources does it save, and which does it not? **Q3:** how does it compare with existing Weak–Strong collaboration methods at their own operating points?

### 4.1 SETUP

**Benchmarks.** $\tau^2$-Bench Airline (50 tasks) and Retail (114 tasks) test multi-turn customer service against a simulated user, mixing database reads, mutations, and replies (Barres et al., 2025). We run three independent trajectories per task and report the two domains separately because their action mix and difficulty differ. AutomationBench tests long business workflows whose success is checked on the final environment state (Shepard & Salimans, 2026); we aggregate its 600 public-domain tasks into one block, with its development domain excluded. MCP-Atlas, run against real tool servers in an edge–cloud deployment, is reported separately (Section 4.3).

**Models and methods.** On $\tau^2$ and AutomationBench, the Weak model is Qwen3.5-4B and the Strong model is Qwen3.5-397B-A17B (Qwen Team, 2026). Weak-only and Strong-only run the same agent loop with a single model. AgentCollab (Gao et al., 2026), the Weak–Strong collaboration baseline, escalates to Strong using its published self-evaluation signal. Speculative Actions (Ye et al., 2026) keeps Strong as the actor and speculates on upcoming steps to hide latency; we port the authors' released agent to $\tau^2$ (Appendix E). It is not run on AutomationBench, which has no official implementation and whose tool results are rarely predictable. All methods run on the same Weak and Strong deployments. SPECAGENT reviews at state-changing actions with chunk size $K = 6$ on $\tau^2$ and $K = 8$ on AutomationBench; other adapter settings and all prompts are listed in Appendix D.

**Serving.** On $\tau^2$ and AutomationBench, all models run on NVIDIA H20 GPUs: the Weak model on a single GPU with SGLang (Zheng et al., 2024), and the Strong model with 16-way tensor parallelism. Every method runs eight tasks concurrently against the same deployments, so wall time is compared under a common load. Decoding settings and the $\tau^2$ user simulator are given in Appendix A.

**Metrics.** Pass@1 is the benchmark-native success rate per trajectory, averaged over all trials; quality differences use a paired bootstrap over tasks (Appendix B.1). Resource columns are per-trajectory means over every agent-model request, including Weak drafts, rejected chunks, verification, repair, and retries. We estimate FLOPs and HBM-read from each request's token counts and the model architecture (Appendix B.3, B.4), charging a prompt prefix once per session under a radix prefix-cache model (Zheng et al., 2024) applied identically to all methods (Appendix B.2); the HBM interval of the MoE Strong model reflects its unknown expert routing during prefill. GPU·s sums request duration times serving GPUs, wall time is mean elapsed time per trajectory, and calls count Weak/Strong generation requests (Appendix B.5).

Table 2: Quality and mean per-trajectory resources of the agent models on $\tau^2$-Bench and AutomationBench. Compute and HBM-read are estimated from logged tokens and model architecture, assuming prefix reuse within a task. HBM-read of a Strong MoE model is a range, because the number of experts a prefill pass reads is not logged; the dense Weak model has a point estimate. Calls count Weak and Strong model calls per trajectory.

| Method | Pass@1 (%) ↑ | Est. compute (PFLOP) ↓ | HBM-read (TB) ↓ | Wall-time (s) ↓ | GPU·s ↓ | Calls (W/S) ↓ |
|---|---|---|---|---|---|---|
| *$\tau^2$-Bench (Airline)* | | | | | | |
| Weak-only (4B) | 80.0 | 0.18 | 83.8 | 69.7 | 66.0 | 14.1/0.0 |
| Strong-only (397B) | 86.0 | 0.78 | [333.2, 341.6] | 305.2 | 4828.4 | 0.0/13.8 |
| AgentCollab | 85.3 | 1.12 | [276.6, 289.4] | 385.2 | 5629.3 | 9.0/18.5 |
| Speculative Actions | 72.0 | 0.89 | [429.5, 438.2] | 665.6 | 9990.6 | 4.5/17.4 |
| SPECAGENT | 86.0 | 1.01 | [147.6, 150.1] | 126.2 | 860.4 | 15.5/3.1 |
| *$\tau^2$-Bench (Retail)* | | | | | | |
| Weak-only (4B) | 80.4 | 0.14 | 48.4 | 45.7 | 41.2 | 13.3/0.0 |
| Strong-only (397B) | 88.6 | 0.58 | [192.8, 201.8] | 162.4 | 2535.6 | 0.0/13.1 |
| AgentCollab | 85.7 | 0.98 | [171.9, 184.2] | 223.2 | 3147.8 | 10.3/16.6 |
| Speculative Actions | 48.5 | 0.68 | [270.4, 280.4] | 263.5 | 3944.7 | 5.0/17.1 |
| SPECAGENT | 87.7 | 0.81 | [107.3, 109.7] | 87.5 | 789.7 | 13.8/3.1 |
| *AutomationBench* | | | | | | |
| Weak-only (4B) | 6.8 | 0.29 | 94.5 | 60.1 | 60.0 | 20.4/0.0 |
| Strong-only (397B) | 15.2 | 1.10 | [324.0, 337.3] | 307.4 | 4483.2 | 0.0/17.4 |
| AgentCollab | 7.3 | 3.99 | [470.0, 510.6] | 932.0 | 8603.0 | 15.7/54.8 |
| SPECAGENT | 10.5 | 1.52 | [264.5, 273.6] | 174.9 | 1761.0 | 24.4/11.8 |

### 4.2 MAIN RESULTS

**Quality (Q1).** On $\tau^2$ Airline, SPECAGENT reaches the Strong-only Pass@1 (86.0 vs. 86.0), up from 80.0 for Weak-only. On Retail it comes within 0.9 points of Strong-only (87.7 vs. 88.6, from 80.4 for Weak-only), recovering 89% of the Weak–Strong gap, and on AutomationBench it recovers 44% (10.5 vs. 6.8 and 15.2). On $\tau^2$, paired bootstrap intervals that resample tasks with all their trials give an observed SPECAGENT−Strong Pass@1 difference of 0.0 points on Airline (95% CI $[-5.3, +5.3]$) and $-0.9$ points on Retail ($[-4.7, +2.9]$).

**Resources (Q2): memory traffic and time fall, compute does not.** Relative to Strong-only (Figure 2), SPECAGENT lowers estimated HBM-read to 43–45% on Airline, 53–57% on Retail, and 78–84% on AutomationBench. Wall time falls to 41%, 54%, and 57%. Estimated FLOPs, however, *rise* by 30%, 38%, and 38%. The two axes diverge as Section 3.3 anticipates: Strong calls drop from 13.8 to 3.1 per Airline trajectory and from 13.1 to 3.1 on Retail, so most Strong decoding is replaced by prefill, while Weak drafts and verification add compute (Section 5).

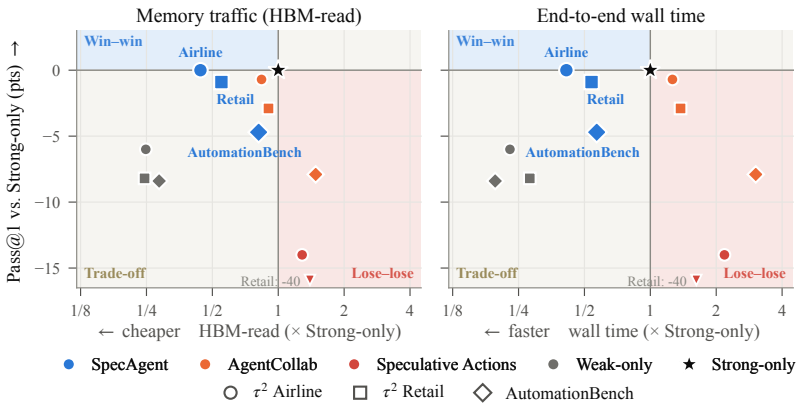


Figure 2: Quality against memory traffic and wall time, relative to each domain's Strong-only run (star). Quadrants, drawn from point estimates, mark win–win, lose–lose, and trade-off regions.

Table 3: Edge–cloud deployment on MCP-Atlas (101 tasks, real MCP servers, one trajectory per task). The Weak model, Qwen3.5-4B, runs on one local RTX 5090; each Strong model is served by its provider's cloud API. The Weak-only row is shared by all groups. Resources are per-task means and API cost is the total over all tasks at list price, excluding the local GPU.

| Method | Pass@1 (%) ↑ | Est. compute (PFLOP) ↓ | HBM-read (TB) ↓ | Strong output tokens ↓ | Wall-time (s) ↓ | API cost ($) ↓ |
|---|---|---|---|---|---|---|
| Weak-only (4B) | 44.6 | 0.67 | 39.1 | – | 109 | 0.00 |
| *Strong: Qwen3.5-397B-A17B* | | | | | | |
| Strong-only (397B) | 72.3 | 1.72 | [119.5, 128.6] | 3.6k | 105 | 1.39 |
| SPECAGENT | 59.4 | 2.64 | [105.7, 109.0] | 1.8k | 120 | 1.12 |
| *Strong: GLM-5* | | | | | | |
| Strong-only (GLM-5) | 76.2 | 10.07 | [377.6, 389.4] | 4.6k | 110 | 8.91 |
| SPECAGENT | 67.3 | 10.50 | [217.0, 223.1] | 2.1k | 108 | 5.54 |
| *Strong: DeepSeek-V4-Flash-0731* | | | | | | |
| Strong-only (DeepSeek) | 84.2 | 3.38 | [150.3, 151.7] | 10.6k | 151 | 1.37 |
| SPECAGENT | 74.3 | 2.63 | [102.7, 103.5] | 3.8k | 215 | 0.81 |

**Comparison with collaboration methods (Q3).** Both baselines spend more Strong work than Strong-only. Speculative Actions keeps the Strong model acting at every step and adds background Strong speculation, so it issues more Strong calls than Strong-only (17.4 vs. 13.8 on Airline) and more memory traffic and time; in the released implementation it also falls below Weak-only quality (72.0 on Airline, 48.5 on Retail; Appendix E). AgentCollab stays near Strong quality on $\tau^2$ (85.3 and 85.7) but issues more Strong calls than Strong-only (18.5 vs. 13.8 on Airline) and takes longer. On AutomationBench it reaches 7.3 with 1.5× the Strong-only HBM-read and 3.0× its wall time. SPECAGENT matches or exceeds AgentCollab's quality on $\tau^2$ (86.0 vs. 85.3 and 87.7 vs. 85.7) with 1.6–1.9× less HBM-read and 2.6–3.1× less wall time, and it is the only collaborative method whose HBM-read and wall time are both below Strong-only on every benchmark.

### 4.3 EDGE–CLOUD DEPLOYMENT: MCP-ATLAS

MCP-Atlas evaluates multi-step tool use against real Model Context Protocol servers for search, code execution, databases, and productivity tools (Anthropic, 2024; Bandi et al., 2026); an LLM judge scores each response by the share of reference claims it covers. We use the 101 public tasks whose servers run without private API keys (Appendix A) and deploy SPECAGENT across the edge and the cloud: Qwen3.5-4B runs locally on a single RTX 5090, and the Strong model is one of Qwen3.5-397B-A17B, GLM-5 (GLM-5 Team, 2026), and DeepSeek-V4-Flash-0731 (DeepSeek-AI, 2026), each behind its provider's API. In this setting the cloud model is billed per token, so we report its output tokens and API cost alongside quality, and compare against Weak-only and

Strong-only. Across the three Strong models, SPECAGENT recovers 53–75% of the Weak–Strong gap while the Strong model decodes only 36–50% of its Strong-only output tokens, and API cost falls to 59–81% (Table 3). For every Strong model the share of the gap recovered exceeds the share of Strong tokens spent, so each Strong token buys more quality inside SPECAGENT than on its own. Wall time does not fall here: the local Weak model takes about as long per task as the hosted Strong models (109 vs. 105–151 seconds), so the latency term of Proposition 2 offers no gain, and the benefit appears instead as lower remote usage and cost.

## 5 ANALYSIS AND DISCUSSION

We relate the measurements of Section 4 to the cost model of Section 3.3, then use it to delimit where SPECAGENT pays off.

**Strong decoding explains the saving.** Proposition 1 computes, under Assumption 2, a quantity that can also be measured directly. Let $c_v$ and $c_r$ be the verifications and separate repair turns per committed step, and $\bar{v}$, $\bar{d}_r$, $\bar{d}$ the mean output tokens of a verdict, a repair, and a Strong-only step. Under Assumption 1 the Strong model's traffic per committed step, relative to Strong-only, is

$$R = c_v \frac{\bar{v}}{\bar{d}} + c_r \frac{\bar{d}_r}{\bar{d}}, \tag{6}$$

of which Proposition 1 is the closed form. Multiplying by the ratio of committed steps per trajectory gives the predicted Strong share of a trajectory. Table 4 evaluates (6) from logged token counts. On $\tau^2$ the Strong model is consulted about once every seven committed steps, and its decoded tokens fall to 15–26% of Strong-only. The observed Strong shares of HBM-read and generation time follow the same ordering across domains; HBM-read runs slightly higher because decoded tokens omit the prefill and KV traffic of each call. On AutomationBench, the Weak model's drafts are rejected more often and each rejection costs a separate repair of about $1.5\bar{d}$ tokens, so the $c_r$ term dominates and the saving shrinks, as the $\rho(1 - p_u)$ term of Proposition 1 anticipates.

**Why time falls while FLOPs rise.** As Section 3.3 explains, decoding is bound by memory bandwidth while a verification prefill over a whole chunk is bound by compute, so moving Strong tokens from decoding to prefill raises FLOPs but sharply reduces time. Figure 3 splits the cost by phase. Verification carries most of SPECAGENT's FLOPs but little of its HBM-read; Weak drafting, itself memory-bound decoding, does the reverse. Wall time decomposes as Proposition 2 describes: Strong generation shrinks to 0.16–0.34 of Strong-only and Weak drafting adds about 0.22–0.24. On AutomationBench, separate repairs dominate the Strong model's share, the $\rho(1 - p_u)$ term at work.

**When verification recovers Strong quality.** Assumption 2 ties acceptance to how well the Weak model handles a task, so we split $\tau^2$ tasks by whether Weak-only solves them in all three trials. Where the Weak model is unreliable, SPECAGENT stays close to Strong-only (70.4 vs. 66.7 on Airline, 18 tasks; 74.2 vs. 75.8 on Retail, 44 tasks), and where it always succeeds both are near ceiling (94.8 vs. 96.9 and 96.2 vs. 96.7). These subsets are small, but they suggest that one review and repair per held action suffices when the Weak model is partly competent; when it is far weaker, as on AutomationBench, repairs grow and less of the gap is recovered.

**Design choices.** Figure 4 varies the three choices of Section 3.2 on Retail. *Where to review:* also holding replies, or reviewing every chunk including reads, more than triples the Strong calls and raises HBM-read by 63–79% and wall time 2.5–3.0×, past Strong-only, while lowering Pass@1.

Table 4: Strong decoding relative to Strong-only: predicted from (6) and observed.

| | Verify | | Repair | | Predicted | Observed | |
|---|---|---|---|---|---|---|---|
| | $c_v$ | $\bar{v}/\bar{d}$ | $c_r$ | $\bar{d}_r/\bar{d}$ | decoded | HBM | time |
| Airline | 0.134 | 0.61 | 0.046 | 1.16 | 0.15 | 0.17 | 0.16 |
| Retail | 0.143 | 1.26 | 0.058 | 1.22 | 0.26 | 0.30 | 0.30 |
| AutomationBench | 0.077 | 0.25 | 0.202 | 1.52 | 0.46 | 0.54 | 0.38 |

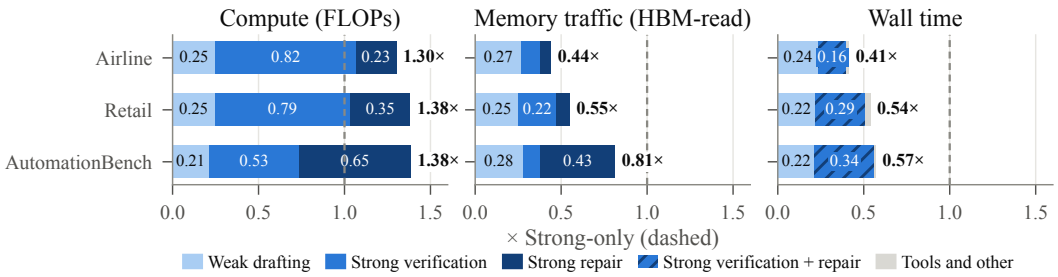


Figure 3: SPECAGENT's cost by phase, relative to Strong-only (dashed). Verification dominates compute but not memory traffic.

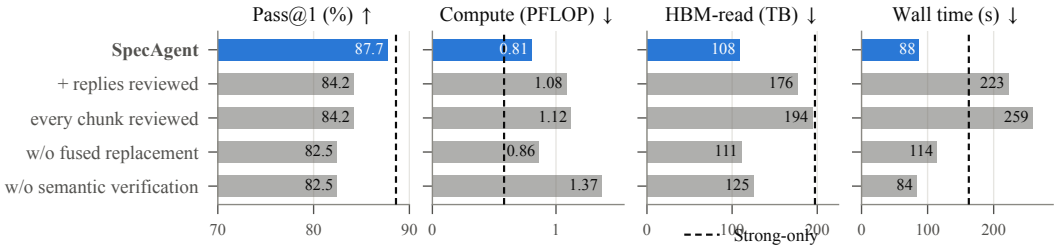


Figure 4: Design ablations on $\tau^2$ Retail, varied one at a time from SPECAGENT; SPECAGENT and Strong-only (dashed) are main-table runs.

*How to repair:* without the fused replacement, every rejection becomes a separate Strong turn, and Pass@1 drops to 82.5. *What to accept:* a verifier that accepts only the exact step the Strong model would take itself also lowers Pass@1 to 82.5 and raises FLOPs by 70%, because valid Weak steps are rejected and regenerated. Semantic acceptance thus appears to be what lets verification keep Weak progress without losing quality.

**Across Strong models.** The MCP-Atlas pairs vary the Strong model with the Weak model fixed (Table 3). SPECAGENT's quality tracks the Strong model's own (59.4, 67.3, and 74.3 against Strong-only scores of 72.3, 76.2, and 84.2), since the Strong model both judges the drafts and writes the repairs. Its token share falls as Strong-only steps grow longer, as the amortized verdict of Proposition 1 predicts: 50%, 46%, and 36% for 3.6k, 4.6k, and 10.6k Strong-only tokens per task. API cost falls less (to 59–81%) because each review resends the context as input, the prefill side of the trade.

**Deployment regimes.** The two propositions also say when the trade does not pay. The latency term $\kappa\, n(1-p_u)/(1-p_u^n)$ requires a Weak step to be faster than a Strong step, which does not hold in the edge–cloud deployment of Section 4.3. Assumption 1 holds while the Strong replica's decode batch stays below the compute–bandwidth balance point, about 37 sequences on H20; with larger batches decoding becomes compute bound, and the 30–38% increase in FLOPs would begin to cost time.

## 6 CONCLUSION

SPECAGENT turns the Strong model from the author of every agent step into a reviewer of grounded drafts: speculation runs freely while execution stays reversible, stops for review only at state-changing actions, and keeps valid progress rather than only agreement. Across $\tau^2$-Bench, AutomationBench, and MCP-Atlas, this retains most of Strong-only quality while cutting memory traffic, wall time, or remote cost, because Strong decoding is replaced by prefill. The gain grows with the Weak progress the Strong model accepts, so stronger Weak drafters and better-calibrated verifiers translate directly into further memory and latency savings without any change to the protocol.

### AI USE STATEMENT

We used large language model assistants to help write and debug experiment code, analysis scripts, and figure scripts, and to edit the manuscript's wording. The research questions, method, experimental design, and conclusions are the authors' own. Every reported number was produced by our evaluation and accounting code from logged runs, and the authors checked all code and text that an assistant produced. The Weak, Strong, and simulator models described in Section 4.1 are part of the system under study, not assistance in preparing the paper.

### ETHICS STATEMENT

This work uses public benchmarks whose environments are sandboxed or backed by public tool servers; no human subjects or personal data are involved. Agents that act on real systems can cause irreversible harm, and SPECAGENT addresses this directly by never executing a state-changing action before the Strong model has reviewed it. Its safeguards are only as good as the classification of actions and the verifier's judgment, so deployments should keep conservative action lists and human oversight for high-stakes operations.

### REPRODUCIBILITY STATEMENT

Section 3 and Algorithm 1 specify the protocol, and Appendix C proves its cost propositions. Section 4.1 gives the models, serving configuration, decoding settings, and task sets; Appendix D lists the per-benchmark adapter settings. Appendix B.3 and the surrounding sections define every resource estimate, including the prefix-reuse model, so the reported FLOPs, HBM-read, GPU·s, and wall time can be recomputed from request logs. All models are publicly available. We will release the code, prompts, and per-trajectory logs.

## REFERENCES

Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam. Automix: Automatically mixing language models. In *NeurIPS*, 2024. arXiv:2310.12963.

Anthropic. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol, 2024.

Chaithanya Bandi, Razvan-Gabriel Dumitru, Ben Hertzberg, Divyansh Agarwal, Geobio Boo, Tejas Polakam, Sami Hassaan, Jeff Da, HiJae Kim, Vipul Gupta, Manasi Sharma, Andrew Park, Martin Dimakis, Ernesto Gabriel Hernandez Montoya, Dan Rambado, Ivan Salazar, Rafael Cruz, MohammadHossein Rezaei, Chetan Rane, Ben Levin, Daniel Yue Zhang, Brad Kenstler, and Bing Liu. MCP-Atlas: A large-scale benchmark for tool-use competency with real MCP servers, 2026. URL https://arxiv.org/abs/2602.00933.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. $\tau^2$-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023a.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023b. Published in TMLR, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid llm: Cost-efficient and quality-aware query routing. In *ICLR*, 2024. arXiv:2404.14618.

Yichao Fu, Rui Ge, Zelei Shao, Zhijie Deng, and Hao Zhang. Scaling speculative decoding with lookahead reasoning. *arXiv preprint arXiv:2506.19830*, 2025.

Wenbo Gao, Renxi Liu, Xian Wang, Fang Guo, Shuai Yang, Xi Chen, Hui-Ling Zhen, Hanting Chen, Weizhe Lin, Xiaosong Li, and Yaoyuan Wang. Agentcollab: A self-evaluation-driven collaboration paradigm for efficient llm agents. *arXiv preprint arXiv:2603.26034*, 2026.

GLM-5 Team. GLM-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Google DeepMind. Gemini 3.1 Pro model card. https://deepmind.google/models/model-cards/gemini-3-1-pro/, 2026.

Yilin Guan, Qingfeng Lan, Sun Fei, Dujian Ding, Devang Acharya, Chi Wang, William Yang Wang, and Wenyue Hua. Dynamic speculative agent planning. *arXiv preprint arXiv:2509.01920*, 2025.

Raghu Vamshi Hemadri, Humaira Firdowse Mohammed, Rishabh Maheshwary, Srivatsava Daruru, Sagar Davasam, Vikas Yadav, Srinivas Sunkara, and Sai Rajeswar. R2V agent: Teaching SLMs when to ask for help. *arXiv preprint arXiv:2605.16604*, 2026.

Coleman Hooper, Minwoo Kang, Suhong Moon, Nicholas Lee, Eric Wen, John Wawrzynek, Michael W. Mahoney, Yakun Sophia Shao, Amir Gholami, and Kurt Keutzer. Speculative interaction agents: Building real-time agents with asynchronous I/O and speculative tool calling. *arXiv preprint arXiv:2605.13360*, 2026.

Wenyue Hua, Mengting Wan, Shashank Vadrevu, Ryan Nadel, Yongfeng Zhang, and Chi Wang. Interactive speculative planning: Enhance agent efficiency through co-design of system and user interface. *arXiv preprint arXiv:2410.00079*, 2024.

Zixiao Huang, Wen Zeng, Tianyu Fu, Tengxuan Liu, Yizhou Sun, Ke Hong, Xinhao Yang, Chengchun Liu, Yan Li, Quanlu Zhang, Guohao Dai, Zhenhua Zhu, and Yu Wang. Reducing latency of LLM search agent via speculation-based algorithm-system co-design. *arXiv preprint arXiv:2511.20048*, 2025.

Jiabao Ji, Yujian Liu, Li An, Rohit Jain, Gungor Polatkan, Siyu Zhu, and Shiyu Chang. Speculate while you reason: Teaching agents to predict their next tool call via joint agent-speculator RL. *arXiv preprint arXiv:2607.25816*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. *arXiv preprint arXiv:2211.17192*, 2023.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, 2024.

Daniel Nichols, Prajwal Singhania, Charles Jekel, Abhinav Bhatele, and Harshitha Menon. Optimizing agentic language model inference via speculative tool calls. *arXiv preprint arXiv:2512.15834*, 2025.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data. In *ICLR*, 2025. arXiv:2406.18665.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*, 2023.

Qwen Team. Qwen3.5: Towards native multimodal agents. https://qwen.ai/blog?id=qwen3.5, 2026.

Daniel Shepard and Robin Salimans. AutomationBench. *arXiv preprint arXiv:2604.18934*, 2026.

Jikai Wang, Juntao Li, Jianye Hou, Bowen Yan, Lijun Wu, and Min Zhang. Efficient reasoning for LLMs through speculative chain-of-thought. *arXiv preprint arXiv:2504.19095*, 2025a.

Zhihai Wang, Jie Wang, Jilai Pan, Xilin Xia, Huiling Zhen, Mingxuan Yuan, Jianye Hao, and Feng Wu. Accelerating large language model reasoning via speculative search. In *ICML*, 2025b. arXiv:2505.02865.

Jinbiao Wei, Kangqi Ni, Yilun Zhao, Guo Gan, and Arman Cohan. Step-level optimization for efficient computer-use agents. *arXiv preprint arXiv:2604.27151*, 2026.

Xiaohongshu Dots Studio and Evolvent AI. VibeLifeBench: Can your life agent be proactive and persistent in a living world? *arXiv preprint arXiv:2608.10875*, 2026.

Naimeng Ye, Arnav Ahuja, Georgios Liargkovas, Yunan Lu, Kostis Kaffes, and Tianyi Peng. Speculative actions: A lossless framework for faster AI agents. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=P0GOk5wslg.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, 2024.

Shuzhang Zhong, Baotong Lu, Qi Chen, Chuanjie Liu, Fan Yang, and Meng Li. DualSpec: Accelerating deep research agents via dual-process action speculation. *arXiv preprint arXiv:2603.07416*, 2026.

# A Experimental Setup

**Benchmarks.** $\tau^2$-Bench Airline (50 tasks) and Retail (114 tasks) are run with three independent trajectories per task. The simulated user is DeepSeek-V4-Flash-0731 (DeepSeek-AI, 2026), served locally with four-way tensor parallelism and shared by all methods. AutomationBench uses its 600 public tasks with one trajectory per task; its simple warm-up domain was used for development and is excluded.

MCP-Atlas uses 101 of its public tasks whose servers run without private API keys, excluding six on which our harness did not complete runs reliably, and runs one trajectory per task. The subset is somewhat easier than the full benchmark (4.5 versus 4.9 ground-truth tool calls and 3.4 versus 3.9 graded claims per task), so its absolute scores are not comparable to the public leaderboard; all methods are compared on the same tasks. Responses are graded by Gemini 3.1 Pro (Google DeepMind, 2026), and a task counts as solved when the response covers at least 75% of its graded claims.

**Models and serving.** On $\tau^2$ and AutomationBench, the Weak model, Qwen3.5-4B, is served with SGLang on a single H20 GPU, and the Strong model, Qwen3.5-397B-A17B, runs as a separate deployment with 16-way tensor parallelism on H20 GPUs. Agents use the servers' default sampling, with no output-length cap and at most 100 steps per trajectory. Thinking is enabled whenever a model generates actions, including Strong repair, and disabled for verification. Every method runs eight tasks concurrently against the same deployments; on Retail, the Weak requests of a trajectory are pinned to one replica. On MCP-Atlas, the Weak model runs on one local RTX 5090 GPU, and each Strong model is called through its provider's API; verification again runs without thinking.

**Baselines.** Weak-only and Strong-only run the same agent loop as SpecAgent with a single model. AgentCollab (Gao et al., 2026) follows the published algorithm: the Strong model acts for the first $K_L = 2$ steps, after which the Weak model acts and each action is followed by the paper's TRUE/FALSE progress check once its observation arrives. A FALSE check hands control to the Strong model for $\min(B_{\max}, B_0 + k\,l)$ steps, with $B_0 = 2$, $k = 2$, $B_{\max} = 6$, and $l$ the number of consecutive failed checks. Because $\tau^2$ agents have no think tool, the progress check runs as a separate call. Speculative Actions is described in Appendix E. On MCP-Atlas, where the Strong models are served through third-party APIs, we compare only against the single-model baselines.

# B Metric Definitions

This section defines every quantity in Table 2. All resource columns are means over the official trajectories of a method (Airline $n = 150$, Retail $n = 342$, AutomationBench $n = 600$) and cover the agent-model requests only; user simulation, judging, and tool serving are excluded.

## B.1 Task Quality

Let $y_{ir} \in \{0, 1\}$ be the benchmark-native success of trial $r$ of task $i$, with $N$ tasks and $R$ trials per task ($R = 3$ on $\tau^2$, $R = 1$ on AutomationBench). We report

$$\text{Pass@1} = \frac{1}{NR}\sum_{i=1}^{N}\sum_{r=1}^{R} y_{ir}, \qquad \text{pass@}R = \frac{1}{N}\sum_{i}\max_{r} y_{ir}, \qquad \text{pass}^R = \frac{1}{N}\sum_{i}\min_{r} y_{ir}. \quad (7)$$

Failed or incomplete trials stay in the denominator. For the paired difference $\Delta = \text{Pass@1}^{\text{SpecAgent}} - \text{Pass@1}^{\text{Strong}}$ we draw $N$ tasks with replacement, keep all $R$ trials of each drawn task for both methods, recompute $\Delta$, and report the 2.5th and 97.5th percentiles over 10,000 resamples.

## B.2 Request Shape and Prefix Reuse

Each request $q$ has $I_q$ input and $O_q$ output tokens, of which a prefix of $C_q$ input tokens is reused from earlier computation. The request then prefills $P_q$ new tokens, runs $D_q$ decode forwards, and

computes $L_q$ output logits:

$$P_q = I_q - C_q, \qquad D_q = \begin{cases} O_q - 1 & P_q > 0 \\ O_q & P_q = 0 \end{cases}, \qquad L_q = O_q, \tag{8}$$

since the prefill pass already emits the first output token. Serving caches measure reuse only partially and are affected by eviction, so we assign $C_q$ with a model of radix prefix caching (Zheng et al., 2024), applied identically to every method. Requests are grouped into sessions that share task, trial, model, role, and serving replica; reuse never crosses sessions. Within a session, requests are processed in order with the rules of Table 5.

Table 5: Prefix-reuse rules. $M$, $B^v$, and $B^c$ are running maxima kept per session and updated after each request.

| Request family | Newly processed tokens $P_q$ | Rationale |
|---|---|---|
| Weak draft, Strong act | $\max(0,\ I_q - M)$ | append-only transcript |
| Strong verification | $c_q + \max(0,\ I_q - c_q - B^v)$ | candidate $c_q$ is new text |
| Strong repair | $300 + \max(0,\ I_q - 300 - B^c)$ | rejection note is new text |

For verification, the candidate length $c_q$ is the Weak output of the same round (the method's median when unavailable); the remaining backbone of the prompt can be reused from earlier verifications. A repair prompt adds a rejection note, for which we assume 300 tokens because note text is not logged. Verification and repair prompts are not assumed to share a prefix with each other.

### B.3 Estimated FLOPs

We count a multiply–add as two FLOPs, so a matrix product of shapes $m \times k$ and $k \times n$ costs $2mnk$. For $T$ token positions whose full-attention queries see $\Pi$ causal query–key pairs in total, one pass through the decoder stack costs

$$F_{\text{stack}}(T, \Pi) = \sum_{\ell \in \mathcal{L}} \Big[ F_{\text{mlp}}(T) + \mathbf{1}[\ell \in \mathcal{L}_{\text{full}}] \big( F_{\text{proj}}(T) + 4 n_h d\, \Pi \big) + \mathbf{1}[\ell \in \mathcal{L}_{\text{lin}}] F_{\text{lin}}(T) \Big], \tag{9}$$

where $4 n_h d\, \Pi$ covers the score and value products of $n_h$ heads of dimension $d$. The components are

$$F_{\text{mlp}}(T) = \begin{cases} 6Thm & \text{dense SwiGLU} \\ 2ThE + 6Thm_s + 6k\,Thm_e & \text{MoE: router, shared, and } k \text{ routed experts} \end{cases} \tag{10}$$

$$F_{\text{proj}}(T) = 2Th\,(2n_h d + 2n_{kv} d) + 2T\,n_h d\,h, \tag{11}$$

$$F_{\text{lin}}(T) = 2Th\,(2d_K + 2d_V + 2n_v) + 2T\,(2d_K + d_V)\kappa + 2Thd_V + 10\,T\,n_v d_k d_v, \tag{12}$$

where $h$ is the hidden size, $m$ the dense MLP width, $E$, $k$, $m_e$, and $m_s$ the number of experts, active experts, and routed and shared expert widths, $n_{kv}$ the number of KV heads, $n_k$ and $n_v$ the Gated DeltaNet key and value heads with dimensions $d_k$ and $d_v$, $d_K = n_k d_k$, $d_V = n_v d_v$, and $\kappa$ the convolution width (Table 6). The query projection is doubled by the output gate, and the last term of Equation 12 a token-linear proxy for the five Gated DeltaNet state operations (decay, state-key read, delta, rank-one update, and query readout). A request then costs

$$F_q = F_{\text{stack}}\big(P_q,\ P_q C_q + \tfrac{P_q(P_q+1)}{2}\big) + F_{\text{stack}}\big(D_q,\ D_q I_q + \tfrac{D_q(D_q+1)}{2}\big) + 2L_q V h, \tag{13}$$

for prefill, decode, and the output head. Only the $k$ active experts enter the MoE term, and FLOPs are not multiplied by the number of serving GPUs. RMSNorm, elementwise activations, softmax, rotary embeddings, multi-token prediction, and inter-GPU communication are not counted.

### B.4 Estimated HBM-Read

HBM-read estimates the bytes a request reads from GPU memory when executed alone (batch size one):

$$B_q = B_q^{\text{w}} + B_q^{\text{kv}} + B_q^{\text{state}} + B_q^{\text{act}}. \tag{14}$$

Table 6: Architecture parameters used by the estimators (text backbone).

| | Qwen3.5-4B (Weak) | Qwen3.5-397B-A17B (Strong) |
|---|---|---|
| Hidden size $h$ / vocabulary $V$ | 2560 / 248,320 | 4096 / 248,320 |
| Layers: full attention / Gated DeltaNet | 8 / 24 | 15 / 45 |
| Query / KV heads, head dim $d$ | 16 / 4, 256 | 32 / 2, 256 |
| DeltaNet value heads $n_v$, $d_k$=$d_v$ | 32, 128 | 64, 128 |
| MLP | dense, $m = 9216$ | MoE, $E = 512$, $k = 10$ |
| Expert / shared width $m_e$ / $m_s$ | – | 1024 / 1024 |
| Weights / KV / recurrent state | BF16 / BF16 / FP32 | BF16 / BF16 / FP32 |

Weights are read once for the prefill pass and once per decode forward. With $W(e)$ the weight bytes of the stack when $e$ routed experts per layer are touched,

$$B_q^{\mathrm{w}} = \mathbf{1}[P_q > 0]\, W(U_q) + D_q\, W(k), \qquad k \le U_q \le \min(E,\, kP_q), \tag{15}$$

where $U_q$ is the unknown number of distinct experts a layer touches during prefill; its two bounds give the reported interval. For the dense model, $W$ does not depend on $e$ and the interval collapses. The remaining terms are

$$B_q^{\mathrm{kv}} = \left(D_q I_q + \tfrac{D_q(D_q+1)}{2}\right) \cdot 2n_{kv}d \cdot 2\,\mathrm{B} \cdot |\mathcal{L}_{\mathrm{full}}|, \tag{16}$$

$$B_q^{\mathrm{state}} = (P_q + D_q) \cdot n_v d_k d_v \cdot 4\,\mathrm{B} \cdot |\mathcal{L}_{\mathrm{lin}}|, \tag{17}$$

$$B_q^{\mathrm{act}} = (P_q + D_q) \cdot h \cdot 2\,\mathrm{B} \cdot |\mathcal{L}|, \tag{18}$$

for the attention cache read during decoding, the recurrent state, and a token-linear activation proxy. HBM-read excludes writes and inter-GPU communication and is not multiplied by the number of serving GPUs. It is an analytical estimate, not a hardware counter measurement. As a spot check, Nsight Compute DRAM counters for the Weak model on one H20 GPU at batch size one agree with the estimated weight and KV reads to within 5% on a held-out request, and within 10% across short request shapes; we did not profile the Strong deployment.

### B.5 Aggregation, GPU Time, Wall Time, and Calls

For a method with official trajectories $i = 1, \ldots, n$ and successful generation requests $\mathcal{Q}_i$, the table reports

$$\overline{F} = \frac{1}{n}\sum_{i=1}^{n}\sum_{q\in\mathcal{Q}_i} F_q, \qquad \overline{B} = \frac{1}{n}\sum_{i}\sum_{q\in\mathcal{Q}_i} B_q, \qquad \overline{G} = \frac{1}{n}\sum_{i}\sum_{q\in\mathcal{Q}_i} t_q N_q, \tag{19}$$

in units of $10^{15}$ FLOPs, $10^{12}$ bytes, and GPU-seconds. Here $t_q$ is the observed request duration and $N_q$ the GPUs serving it: 16 for the Strong model (tensor parallelism 16) and one for each Weak request. $\overline{G}$ charges every request its full duration on the GPUs that serve it, as if it ran alone. It is applied identically to all methods, but because concurrent requests share a deployment and overlap in time, it is an upper bound on actual occupancy; we use it for relative comparison. HBM intervals sum the per-request bounds and are rounded outward.

All methods run eight tasks concurrently. Wall time is the mean elapsed time per trajectory, from task start to agent completion on $\tau^2$, including tool calls and user-simulator turns, and from the first model call to scoring end on AutomationBench. Calls are the mean number of successful generation requests per trajectory, split by the Weak and Strong model; transport retries of one logical request count once.

### B.6 Supplementary Results

Table 7 reports pass@3 and pass$^3$ for the $\tau^2$ single-model baselines over the same three trials per task.

Table 7: Completed $\tau^2$-Bench solo baselines. Pass@1 is mean binary reward per attempt; pass@3 requires at least one successful attempt per task, and pass$^3$ requires all three. Scores are percentages. Airline contains 50 tasks and Retail 114 tasks, each with three attempts.

| Domain | Method | Pass@1 (%) ↑ | Pass@3 (%) ↑ | Pass$^3$ (%) ↑ |
|---|---|---|---|---|
| Airline | Weak-only (4B) | 80.0 | 96.0 | 64.0 |
| | Strong-only (397B) | 86.0 | 96.0 | 72.0 |
| Retail | Weak-only (4B) | 80.4 | 95.6 | 61.4 |
| | Strong-only (397B) | 88.6 | 96.5 | 76.3 |

## C Proofs for Section 3.3

Fix a task $u$ and write $p = p_u$. Consider one cycle: a chunk of length $n$ is drafted, verified, and either committed in full or cut at the first rejected step and repaired. Let $S$ be the number of steps the cycle commits. Under Assumption 2, the cycle commits at least $k \le n$ steps exactly when the first $k-1$ drafted steps are accepted: if $j = n$ all $n$ steps are committed, and if $j < n$ the $j$ accepted steps and the repair are. Hence $\Pr(S \ge k) = p^{k-1}$ for $1 \le k \le n$ and

$$\mathrm{E}[S] = \sum_{k=1}^{n} p^{k-1} = \frac{1-p^n}{1-p}. \tag{20}$$

A cycle needs a repair when $j < n$, with probability $1 - p^n$.

**Proof of Proposition 1.** By Assumption 1, the Strong cost of a cycle, in weight passes, is $v$ for the verdict plus $d$ for a separate repair turn, which occurs with probability $\rho(1-p^n)$. Its expectation is $v + \rho(1-p^n)d$. Within the task, cycles are independent and identically distributed, so by the renewal–reward theorem the Strong cost per committed step is this expectation divided by $\mathrm{E}[S]$. Dividing by the cost $d$ of a Strong-only step gives

$$R_u = \frac{v + \rho(1-p^n)d}{d} \cdot \frac{1-p}{1-p^n} = \frac{1-p}{1-p^n} \cdot \frac{v}{d} + \rho(1-p).$$

Across tasks, SpecAgent spends $T_u R_u d$ weight passes on task $u$ and Strong-only spends $T_u d$, so the ratio of totals is $\mathrm{E}_u[T_u R_u]/\mathrm{E}_u[T_u]$.

**Proof of Proposition 2.** The cycle's Strong time is proportional to its weight passes, as above. The Weak model drafts all $n$ steps before the verdict, taking time $n\kappa$ in units of a Strong-only step. Calls within a cycle run sequentially, so the cycle's expected time is $n\kappa + v/d + \rho(1-p^n)$ in the same units, and dividing by $\mathrm{E}[S]$ yields $L_u = R_u + \kappa\, n(1-p)/(1-p^n)$. The ratio over tasks follows as for $R$. □

## D Benchmark Adapters

All benchmarks run the protocol of Algorithm 1; an adapter $E$ binds only its benchmark-specific parts. Three bindings are shared. A fused replacement is accepted only if it matches the tool's argument schema and executes in the environment ($E$.Valid); otherwise the Strong model takes one repair turn. The verifier sees committed actions and observations but not the Weak model's reasoning. Table 8 lists the bindings that differ.

Table 8: Adapter bindings that differ across benchmarks.

| | $\tau^2$-Bench | AutomationBench | MCP-Atlas |
|---|---|---|---|
| Held ($E$.StateChanging) | write tools, transfer | write tools, final message | write tools, final answer |
| Chunk size $K$ | 6 | 8 | 6 |
| Verifier context | full history | full history | old tool results truncated |

**Choices.** Tools are classified as reads or writes from each benchmark's documentation and their observed effect on the environment. Rejecting a read is a logical de-commitment: the read is removed from the agent's history and from what the Strong model reviews, but an external service may still keep traces of it, such as access logs. The guarantee is therefore that no action classified as state-changing executes before review, not that every effect of speculation is physically rolled back. Likewise, a reply sent to the $\tau^2$ simulated user cannot be undone. On $\tau^2$-Bench, replies to the simulated user are not held: they leave the database unchanged, the user's next turn returns as an observation, and holding them more than triples Strong calls without improving quality (Figure 4). On AutomationBench and MCP-Atlas the final message is the graded answer, so it is held like a write, and on MCP-Atlas it is reviewed against the evidence gathered by earlier reads. MCP-Atlas tool outputs can be long, so older tool results are truncated in the verifier's view. $K$ bounds how many reads accumulate before an unreviewed commit.

### D.1 Prompts

On $\tau^2$, Weak drafting and the single-model baselines use the benchmark's default agent prompt with the domain policy. The Strong verifier and the separate repair turn use the prompts below, where {domain} and {domain_policy} are filled with the domain name and the benchmark's policy text. The Retail run uses these prompts as shown; the Airline run uses an earlier revision of the verifier prompt that differs only in how pending steps are defined. The other benchmarks use the same structure with their own policies and tools. All prompts are released with the code.

**Verifier.**

```
You are the Strong trajectory reviewer for a Weak-led customer-service agent
in the "{domain}" domain. Assess whether the draft is a justified continuation,
retain its longest valid prefix, and redirect it when necessary.

1. INPUT AND EVIDENCE

- Input: task, confirmed_history, proposed_trajectory, candidate_length K,
  invoked_tool_contracts, and the domain policy below.
- Candidate steps are indexed 1..K. pending=true means the step has not
  happened (a staged mutation or an unsent reply). A step without pending
  has already executed; its observation is real.
- Ground judgments in original observations. Assistant hypotheses and
  review notes are not factual evidence.

Domain policy:
{domain_policy}

2. REVIEW THE DIRECTION

Evaluate the chunk against the customer's request, the policy, and visible
history. Notice ignored contradictions, mutations whose preconditions are
unmet, replies that claim work that was not done, and unproductive repeats.

Customer-service replies:
- Asking for a user id, reservation id, date, or name that is not yet in
  the task, customer messages, or tool observations is a justified
  continuation. Do not reject it for failing to finish the request.
- A reply that promises a cancellation, refund, booking change, waiver,
  or other state change is a commit. Reject it unless that mutation is
  already in confirmed_history or this chunk with a successful observation.
- When the policy requires explicit customer confirmation before acting,
  a draft that mutates without that answer is unjustified. "The customer
  implied yes" is not confirmation.

Policy:
- A mutation or a denial is justified only by a sentence in the domain
  policy above. If you cannot point to that sentence, do not invent a rule.
- If two policy-legal outcomes remain, do not reject the draft for picking
```

```
  one unless the conversation plus policy uniquely selects the other.

Executed reads (pending=false lookups):
- The observation is already real. Rejecting the read cannot undo it; it
  only discards later steps. Do not zero- or partial-accept a reads-only
  prefix because a later mutation would be better, because the agent is
  "still searching", or because the matching reservation has not turned
  up yet. Searching the customer's existing reservations for the one they
  described is justified.
- Reject an executed read only when the tool or required arguments are
  invalid: unknown tool, missing required argument, or an identifier that
  never appeared in the task, customer messages, or prior observations.

Pending mutations (pending=true tool calls):
- Judge the call as if it will run. Wrong user id, reservation id, payment
  id, amount, cabin, or date is a reject even if a later undo exists.

3. RETAIN THE LONGEST VALID PREFIX

Set accepted_steps to the largest j for which steps 1..j form a justified
continuation, stopping before the first unjustified action.
- Judge each action using evidence available before it.
- j applies only to the candidate chunk. j=0 discards that chunk, not
  confirmed_history.
- Stop at unknown tools, invalid required arguments, policy-illegal
  mutations, or a reply that ends the turn with required work missing.

4. EXPLAIN THE REJECTION WHEN j < K

Write reason as a concise, evidence-based decision brief: what is wrong,
why continuation stops at this prefix, and what the next step should
establish. The next actor receives confirmed_history, steps 1..j, and
reason, without the discarded suffix. Do not present facts found only in
the discarded suffix as retained evidence.
If you also emit replacement_action, reason only diagnoses the discarded
suffix; it must not assign extra work to a second Strong actor.

5. PROPOSE ONE TOOL ACTION OR HAND OFF

- If you can specify a useful next tool call from the retained context,
  you MUST return it as replacement_action: {"tool": "...", "arguments": {...}}.
  Returning null spends a second Strong call. Use null only when the next
  step is a customer reply, or you cannot fill schema-valid arguments from
  the retained prefix.
- This field is tool-only: one call, a supplied tool, complete schema-valid
  arguments. Mutating tools are allowed when the policy's preconditions
  are already satisfied by the retained prefix.
- Do not return null "to be safe" when a lookup tool already has its
  arguments in the retained observations.

6. OUTPUT

Return exactly one JSON object with exactly these three keys:
{"reason": <string or null>, "accepted_steps": <integer 0..K>,
 "replacement_action": <one tool action or null>}
- Full acceptance (j == K): reason and replacement_action must both be null.
- Partial acceptance or rejection (j < K): reason must contain the diagnosis;
  replacement_action follows section 5.
- Do not emit rejection_code, corrector, or any other key.
```

**Repair.**

```
You are the Strong customer-service agent continuing from a reviewed
trajectory. Reassess the conversation and produce exactly one useful next
step. You are invoked only when replacement_action was null or failed the
```

```
schema guard; do not re-verify the retained prefix.

1. INPUT AND EVIDENCE

- You receive the original task and tools, the retained dialogue, and
  review.reason. The retained dialogue is confirmed history plus the
  accepted prefix, with actions and original observations.
- Treat the review as a diagnosis to examine, not as new factual evidence.
  Check its material claims against retained observations; when they
  conflict, follow the observations.

2. CHOOSE ONE STEP

- Reply when the required work for this customer turn is done, or when
  the policy requires asking the customer something you do not have.
- Prefer a missing lookup or a required confirmation question over a
  speculative mutation.
- Otherwise issue one tool call targeting the blocking evidence gap or
  the policy-legal mutation whose preconditions are already met.
- Follow the supplied schema. Do not invent identifiers.

3. OUTPUT

Produce exactly one assistant step: either one tool call or one reply.
Do not mention the reviewer, the discarded draft, or this protocol.
```

## E SPECULATIVE ACTIONS ON $\tau^2$-BENCH

Speculative Actions (Ye et al., 2026) targets latency rather than Strong-model work. An authoritative Actor produces every committed step, while a faster Speculator predicts upcoming environment steps and executes them in advance. In the authors' $\tau$-bench retail setting, the Speculator predicts the user's next query and the API calls needed to answer it; the paper describes validating these guesses once the real query arrives and restricting speculative side effects to idempotent, reversible, or sandboxed actions, and it reports API-prediction accuracy rather than task success.

We ported the authors' released $\tau$-bench agent to $\tau^2$-Bench with our model pair. The Strong model acts at every step. Whenever it replies to the user, the Weak model role-plays the next user message in the background, and a Strong decider continues from that guess for up to five tool steps; the resulting tool calls and observations are added to the context on the next user turn. As in the released code, these background tool calls execute in the live environment without a read-only filter, and their results are added without checking the guess against the real user message.

This configuration reached 72.0% Pass@1 on Airline and 48.5% on Retail, below Weak-only (80.0% and 80.4%), while issuing 17.4 and 17.1 Strong calls per trajectory, more than Strong-only (13.8 and 13.1). The extra Strong calls come from the background decider, which runs on top of the Strong actor. Because background calls may change environment state and add observations on the basis of a guessed user message, the released implementation is not lossless in this setting, which plausibly explains part of the quality drop; we did not diagnose it further. These numbers therefore characterize the released implementation rather than the guarded design described in the paper.