
SPECULATIVE ACTIONS: A LOSSLESS FRAMEWORK FOR FASTER AGENTIC SYSTEMS

Naimeng Ye*, Arnav Ahuja*, Georgios Liargkovas*, Yunan Lu*, Kostis Kaffes, Tianyi Peng
Columbia University
New York, New York, USA
{ny2336, aa5790, gl2902, yl4021, kk3664, tp2845}@columbia.edu

ABSTRACT

Despite growing interest in AI agents across industry and academia, their execution in an environment is often slow, hampering training, evaluation, and deployment. For example, a game of chess between two state-of-the-art agents may take hours. A critical bottleneck is that agent behavior unfolds sequentially: each action requires an API call, and these calls can be time-consuming. Inspired by speculative execution in microprocessors and speculative decoding in LLM inference, we propose *speculative actions*, a lossless framework for general agentic systems that predicts likely actions using faster models, enabling multiple steps to be executed in parallel. We evaluate this framework across three agentic environments: gaming, e-commerce, web search, and a “lossy” extension for an operating systems environment. In all cases, speculative actions achieve substantial accuracy in next-action prediction (up to 55%), translating into significant reductions in end-to-end latency. Moreover, performance can be further improved through stronger guessing models, top- K action prediction, multi-step speculation, and uncertainty-aware optimization, opening a promising path toward deploying low-latency agentic systems in the real world.

1 INTRODUCTION

Large language model (LLM)-driven agents are shifting from single-shot predictions to processes that run inside rich environments: browsers, operating systems, game engines, e-commerce stacks, and human workflows. These environments are not incidental; they determine what the agent can observe and do, gate progress through interfaces and rate limits, and dominate end-to-end latency. In practice, the agent’s behavior unfolds as a sequence of environment steps (tool calls, MCP server requests, human-in-the-loop queries, and further LLM invocations), each with non-trivial round-trip time and cost. As capabilities improve, a new bottleneck emerges: time-to-action in the environment. Even when accuracy is high, an agent that pauses too long between steps is impractical for interactive use or high-throughput automation.

OS Tasks (Abhyankar et al., 2025)	Deep Research (OpenAI, 2025)	Data Pipeline (Jin et al., 2025)	Kaggle Chess Game (Kaggle, 2025)
10–20 min	5–30 min	30–45 min	1 hour

Table 1: **Estimated time state-of-the-art AI agents spend on various tasks/environments.**

As shown in Table 1, AI agents may require tens of minutes to hours to complete a single run across different environments, a cost that grows significantly when hundreds or thousands of iterations are needed for reinforcement learning or prompt optimization (Agrawal et al., 2025).

Fundamentally, this inefficiency arises from the inherently sequential nature of API calls. Thus, we ask a simple question in this paper:

Must an agent interact with its environment in a strictly sequential manner?

*Equal contribution

Our answer is no. Inspired by speculative execution in microprocessors and speculative decoding for LLM inference, we propose *speculative actions*: a general framework that allows agents to predict and tentatively pursue the most likely next actions using faster models, while slower ground-truth executors (powerful LLMs, external tools, or humans) catch up. In effect, the agent stages environment interactions (prefetching data, launching safe parallel calls, and preparing reversible side effects) so that validation, not waiting, is the critical path. When those slower evaluators confirm the guesses, progress has already been made; when they disagree, we execute as usual. The result is an *as-if-sequential, lossless interface with parallel, opportunistic internals*.

Concretely, in such agents, speculative actions introduce two roles in the environment loop:

- *Actor(s)*: authoritative but slower executors (e.g., more capable LLMs, external APIs/tools, the environment’s own responses, or humans) whose outputs materialize the ground truth for correctness and side effects.
- *Speculator(s)*: inexpensive, low-latency models that predict the next environment step, i.e., the action, its arguments, and the expected observation or state delta. Examples include smaller LLMs, simplified use of the same LLM with reduced prompts and reasoning steps, and domain heuristics.

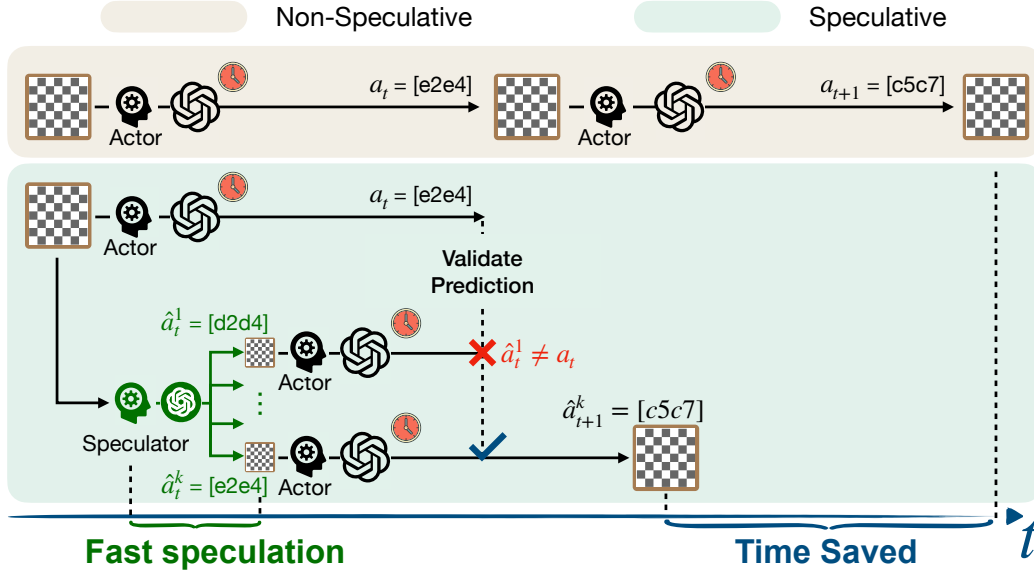


Figure 1: Illustration of our framework in a chess-playing environment. While the Actor issues an LLM call to decide the next move, the Speculator uses a faster model to guess it. These guesses enable parallel API calls for the next steps, and once a guess is verified, the system gains time through parallelization. The process runs in the backend, ensuring a lossless speedup for the user.

A key design goal is losslessness relative to the environment’s baseline semantics: speculative actions should not degrade final outcomes versus a strictly sequential agent. We achieve this with (a) semantic guards (actors confirm equivalence of state transitions before commit), (b) safety envelopes (only idempotent, reversible, or sandboxed speculative side effects), and (c) repair paths (rollback or compensating actions when a guess is rejected). In many environments (e.g., web search pipelines, shopping carts before checkout, and OS-level operations in a sandbox) these patterns are natural and inexpensive to implement.

Can we guess the next API calls of agents? We show that, in practice, API intents can often be guessed with reasonable accuracy. In particular, we demonstrate speculative actions across four environments, each highlighting different aspects of agent latency:

- **Turn-based gameplay** (e.g., chess): the Speculator can predict the opponent’s move while waiting for its turn. See Fig. 1.

- **E-commerce:** while conversing with a human shopper, the Speculator can proactively infer the shopper’s intent (e.g., returning an item), and safely trigger tool calls in advance (e.g., checking return eligibility).
- **Multi-hop web search:** while awaiting results from slow external calls (e.g., Wikipedia), the Speculator can guess answers from its knowledge base, and execute subsequent search queries.
- **Operating systems (lossy extension):** speculative, reversible actions react immediately to workload and environment changes, boosting end-to-end application performance while actors confirm.

Across these settings, we observe substantial acceleration—up to 55% accuracy in predicting the next API calls and 20% end-to-end lossless speedup. These results are achieved with a simple implementation using single-step speculation, and can be improved upon with advanced techniques such as adaptive speculation. We make our code publicly available at <https://github.com/naimengye/speculative-action>.

1.1 RELATED WORK

Speculation in microprocessors Speculation emerged in computer architecture to increase parallelism by executing instructions before their outcomes were resolved (Tomasulo, 1967), rolling back when predictions were wrong and became central to high-performance processors (Lam & Wilson, 1992). In light of security vulnerabilities that exploit microarchitectural speculative execution, (Mambretti et al., 2019) developed Speculator to analyze CPU speculation behavior.

Speculative program execution Thread-level speculation executes sequential program fragments in parallel. It assumes that those program fragments are independent and can be safely parallelized, but must roll back and resolve conflicts if it turns out some threads are data dependent (Estebanez et al., 2016). Recently, (Liargkovas et al., 2023) explored the use of tracing and containment to speculatively but safely run shell scripts out of order.

Speculative decoding in LLM inference The same predict-verify pattern was recently applied to large language models. Speculative decoding accelerates autoregressive inference by using a small draft model to propose tokens that a larger target model verifies in batches, committing correct ones and regenerating failures (Leviathan et al., 2023; Zhang et al., 2024; Chen et al., 2023). At the reasoning level, speculation has also been used to accelerate chain-of-thought processes (Wang et al., 2025b;a; Fu et al., 2025). In all cases, speculation reduces latency in sequential pipelines by executing likely future steps in parallel with their validation.

Other applications of speculations Speculation has been widely applied in a number of systems contexts. For example, Speck (Nightingale et al., 2008) uses it to parallelize otherwise sequential security checks, and AutoBash (Su et al., 2007) uses it to test configuration changes in isolation. More recently, Farias et al. (2024) developed a speculative, GPU-friendly policy simulation technique for optimizing supply chain systems.

Motivated by these work, we propose a speculative framework for agentic systems to achieve lossless execution speedups. Our approach is aligned with recent studies that speculate LLM API calls for planning tasks (Hua et al. (2024), Guan et al. (2025)), but generalizes the idea to the **entire agentic environment**—by not only speculating LLM calls, but also internal and external tool APIs, MCP-server APIs, and even human responses. This yields a unified framework for agentic speculation, particularly consistent with the emerging “environment” and MCP perspectives on agentic systems, providing a powerful mechanism for breaking the bottleneck in the training and deployment of AI agents.

2 FRAMEWORK

An agentic system is usually modeled as a Markov Decision Process (MDP) (s_t, a_t) , where s_t denotes the state and a_t the agent’s action at step t . This model admits considerable flexibility: an action may represent a chatbot response, the choice of a tool to invoke, or a button clicked by a computer-use agent, among others.

From a systems perspective, we model each action in an agentic system as an *API call*, which may block execution until a response is returned. This abstraction offers two key advantages: (1) it precisely defines what constitutes an action, and (2) it provides a unified framework for optimizing system latency, as we will see shortly. Notably, this perspective aligns with the recent development of MCP servers for agentic systems (Anthropic, 2024).

Formally, at each step t , the policy π maps the current state s_t to an API call:

$$(h_t, q_t) \leftarrow \pi(s_t),$$

where h_t specifies the target API to invoke and q_t its associated parameters. We write

$$\bar{a}_t \leftarrow h_t(q_t) \quad a_t \leftarrow \text{await}(\bar{a}_t)$$

to denote an asynchronous API invocation that returns a *future* (a pending action), and the *await* for when the response actually arrives. We use the bar notation (e.g., $\bar{a}$) for futures and a cache $C : (h, q) \mapsto \bar{a}$ that maps an API call specifier to its pending response. The left squiggly arrow indicates an asynchronous call with non-negligible delay.

The system subsequently transitions to the next state via a transition function f : $s_{t+1} \leftarrow f(s_t, a_t)$. As a concrete example, consider chess: the policy π determines how to construct the prompt based on the current board state, a_t corresponds to the move proposed by the LLM’s response, and f updates the board configuration accordingly. Note that the LLM call is the API, its *response* is the move a_t .

This formulation subsumes a broad range of realizations:

- **LLM calls:** each invocation of an LLM within the agent can be treated as an action.
- **Tool / MCP server calls:** each actual call for internal/external tools is treated as an action: e.g., terminal access, web search, deep research APIs, weather APIs, or browser-use MCPs.
- **Human-as-an-API calls:** furthermore, human responses themselves can be abstracted as API calls, often incurring even longer latencies than automated tools.

Given this abstraction, the fundamental bottleneck in executing agentic systems becomes apparent: each API call must complete before the next can be issued. To break this sequential dependency, we propose to **speculate a set of API responses** $\{\hat{a}_t\}$ using a faster model while waiting for the true response a_t . This enables speculative API calls for step $t + 1$ to be launched in parallel. At time t , if the API call (h_t, q_t) can be found in the cache (cache hit), the system can skip the actual invocation and only wait for the pending action corresponding to this call to return (if not already returned). Formally, the algorithm is specified in Algorithm 1.

The resulting speedup relies on two key assumptions:

Assumption 1 (Speculation accuracy). *The speculative model $\hat{g}$ can guess the current-step response a_t accurately enough that the implied next call $(h_{t+1}, q_{t+1}) = \pi(f(s_t, \hat{a}_t))$ matches the true next call with non-zero probability $p > 0$.*

As shown later, this often holds in practice because API responses are typically predictable.

Assumption 2 (Concurrent, reversible pre-launch). *Multiple API calls can be launched concurrently, and pre-launched calls that do not correspond to the realized trajectory have no externally visible side effects (or can be rolled back).*

In practice, this assumption is satisfied under modest traffic for many external APIs (e.g., web search, OpenAI LLM queries, email lookups). For self-hosted LLMs, concurrent calls also incur only minimal additional cost due to continuous batching.

We can then establish the following result (with proof deferred to the Appendix).

Proposition 1. *Under Assumptions 1–2, suppose at each step the speculative branch implies the correct next call (h_{t+1}, q_{t+1}) with probability p , independently across $t \in [1, T - 1]$. Let the latency of $\hat{g}$ be $\text{Exp}(\alpha)$ and the latency of the actual API call be $\text{Exp}(\beta)$ with $\beta < \alpha$. All latencies and guesses occur independently. Assume the transition f and API parameter construction π are negligible. Then the ratio between the expected runtime of Algorithm 1, denoted $\mathbb{E}[T_s]$, and the expected runtime of sequential execution, $\mathbb{E}[T_{\text{seq}}]$, is*

$$\frac{\mathbb{E}[T_s]}{\mathbb{E}[T_{\text{seq}}]} = 1 - \frac{1}{T} \frac{\alpha}{\alpha + \beta} \left[\frac{(T-1)p}{1+p} + \frac{p^2}{(1+p)^2} - \frac{p^2}{(1+p)^2} (-p)^{T-1} \right] \xrightarrow{T \rightarrow \infty} 1 - \frac{p}{1+p} \cdot \frac{\alpha}{\alpha + \beta}$$

Algorithm 1 Speculative actions with k -way parallel next calls

Require: Initial state s_0 , horizon T , transition f , policy π , predictor $\hat{g}$, cache C . We use $\bar{a}$ to denote pending action.

```
1: for  $t = 0$  to  $T - 1$  do
2:   Policy:  $(h_t, q_t) \leftarrow \pi(s_t)$ 
3:   if  $(h_t, q_t) \in C$  then ▷ Cache hit
4:      $\bar{a}_t \leftarrow C[(h_t, q_t)]$ 
5:      $a_t \leftarrow \text{await}(\bar{a}_t)$  ▷ Await pending action if not returned already
6:      $s_{t+1} \leftarrow f(s_t, a_t)$ 
7:     continue
8:   end if
9:   Actor: Issue real request (returns future):  $\bar{a}_t \leftarrow h_t(q_t)$ 
10:  Speculator:  $\{\hat{a}_t^{(i)}\}_{i=1}^k \leftarrow \text{await}(\hat{g}(s_t, (h_t, q_t)))$  ▷ Actor and speculator run in parallel
11:  for  $i = 1$  to  $k$  do ▷ One-step speculative rollout per guess
12:     $\hat{s}_{t+1}^{(i)} \leftarrow f(s_t, \hat{a}_t^{(i)})$ 
13:     $(\hat{h}_{t+1}^{(i)}, \hat{q}_{t+1}^{(i)}) \leftarrow \pi(\hat{s}_{t+1}^{(i)})$ 
14:    Pre-launch:  $\bar{a}_{t+1}^{(i)} \leftarrow \hat{h}_{t+1}^{(i)}(\hat{q}_{t+1}^{(i)})$  ▷ Return pending action, hence non-blocking
15:     $C[(\hat{h}_{t+1}^{(i)}, \hat{q}_{t+1}^{(i)})] \leftarrow \bar{a}_{t+1}^{(i)}$  ▷ Cache speculative pending actions
16:  end for
17:  Wait for resolved  $a_t$  from Actor:  $a_t \leftarrow \text{await}(\bar{a}_t)$ 
18:   $s_{t+1} \leftarrow f(s_t, a_t)$ 
19: end for
```

Proof of this proposition is given in Section A. Proposition 1 suggests an upper bound of the end-to-end latency reduction of 50%, in an ideal scenario with $p = 1$ and $\alpha = \infty$, but the bound can be further improved by the multi-step extension below.

Extension. Algorithm 1 is only a simple demonstration of the speculative action idea. For example, one can naturally extend Algorithm 1 to *multi-step speculation*, where the Speculator predicts not only the next step but up to s steps ahead. This yields a tree-search structure with deeper rollouts. This can be further combined with *adaptive speculation*: instead of generating k guesses for a_t uniformly, the Speculator also estimates confidence for each guess (e.g., via prompting LLMs or uncertainty-quantification methods). The most promising branches can then be expanded in a beam-search-like manner. Together, these ideas highlight the richness of speculative actions, which we leave for future deeper investigation. Despite Algorithm 1’s simplicity, the results from the four use cases in the following sections are already highly promising.

Cost-latency tradeoff. Performing more speculative API calls (e.g., increasing k) improves accuracy but also raises costs when pricing is based on the number of calls or tokens. Although cost is not the focus of this work, we provide an analysis of our experiments in Appendix B. For self-hosted LLMs, this increased cost is largely mitigated through batching.

Side effects and safety. Speculation executes a hypothesized next action $\hat{a}_{t+1}$ that may be wrong, so safety requires the ability to simulate first and then commit or roll back. In domains like chess, rollback is trivial; in others, overwrite is easy (e.g., OS tuning). But many systems involve irreversible or externally visible effects (e.g., deleting records, placing orders), where naive speculation is harmful. Thus, speculation needs to be limited to cases where mispredictions are reversible, via forking, snapshot restoration, or roll-forward repair (e.g., refund/replace).

3 ENVIRONMENTS

We now instantiate speculative actions in three environments—chess, e-commerce dialogue, and multi-hop web search—chosen to stress distinct latency bottlenecks (reasoning, tool/API round trips, and information retrieval). In each case we pair a fast Speculator with a slow Actor and implement Algorithm 1. We report prediction accuracy and end-to-end time saved.

3.1 CHESS ENVIRONMENT

We demonstrate the effectiveness of our framework in the context of competitive multi-agent gameplay, focusing on chess as a canonical turn-based example. In standard play, analysis is strictly sequential: each player begins analysis only *after* the opponent has completed their turn. This serialization introduces substantial idle time. Particularly when both players rely on computationally intensive reasoning models, a single game can stretch to hours of wall-clock time. Our framework relaxes this constraint through speculative parallel analysis, allowing players to anticipate and prepare for likely opponent moves in advance. We show that this results in significant reductions in overall game duration.

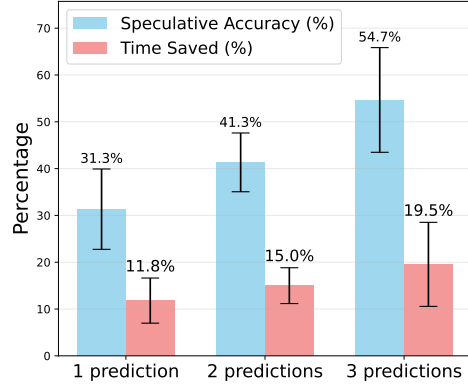


Figure 2: Percentage of time saved and percentage of correct predictions across 5 runs at 30 steps.

3.1.1 IMPLEMENTATION

We implement our framework on top of TextArena (Guertler et al., 2025), which provides a standardized game-play interface for LLM-driven agents.

Speculative Pipeline. At turn t , the game state s_t corresponds to the current board position. The in-turn player issues an API call h_t with parameter q_t constructed from s_t together with a reasoning-eliciting prompt. At this point, player P is to move, and player Q awaits. Our framework proceeds as follows:

- **Current in-turn player P :** the player receives s_t , makes an API call h_t to the agent with parameter $q_t = (s_t, \text{prompt})$. This API call returns the next move $a_t = h(q_t)$, typically with high latency due to deep and extensive reasoning.
- **Other out-of-turn player Q :**
 1. **Prediction phase.** The Speculator also receives the board state s_t and issues an API call $\hat{h}_t$, using a prompt optimized for speed rather than depth. It returns the top- k move predictions $\hat{a}_t^{(1)}, \hat{a}_t^{(2)}, \dots, \hat{a}_t^{(k)}$, ordered by confidence.
 2. **Parallel Computation.** For each predicted move $\hat{a}_t^{(i)}$, the out-of-turn player Q immediately launches a parallel process analyzing a next move $\hat{a}_{t+1}^{(i)} = h_{t+1}(\hat{s}_{t+1}, \text{prompt})$ for $i \in \{1, \dots, k\}$, where $\hat{s}_{t+1}^{(i)} = f(s_t, \hat{a}_t^{(i)})$ denotes the next state resulting from applying predicted action $\hat{a}_t^{(i)}$ to s_t .
 3. **Validation.** When the current in-turn player P finishes reasoning and returns its move a_t , we immediately check whether it matches any of the predicted moves $\hat{a}_t^{(1)}, \hat{a}_t^{(2)}, \dots, \hat{a}_t^{(k)}$.
 4. **Commit or Restart.** If a match exists, we commit to the corresponding speculative branch, advancing directly to $s_{t+1} = f(s_t, a_t)$. The game thus skips ahead, terminating other threads. If no match exists, we discard all speculative branches and continue with Q 's regular move computation $a_{t+1} = h_{t+1}(f(s_t, a_t), \text{prompt})$.

This pipeline is *lossless*: the final trajectory remains identical to non-speculative play, but time is saved through parallelized reasoning.

Agent Configuration We find that using the same model for both Speculator and Actor, but with different prompts, maximizes prediction accuracy while keeping speculation fast. Accordingly, in our experiments, the Actor is instantiated as GPT-5 with high reasoning effort, with each move is produced via an API call. The Speculator also employs GPT-5, but configured with low reasoning effort and a specialized system prompt designed for rapid move prediction rather than exhaustive analysis.

3.1.2 RESULTS

We evaluate our framework in terms of both time saved and prediction accuracy.

We track two metrics: (i) prediction accuracy, defined as the fraction of rounds in which any speculative prediction matches the actual move, and (ii) time saved, computed as $(T_{seq} - T_s)/T_{seq}$, where T_s and T_{seq} denote speculative and sequential execution times, respectively.

Time Saved and Accuracy increases with more predictions. Figure 2 reports results over 30 steps. Our framework consistently reduces execution time compared to sequential play, with larger savings as the number of speculative predictions increases. Across 5 runs, using 3 predictions yields an average time saving of 19.5% with an average prediction accuracy of 54.7%.

Randomness of agent call in game-play Variance in Figure 2 reflects realistic game-play dynamics from actual game runs with live API calls. Even with correct predictions, time savings vary: when the resulting position admits an obvious response, little acceleration occurs, as computation would be fast regardless. Significant gains arise only when predictions lead to positions requiring deep analysis. Thus, the effectiveness of speculation depends jointly on prediction accuracy and the complexity of the resulting positions.

3.2 E-COMMERCE ENVIRONMENT

Beyond competitive game-play, customer-agent interactions in the e-commerce provide a real-world setting where speculative actions can substantially improve performance. In a typical workflow, the customer submits a query through a chat interface and waits for the agent to respond. When the agent needs to invoke multiple API calls sequentially—for example, processing a return may involve retrieving order information, validating eligibility for each item, and initiating the return—the resulting latency can significantly degrade user experience. By contrast, if some API calls are correctly speculated and executed in advance, the agent can return results immediately once the query arrives, greatly reducing latency and making the interaction feel seamless. We evaluate this setting using the retail environment from *τ -bench* Yao et al. (2024).

3.2.1 EXPERIMENTAL SETUP

Speculative Pipeline In this scenario, the current state s_t is defined as the conversation history up to turn t , and h_t is the API calls required to answer the user’s query (eg. `get_user_details`, `get_order_details`). Our Speculator will predict

1. The user’s query $\hat{a}_t$;
2. The target API calls and their corresponding parameters $(\hat{h}_{t+1}^{(i)}, \hat{q}_{t+1}^{(i)})$ for $i \in \{1, \dots, k\}$, conditioned on the current state s_t and the predicted user’s query from step 1. Since the number of API calls for each turn is not fixed, the Speculator must also predict k .

Agent configuration We test various models as Speculator: OpenAI GPT models (gpt-5-nano, gpt-5-mini, gpt-5) and Google Gemini models (gemini-2.5-flash) with different reasoning budgets (1024, 2048, 4096 tokens). Prior work Jiang et al. (2023); Chen et al. (2025) shows that heterogeneous LLM ensembles often outperform individual models, and multiple models can execute in parallel under the same time budget. Motivated by these findings, we evaluate two Speculator configurations: (i) a *single-model* setting, where speculation relies on one model, and (ii) a *multi-model* setting, where models with comparable capacity and reasoning budgets run in parallel (i.e., gpt-5-nano with low-budget Gemini, gpt-5-mini with medium-budget Gemini, and gpt-5 with high-budget Gemini). Their predictions are then aggregated into a shared candidate set of speculative actions. At runtime, when the user simulator provides the actual utterance, the Actor compares the speculative API calls with the ground-truth APIs. Correct predictions are committed immediately, eliminating latency, while incorrect predictions are safely discarded without affecting correctness.

Evaluation We evaluate performance using **APIs prediction accuracy**, defined as the fraction of speculative API calls that match the ground-truth APIs required to resolve the user’s query. This metric directly reflects the proportion of turns in which the user receives an immediate response,

without waiting for API execution. In other words, higher prediction accuracy directly translates into greater time savings.

3.2.2 RESULTS

Figure 3 shows that between 22% and 38% of API calls are correctly predicted by the Speculator. Accuracy improves with stronger models and the multi-agent configuration consistently outperforms single-model speculation. Importantly, low-budget models speculate in only 2–3 seconds (per the LLM API providers leaderboard¹), well below the average user typing time of about 30 seconds (assuming 40 words per minute). This means that in roughly one-third of turns, the agent can respond faster than sequential execution, without waiting for API execution. These results demonstrate that speculation can transform user experience from perceptibly delayed to effectively real-time in tool-heavy environments.

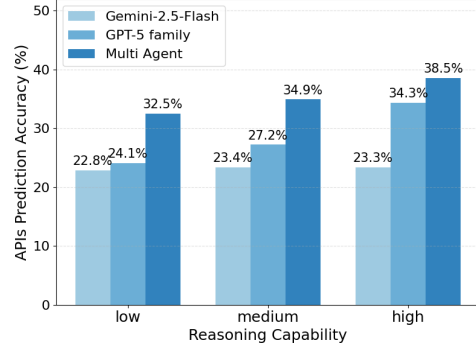


Figure 3: APIs prediction accuracy across different Speculator models with various reasoning capability.

3.3 HOTPOTQA ENVIRONMENT

We further evaluate our framework on HotpotQA, a setting where the main performance bottleneck arises from information retrieval latency. In this example, the agent must answer multi-hop questions through sequential Wikipedia API calls Yang et al. (2018). This tool-calling pattern mirrors real-world agentic workflows with high network latency per round-trip. We apply the framework from Section 2, where the Speculator predicts likely Wikipedia content while the actual API call executes. This parallelism enables the agent to continue reasoning on provisional information rather than blocking on API latency.

3.3.1 EXPERIMENTAL SETUP

We build our framework upon ReAct (Yao et al. (2023)), which interleaves chain-of-thought with tool use.

Speculative Pipeline In this scenario, the state s_t consists of the entire history of reasoning traces and retrieved information (API responses). At each step, the Actor takes in the current state s_t , selects an API call $h_t \in \{\text{Search}(), \text{Lookup}(), \text{Finish}()\}$ and a corresponding parameter q_t , e.g. $\text{Search}(\text{entity})$. The call $h_t(q_t)$ returns a response a_t , typically providing information about the queried entity. Our speculative framework operates as follows:

1. Speculator predicts API call response $\hat{a}_t^{(i)}$, yielding predicted next states $\hat{s}_{t+1}^{(i)} = f(s_t, \hat{a}_t^{(i)})$, $i \in \{1, \dots, k\}$.
2. Based on the states, the Actor generates reasoning traces and subsequently determines the next API decision $(\hat{h}_{t+1}^{(i)}, \hat{q}_{t+1}^{(i)})$ for $i \in \{1, \dots, k\}$.

Evaluation We evaluate the effectiveness of the speculative pipeline by the accuracy of the predicted API call decisions $(\hat{h}_{t+1}, \hat{q}_{t+1})$. Specifically, we compare the predicted call against the ground-truth call (h_{t+1}, q_{t+1}) obtained under the true response a_t . We employ a strict match criterion, counting a prediction as correct only when $\hat{h}_{t+1} = h_{t+1}$ and $\hat{q}_{t+1} = q_{t+1}$. This stringent criterion captures whether speculation enables meaningful progress, as even minor parameter differences (synonyms, word order) count as mismatches.

Agent configuration We evaluate speculative accuracy across three Speculator models: GPT-5-nano, GPT-4.1-nano and Gemini-2.5-flash. For each model, we measure the top-k prediction accuracy, with $k \in \{1, 3\}$.

3.3.2 RESULTS

¹<https://artificialanalysis.ai/leaderboards/providers>

Figure 4 shows that our Speculator successfully predicts the ground truth API call up to 46% of the time with top-3 prediction, despite our strict matching criterion. This accuracy improves significantly from top-1 to top-3 predictions, demonstrating that modest increases in speculation width yield substantial accuracy gains. Our speculation provides value by precomputing reasoning paths during otherwise idle API waiting time.

Model Patterns We observe high variation in API decision across different Speculators. These are largely driven by phrasing discrepancies – some models phrase the calls concisely while some over-specify. Interestingly, stronger models often yield lower accuracy, as their more diverse and context-specific queries (e.g., “List of Nobel laureates in physics 1970s” vs. “1970s Nobel Prize Physics winners list”) are penalized under strict matching. In contrast, weaker models tend to produce simpler, more predictable outputs.

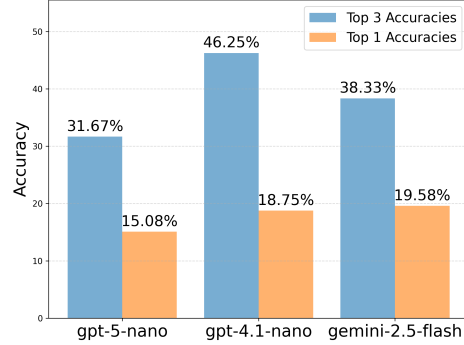


Figure 4: Accuracy with gemini-2.5-flash as the Actor. Speculating multiple actions ($k=3$) yields higher accuracy than predicting a single action.

4 BEYOND LOSSLESS SPECULATION: OS HYPERPARAMETER TUNING ENVIRONMENT

Thus far, our experiments have focused on *lossless* speculation, where speculative actions are validated sequentially before commitment. We now turn to a *lossy* setting that relaxes this step-by-step constraint. In latency-sensitive environments like an operating system, waiting for a powerful but slow Actor (10-15s deliberation) can leave the system in a degraded state. Our framework uses a fast Speculator to apply immediate adjustments, improving real-time performance, while the Actor deliberates. This is made safe by a last-write-wins mechanism—the Actor’s final decision simply overwrites any speculative action, removing the need for complex rollbacks. This method accelerates convergence and improves reaction time, which we evaluate on the `sysbench cpu` benchmark, a CPU-bound workload (Kopytov, 2020).

4.1 EXPERIMENTAL SETUP

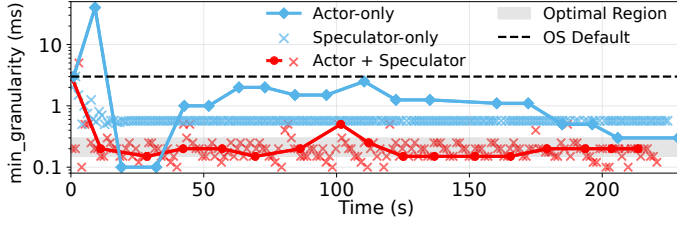
We tune a single parameter of Linux’s Completely Fair Scheduler (CFS), `min_granularity`, which controls a task’s minimum timeslice. This knob strongly affects scheduling performance: smaller timeslices reduce latency but can degrade throughput, yielding a classic trade-off. Prior work (Liargkovas et al., 2025) demonstrated that an LLM-based agent can optimize this parameter; here, we extend that setup with a speculative control loop.

Our system consists of a fast Speculator and a slow Actor. The Speculator proposes a bounded parameter update each second using the latest performance metric. The Actor, in contrast, responds every 10–15 seconds after analyzing a compressed chronology of the Speculator’s recent (measurement, action) pairs. When the Actor’s response arrives, its chosen setting is immediately applied and its state updates the Speculator’s context, ensuring subsequent fast steps proceed from a validated narrative rather than drifting.

Evaluation We evaluate three systems:

1. **Actor-only**: slow but deliberative (10–15 s interval);
2. **Speculator-only**: fast (1 s interval) but non-extensive;
3. **Speculator–Actor**: combined system using speculative updates between Actor decisions.

We are interested in (i) reaction time, (ii) convergence speed to the optimal setting, and (iii) ability to avoid local minima that trap the Speculator-only baseline.



Configuration	Latency p95 (ms)
Untuned	102.97
Actor-only	54.00
Actor + Spec.	37.93

Figure 5: **(Left)** Comparison of **Speculator-Actor**, **Speculator-only**, and **Actor-only** convergence. The Speculator shortens time spent exploring poor settings. The Speculator-only agent stabilizes quickly but at a worse final value. **(Right)** Average p95 latency over a 20-second tuning experiment showing that rapid reaction offers immediate performance benefits (see §B.3.4). Lower is better.

4.2 RESULTS

Speculator mitigates poor-reaction slowdowns The Speculator dramatically improves reaction time, as observed Figure 5 (Right). During the recovery period, the full Speculator-Actor system maintains an average p95 latency of 37.93 ms. This is a substantial improvement over the Actor-only baseline, which was forced to endure the poor performance for much longer and averaged 54.00 ms. The Speculator’s rapid correction prevents the system from lingering in a high-latency state (initially 102.97 ms), providing immediate benefits while the slower Actor deliberates (the full experiment is detailed in §B.3.4).

Speculator accelerates convergence to optimum The Speculator’s high-frequency updates also significantly accelerate convergence. As shown in Figure 5 (Left), the Speculator-Actor system reaches an optimal setting (e.g., 0.2 ms min_granularity) in approximately 10–15 seconds. In contrast, the Actor-only agent takes around 200 seconds—20x slower—and remains trapped in highly suboptimal states (e.g., latency >120 ms) for a long period of time. The Speculator’s rapid exploration provides the Actor with a richer performance landscape, allowing it to steer the system away from these pathological regions more quickly.

Speculator-only is good, but not optimal Figure 5 also shows that while the Speculator-only system reacts quickly, it converges prematurely to a suboptimal configuration: min_granularity of 0.55 ms (36.24 ms latency). This is significantly worse than the 0.2 ms (30.26 ms latency) achieved by the full Speculator-Actor system. Without the Actor’s guidance, the Speculator lacks the reasoning depth to escape this local minimum.

The joint Speculator–Actor system combines fast adaptation with strategic guidance, achieving both responsiveness and optimal steady-state performance.

5 CONCLUSION

This paper introduced **Speculative Actions**, a lossless framework for accelerating general agentic environments by breaking the strict sequentiality of their interaction loops. Unlike speculative decoding in LLM inference, our approach generalizes speculation to the full spectrum of agent–environment interactions—treating every step, whether an LLM call, tool invocation, MCP request, or human response, as an API call subject to prediction and parallelization. By pairing a fast *Speculator* with a slower but authoritative *Actor*, the framework enables agents to anticipate and prepare likely next actions in parallel, transforming otherwise idle waiting time into productive computation. We provided a formal analysis of its expected speedup, showing up to 50% theoretical latency reduction under simple assumptions, and instantiated the framework across four representative environments. Empirically, we observed substantial reductions in wall-clock time and latency, verifying that speculative actions consistently improve efficiency without compromising correctness. Notably, while most environments adopt a strictly lossless validation process, our systems-level experiment demonstrates that relaxing this constraint enables fast exploratory updates, extending speculation to adaptive control scenarios.

More broadly, speculative actions reveal a systems-level design principle for modern agentic platforms: *opportunistic parallelism in environment interactions*. Future work could extend this idea through multi-step and uncertainty-aware speculation, hierarchical actor–speculator architectures, or tighter integration with reinforcement learning and environment simulators. Together, these directions point toward general-purpose, real-time agentic systems where speculation becomes a first-class mechanism for efficient decision making.

REFERENCES

- Reyna Abhyankar, Qi Qi, and Yiying Zhang. Osworld-human: Benchmarking the efficiency of computer-use agents. *arXiv preprint arXiv:2506.16042*, 2025.
- Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.
- Anthropic. Introducing the model context protocol. <https://www.anthropic.com/news/model-context-protocol>, November 2024. Accessed: 2025-09-24.
- Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023. URL <https://arxiv.org/abs/2302.01318>.
- Zhijun Chen, Jingzheng Li, Pengpeng Chen, Zhuoran Li, Kai Sun, Yuankai Luo, Qianren Mao, Ming Li, Likang Xiao, Dingqi Yang, Yikun Ban, Hailong Sun, and Philip S. Yu. Harnessing multiple large language models: A survey on llm ensemble, 2025. URL <https://arxiv.org/abs/2502.18036>.
- Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 1–14, July 2019. URL <https://www.flux.utah.edu/paper/duplyakin-atc19>.
- Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. A survey on thread-level speculation techniques. *ACM Comput. Surv.*, 49(2), June 2016. ISSN 0360-0300. doi: 10.1145/2938369. URL <https://doi.org/10.1145/2938369>.
- Vivek Farias, Joren Gijsbrechts, Aryan Khojandi, Tianyi Peng, and Andrew Zheng. Speeding up policy simulation in supply chain rl. *arXiv preprint arXiv:2406.01939*, 2024.
- Yichao Fu, Rui Ge, Zelei Shao, Zhijie Deng, and Hao Zhang. Scaling speculative decoding with lookahead reasoning. *arXiv preprint arXiv:2506.19830*, 2025.
- Yilin Guan, Wenyue Hua, Qingfeng Lan, Sun Fei, Dujian Ding, Devang Acharya, Chi Wang, and William Yang Wang. Dynamic speculative agent planning, 2025. URL <https://arxiv.org/abs/2509.01920>.
- Leon Guertler, Bobby Cheng, Simon Yu, Bo Liu, Leshem Choshen, and Cheston Tan. Textarena, 2025. URL <https://arxiv.org/abs/2504.11442>.
- 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, 2024. URL <https://arxiv.org/abs/2410.00079>.
- Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. LLM-blender: Ensembling large language models with pairwise ranking and generative fusion. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14165–14178, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.792. URL <https://aclanthology.org/2023.acl-long.792/>.

-
- Tengjun Jin, Yuxuan Zhu, and Daniel Kang. Elt-bench: An end-to-end benchmark for evaluating ai agents on elt pipelines. *arXiv preprint arXiv:2504.04808*, 2025.
- Kaggle. Game arena. <https://www.kaggle.com/game-arena>, 2025. Accessed: 2025-09-21.
- Alexey Kopytov. Sysbench: Scriptable benchmark tool. <https://github.com/akopytov/sysbench>, 2020. Accessed: 2025-09-22.
- Monica S Lam and Robert P Wilson. Limits of control flow on parallelism. *ACM SIGARCH Computer Architecture News*, 20(2):46–57, 1992.
- Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.
- Georgios Liargkovas, Konstantinos Kallas, Michael Greenberg, and Nikos Vasilakis. Executing shell scripts in the wrong order, correctly. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pp. 103–109, 2023.
- Georgios Liargkovas, Vahab Jabrayilov, Hubertus Franke, and Kostis Kaffes. An expert in residence: Llm agents for always-on operating system tuning. In *Proceedings of the NeurIPS 2025 Workshop on Machine Learning for Systems (MLForSys)*, San Diego, CA, USA, December 2025. NeurIPS. Accepted paper.
- Andrea Mambretti, Matthias Neugschwandner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: a tool to analyze speculative execution attacks and mitigations. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 747–761, 2019.
- Edmund B Nightingale, Daniel Peek, Peter M Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. *ACM SIGARCH Computer Architecture News*, 36(1):308–318, 2008.
- OpenAI. Introducing deep research. <https://openai.com/index/introducing-deep-research/>, 2025. Accessed: 2025-09-24.
- Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: improving configuration management with operating system causality analysis. *ACM SIGOPS Operating Systems Review*, 41(6):237–250, 2007.
- Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.
- Jikai Wang, Juntao Li, Jianye Hou, Bowen Yan, Lijun Wu, and Min Zhang. Efficient reasoning for llms through speculative chain-of-thought, 2025a. URL <https://arxiv.org/abs/2504.19095>.
- 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, 2025b. URL <https://arxiv.org/abs/2505.02865>.
- Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering, 2018. URL <https://arxiv.org/abs/1809.09600>.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL <https://arxiv.org/abs/2210.03629>.
- Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ -bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains, June 2024. URL <http://arxiv.org/abs/2406.12045>. arXiv:2406.12045.
- Chen Zhang, Zhuorui Liu, and Dawei Song. Beyond the speculative game: A survey of speculative execution in large language models. *arXiv preprint arXiv:2404.14897*, 2024.

A PROOF OF PROPOSITION 1

Proof. Baseline. In sequential execution, each of the T steps requires one call to the true model h with mean latency $1/\beta$. Therefore

$$E[T_{\text{seq}}] = \frac{T}{\beta}.$$

Expected time saved per hit. Consider two consecutive steps $(t, t+1)$. In the baseline, the total completion time is $R = B + C$, where $B, C \sim \text{Exp}(\beta)$ are the latencies of step t and step $t+1$. With speculation, we launch $A \sim \text{Exp}(\alpha)$ during step t . If the guess is correct, the $(t+1)$ call C can be issued once either A or B finishes, so the block completes at

$$S = C + \min\{A, B\}.$$

Thus, when a guess is correct, our expected time saved is

$$R - S = (B - A)_+,$$

where $(x)_+ = \max\{x, 0\}$.

By independence of A, B ,

$$\mathbb{E}[(B - A)_+] = \int_0^\infty \int_0^b (b - a) \alpha e^{-\alpha a} \beta e^{-\beta b} da db = \frac{\alpha}{\beta(\alpha + \beta)}.$$

Expected number of hits. We denote the expected number of hits by round n as S_n , that is

$$\mathbb{E}[\text{number of hits by round } n] = S_n$$

We then have $S_0 = 0, S_1 = p$. In round 1, either (i) we hit with probability p , in which case round 2 cannot be a hit (there is no speculation window immediately after a correct speculation), contributing $1 + S_{n-2}$; or (ii) we miss with probability $1 - p$, after which round 2 proceeds normally, contributing S_{n-1} . We then have the following recursion

$$S_n = p(1 + S_{n-2}) + (1 - p)S_{n-1}$$

Solve this linear recurrence by splitting into homogeneous and particular parts.

(1) *Homogeneous part*

$$S_n^h = pS_{n-2} + (1 - p)S_{n-1}$$

The characteristic equation is

$$r^2 - (1 - p)r - p = 0 = (r - 1)(r + p) \implies \text{the roots are } r_1 = 1, r_2 = -p$$

Therefore,

$$S_n^{(h)} = C_1 + C_2(-p)^n.$$

(2) *Particular solution* The forcing term is constant ($+p$), and $r = 1$ is a root, so a constant trial collides with the homogeneous part. Try $S_n^{(p)} = an$ and substitute:

$$an = (1 - p)a(n - 1) + pa(n - 2) + p = an - a(1 + p) + p,$$

which gives $a(1 + p) = p$ and thus

$$S_n^{(p)} = \frac{p}{1 + p} n.$$

Combine:

$$S_n = C_1 + C_2(-p)^n + \frac{p}{1 + p} n.$$

Use $S_0 = 0$ and $S_1 = p$:

$$0 = C_1 + C_2, \quad p = C_1 + C_2(-p) + \frac{p}{1 + p}.$$

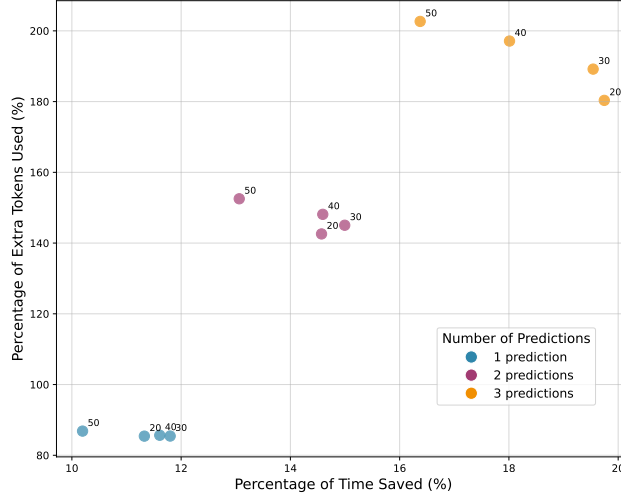


Figure 6: Percentage of extra tokens used against percentage of time saved for different numbers of predictions at different numbers of steps, averaged across 5 runs.

Solving yields $C_2 = -\frac{p^2}{(1+p)^2}$ and $C_1 = \frac{p^2}{(1+p)^2}$.

Hence the closed form solution is

$$S_n = \frac{p}{1+p} n + \frac{p^2}{(1+p)^2} (1 - (-p)^n)$$

Total saving. There are $T - 1$ potential speculation windows, hence

$$E[T_s] = \frac{T}{\beta} - S_{T-1} \frac{\alpha}{\beta(\alpha + \beta)}$$

Final ratio. Dividing by $E[T_{\text{seq}}] = T/\beta$ gives

$$\frac{E[T_s]}{E[T_{\text{seq}}]} = 1 - \frac{1}{T} \frac{\alpha}{\alpha + \beta} \left[\frac{(T-1)p}{1+p} + \frac{p^2}{(1+p)^2} - \frac{p^2}{(1+p)^2} (-p)^{T-1} \right]$$

□

Taking $T \rightarrow \infty$, we get exactly that the ratio converges to $1 - \frac{p}{1+p} \frac{\alpha}{\alpha + \beta}$

B ADDITIONAL ENVIRONMENT DETAILS

B.1 CHESS

Trade-off between Time Saved and Token Cost. In addition to time savings, we also measure the additional token cost of parallel speculation. We track this using the metric **extra token percentage**: $(M_{\text{sequential}} - M_{\text{speculative}})/M_{\text{sequential}}$. Figure 6 shows the trade-off between time saved and token cost for each number of predictions. We observe that the extra token percentage increases as the number of predictions increases, while the time saved percentage also increases, creating a clear trade-off.

B.2 ECOMMERCE

τ -bench: A benchmark designed for dynamic task-oriented dialogues between a user (simulated by language models) and an API-augmented agent. The benchmark spans two domains — retail and

airline, with structured databases, domain-specific tools. We focus on the retail domain, where the agent assists users with operations such as canceling or modifying pending orders, initiating returns or exchanges, or providing product and order information. The benchmark defines 115 tasks with 15 APIs (7 write, 8 read-only).

Trade-off between Prediction Accuracy and Cost. The time cost in Figure 7a consists of latency (Time to First Token) and output response time. The dashed vertical line represents the average user typing time, estimated at 40 words per minute. At this threshold, the multi-agent setting achieves approximately 34% prediction accuracy, meaning that in over one-third of cases the agent can return an immediate response without waiting for API execution. This demonstrates that speculation can transform user experience from perceptibly laggy to effectively real-time in tool-heavy environments.

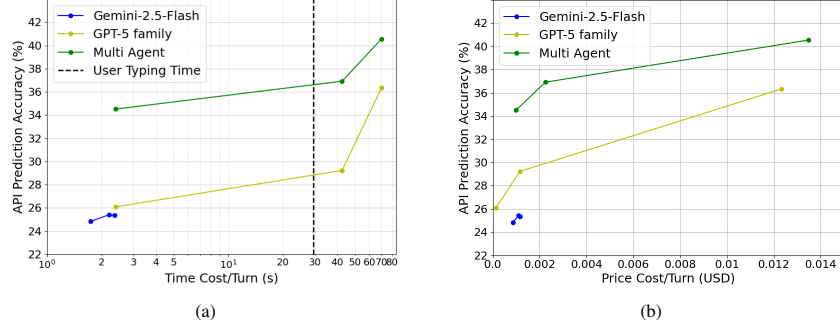


Figure 7: **Prediction Accuracy against Speculator’s Cost across different models.** (a) Accuracy–Speculator time cost trade-off across models. The dashed line shows average user typing time. (d) Accuracy–Speculator price trade-off across models, reflecting the monetary cost of speculative execution.

B.3 OPERATING SYSTEM TUNING

B.3.1 EXPERIMENTAL SETUP AND IMPLEMENTATION DETAILS

System and Workload Configuration All experiments were conducted on a dedicated machine with 2× Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz, 192 GB DDR4 RAM, and a 1 TB NVMe SSD running Ubuntu 22.04 with Linux Kernel 5.15, hosted on Cloudlab (Duplyakin et al., 2019).

We run `sysbench cpu` (Kopytov, 2020), a CPU-bound benchmark that repeatedly calculates a large prime number sequence. The benchmark reports several performance metrics every second. We run `sysbench` on 16 concurrent threads pinned on two CPU cores.

Tuner Implementation Details The system consists of two agents, a fast Speculator and a slow Actor, which collaborate to minimize the p95 latency of the workload. At each step, the tuner proposes a new configuration, which is applied to the live system. Applying the proposed parameters is a near-instant operation.

CFS Parameter Details The Completely Fair Scheduler (CFS) is a CPU scheduler for Linux that aims to give every task a fair share of CPU time. It exposes various hyperparameters that allow administrators to adjust its behavior. We tuned `min_granularity_ns`, which enforces a minimum timeslice a task will receive. The prompt templates guided the agents to explore a range from 50,000 to 50,000,000 nanoseconds (0.05 ms to 50 ms). The default value on Kernel 5.15 is 3 ms. Lower values for this parameter are expected to increase responsiveness at the cost of higher context-switching overhead, while higher values improve throughput but can worsen latency.

History Compression History compression is managed via distinct prompt structures for the two agents. When the slower Actor is invoked, its prompt context contains a fully compressed summary of all actions taken during its deliberation window. Each action from the Speculator is listed as a concise (parameter, result) pair. In contrast, the faster Speculator receives a hybrid context: it sees

the same compressed history from the last Actor cycle, supplemented by the full, verbose replies from its own most recent actions. This dual-context mechanism allows the Actor to analyze long-term trends from a compact summary, while the Speculator retains immediate, detailed context for its rapid, reactive decisions.

B.3.2 PROMPT ENGINEERING FOR MULTI-AGENT OPTIMIZATION

The following are the prompt templates used to guide the two LLM agents.

Initial System Prompt for Actor and Speculator

You are a Linux kernel scheduler tuning expert with deep knowledge of the Completely Fair Scheduler (CFS).

MULTI-AGENT ROLE: You are part of a MULTI-AGENT System.

[For Actor] You are the Actor. Your role is to provide thoughtful, well-analyzed parameter recommendations. You work alongside a Speculator that explores the parameter space rapidly. You will receive accumulated results from multiple agent calls to perform deeper analysis and identify trends.

[For Speculator] You are the Speculator. Your role is to provide immediate, intuitive parameter recommendations for each window. You work alongside an Actor that performs deeper analysis.

Your goal is to MINIMIZE p95 latency for a CPU-bound workload. The workload performance metrics might be NOISY, so look for consistent trends across configurations.

Tunable CFS parameter:

- **min_granularity_ns:** Minimum time slice before preemption. Lower values increase responsiveness but also overhead. Higher values improve throughput but can worsen latency.

Parameter Range:

- **min_granularity_ns:** 50,000 to 50,000,000 nanoseconds

Performance data will be provided in future calls. Respond **ONLY** in the format shown below:

Analysis: <Your one or two-sentence decision reasoning>

Config: { "min_granularity_ns": <int> }

Update for Speculator

[Context includes the compressed history for calls 1-10 and the raw Speculator responses for iterations 11-18]

CURRENT BEST: p95 latency=[value] at call #[value]

Latest Result for call #19:

Config: "min_granularity_ns": [value] → p95 latency=[value]

Please provide your analysis and the next configuration for iteration #20.

Update for Actor

[Context includes the compressed history for calls 1-10]

CURRENT BEST: p95 latency=[value] at call #[value]

RESULT for call #11 [SPECULATOR]: min_granularity_ns=[value] → p95 latency=[value]

RESULT for call #12 [SPECULATOR]: min_granularity_ns=[value] → p95 latency=[value]

...

RESULT for call #19 [SPECULATOR]: min_granularity_ns=[value] → p95 latency=[value]

Please provide your analysis of the trend and the next configuration for call #20.

Sample Agent Response

Analysis: The performance peaked at 300,000 ns, suggesting the optimal value is likely in that region.
I will narrow the search around that peak.
Config: { "min_granularity_ns": 250000 }

B.3.3 TOKEN USAGE AND COSTS

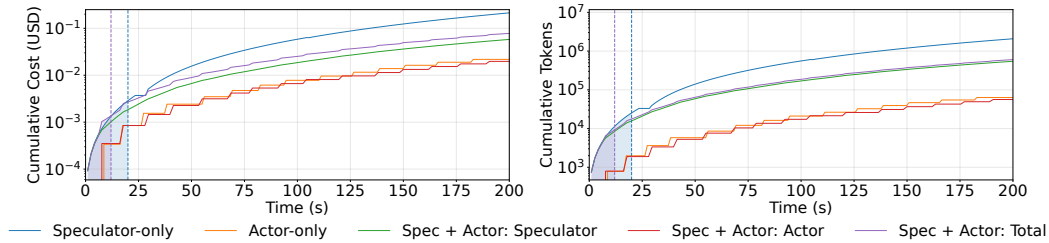


Figure 8: **Cumulative token usage and cost over time.** The left and right plots show the cumulative cost (USD) and total tokens used, respectively, for all three configurations. The vertical lines mark the observed convergence point for each system.

Table 2: Cumulative tokens and cost (in cents) at selected time marks.

Elapsed Time	Actor-only		Speculator-only		Actor+Speculator (Total)	
	Tokens	Cost (cents)	Tokens	Cost (cents)	Tokens	Cost (cents)
Base	744	0.02	690	0.01	690	0.03
10s	790	0.03	9,539	0.11	8,548	0.13
30s	3,631	0.15	45,768	0.57	32,459	0.48
60s	8,581	0.35	205,794	2.24	84,568	1.18
120s	26,398	0.96	778,253	8.12	261,855	3.53
200s	63,376	2.18	2,099,894	21.5	607,877	7.83

As illustrated in Figure 8 and detailed in Table 2, the high frequency of the Speculator leads to rapid growth in token consumption and cost. In practice, however, this growth is bounded by the system’s fast convergence. The combined Actor-Speculator system converges in approximately 15 seconds, while the Speculator-only system converges in 20 seconds. The Actor-only system converges after 200 seconds. Once an optimal state is reached, the tuning process concludes, rendering the potential for long-term exponential cost negligible in this context. Several optimization strategies, like truncating the context to a fixed window or disabling exploration after convergence, could further mitigate token growth but are left for future work.

B.3.4 SPECULATIVE REACTION TIME BENEFITS

To provide a targeted example of how speculation mitigates transient performance loss, we conducted a controlled experiment. In this scenario, the system is deliberately perturbed at time t_0 by setting the `min_granularity` parameter to a highly suboptimal value (10 ms). We then compare the system’s recovery under two configurations: the Actor-Speculator system and an Actor-Only baseline, which replays only the actions proposed by the Actor from the full Actor-Speculator trace.

As shown in Figure 9, the Actor-Speculator system reacts almost instantly. The fast Speculator, seeing the immediate performance degradation, applies a corrective action that brings the system back to an efficient state in about one second. In contrast, the Actor-Only system is forced to endure

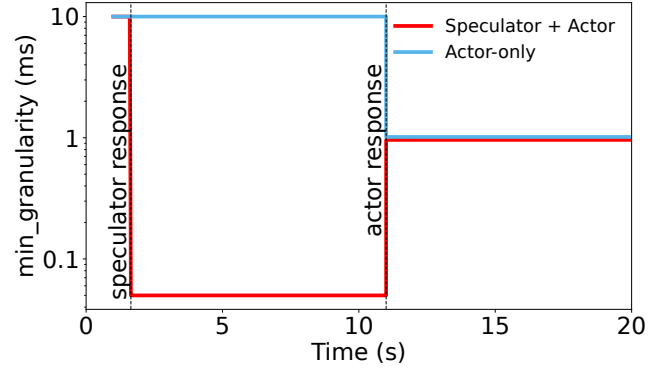


Figure 9: A controlled experiment showing the system’s step response after a manual perturbation at $t = 0$. The **Actor-Speculator** system corrects the poor setting within a second, while the **Actor-only** system must wait over 10 seconds for its next decision cycle. The quantitative results of this experiment are summarized in Figure 5 (Right) in the main text.

the poor performance for over 10 seconds, as it must wait for the slower Actor to complete its deliberation cycle before it can act. The performance gap shown in the plot is quantified in the main text (Figure 5, Right).