Multi-Step Speculative Actions: From Single Guesses to Speculative Tree in LLM Agents

Anonymous ACL submission

Abstract

LLM-based agents are bottlenecked by the strictly sequential nature of their interaction loop: each step must wait for the previous LLM call to resolve before the next can begin. Speculative actions extend the speculative-decoding paradigm to the action level by overlapping a slow *Actor* with a fast *Speculator*, yielding lossless speedups. Existing approaches either expand speculation *horizontally* (multiple candidate actions in parallel at one step) or *vertically* (a single deep chain of pre-launched calls), but no prior work jointly exploits both dimensions. We propose **Multi-Step Speculative Actions (MSSA)**, a unified, event-driven algorithm that builds a tree of pre-launched Actor and Speculator calls during the time the root Actor is in flight, allowing multiple confirmed steps to cascade back-to-back without any cold restart. On a chess agent benchmark with three Actor–Speculator configurations, MSSA achieves up to 40.0% end-to-end latency reduction over sequential execution, exceeding the strongest breadth-only baseline by up to 20.6 percentage points. Our analysis shows that cascaded hits—consecutive confirmations enabled by depth speculation—become the dominant source of speedup as the speculation breadth grows, confirming that breadth and depth interact constructively. Code is available at annonymous link - `https://github.com/flamingo-anonymous/multi-step-spec-action`

1 Introduction

The latency of a single LLM forward pass has become a first-class bottleneck for large-scale deployment. Speculative decoding (Leviathan et al., 2023; Chen et al., 2023) addressed this at the *token* level by pairing a fast draft model with a slow verifier, delivering 2–3× wall-clock speedups without changing the output distribution. The same idea is increasingly central in tree-based extensions such as SpecInfer (Miao et al., 2024), Medusa (Cai et al., 2024), and EAGLE (Li et al., 2024).

A parallel research thread has scaled language models far beyond single-shot generation: LLM-based *agents* now orchestrate multi-step trajectories of tool calls, API invocations, and environment interactions to accomplish complex tasks (Yao et al., 2023; Shinn et al., 2023; Park et al., 2023; Liu et al., 2024). These agents are typically structured as a strictly sequential perceive–reason–act loop: every step blocks until the previous LLM call returns. State-of-the-art agentic systems require tens of minutes per episode for OS-level tasks (Xia et al., 2024a; Jimenez et al., 2024), deep research (Yoran et al., 2024), multi-step web search (Mialon et al., 2024), or complex games (Guertler et al., 2025; Ye et al., 2025). In real-time or safety-critical settings—autonomous driving (Mao et al., 2023; Sima et al., 2024), clinical decision support (Tu et al., 2025), and financial monitoring or trading (Yu et al., 2024)—even a few seconds of sequential overhead can be unacceptable, yet switching to a smaller model sacrifices reasoning quality. The central question is therefore: *how can we accelerate LLM agents without compromising the actions they take?*

Speculative actions (Ye et al., 2025) answer this by lifting speculative decoding from the token level to the action level. Considering an agentic setting, While a slow *Actor* computes the ground-truth action at the current step, a fast *Speculator* predicts likely next actions and pre-launches the corresponding Actor calls for the next step. If any pre-launched speculator call matches the Actor's true output, then the next step is already in flight when needed; if none matches, speculative work is discarded and execution proceeds normally, guaranteeing identical final trajectories to a non-speculative agent.

The gap. Two strategies dominate prior speculative-action work, and each uses only one way of spending speculative compute. **Breadth-focused** speculation (Ye et al., 2025) emits b candidate actions in parallel at the current step. When one of these guesses turns out to match the action the Actor actually takes—a *hit*—the Actor call for the *next* step has already been pre-launched, so that step is effectively free. However, breadth looks only one step ahead and pipeline must begin again from scratch (it *restarts cold*) at every step. This caps the achievable speedup at one saved step per hit. This will be more clear in Section 3 **Depth-focused** speculation (Ye et al., 2025; Guan et al., 2025) instead extends a *single* predicted branch many steps into the future, so that several consecutive steps can be confirmed back-to-back without waiting—a *cascade*—lifting the one-step ceiling. The drawback is that a deep chain is built from a sequence of guesses stacked on top of one another, and the whole chain is useful only if every guess along it is correct. A single wrong guess early in the chain invalidates everything below it, so the probability that the chain survives shrinks quickly as it grows deeper. Crucially, *no prior work jointly explores breadth and depth* or characterizes how to allocate a fixed speculative budget across the two.

Contributions. We close this gap with the following contributions.

1. We propose **Multi-Step Speculative Actions (MSSA)**, a single event-driven algorithm that simultaneously expands the speculative tree in breadth and depth, subject to a single budget parameter (*max_inflight_actor*).
2. We provide an empirical study on a chess agent across three Actor–Speculator pairs spanning GPT-OSS and Qwen-3, showing that MSSA consistently outperforms the breadth-only baseline—by up to $+20.6$ percentage points of time saving.
3. Through breadth and depth ablations, we identify cascaded hits as the primary driver of speedup beyond what breadth alone can deliver, and show that neither dimension yields unbounded gains: time saving plateaus as breadth grows past the point where the Speculator's coverage saturates, and as the Actor budget grows past the point where the tree can productively use additional depth. This characterizes a practical sweet spot in the breadth–depth tradeoff.

2 Related Work

Speculative decoding. Stern et al. (2018) first explored parallel decoding for autoregressive models. Speculative decoding in its modern form (Leviathan et al., 2023; Chen et al., 2023) pairs a small draft model with a large verifier to produce identical samples in fewer wall-clock steps. Subsequent work extends this with tree-structured verification (Miao et al., 2024), multi-head drafts (Cai et al., 2024), feature-level prediction (Li et al., 2024), and adaptive draft selection (Kim et al., 2023); see Xia et al. (2024b) for a survey. This direction continues to advance. Recent work refines the draft–verifier interface along complementary axes: Li et al. (2026) predict fused multi-layer features at training time to improve draft fidelity; Bachmann et al. (2025) relax strict token-level alignment so that objectively valid drafts are not needlessly rejected; and Timor et al. (2025) dispense with the shared-vocabulary assumption to permit lossless drafting across heterogeneous tokenizers. All of these operate at the token level and therefore reduce the cost of a *single* LLM call, not of a multi-step agent loop.

LLM agents. Agentic frameworks compose LLM calls with tool invocations and environment feedback (Yao et al., 2023; Schick et al., 2023; Shinn et al., 2023). Benchmarks such as τ-bench (Yao et al., 2025), AgentBench (Liu et al., 2024), WebShop (Yao et al., 2022), SWE-bench (Jimenez et al., 2024), and TextArena (Guertler et al., 2025) quantify capability but also expose long episode latencies as a major obstacle to deployment and reinforcement-learning training.

Speculative actions. Ye et al. (2025) introduced action-level speculative execution and studied a breadth variant that pre-launches b candidate actions per step. Guan et al. (2025) propose Dynamic Speculative Agent Planning, which uses asynchronous online reinforcement learning to adapt the depth of a single forward chain while trading off latency against cost. Several concurrent works also speed up the agent loop by overlapping the slow model with faster speculation, but each explores a different direction: Nichols et al. (2025) predict tool calls with a lightweight speculator verified by the main model; Sui et al. (2026) exploit recurring control-flow patterns to speculatively execute entire tool-call sequences; Hooper et al. (2026)

Symbol	Meaning
Environment & policy	
s_t	State at step t; s_0 is the initial state.
T	Horizon (total number of steps in an episode).
$\pi(s)$	Policy mapping state s to (h, q).
h_t, q_t	Header (which API/tool) and query (its arguments) at step t.
$f(s, a)$	Deterministic transition function; $s_{t+1} = f(s_t, a_t)$.
Roles	
Actor	Slow, authoritative LLM (or tool); returns a_t given (h_t, q_t).
a_t	Ground-truth action committed at step t.
Speculator $\hat{g}$	Fast predictor; returns b candidate actions from (s_t, h_t, q_t).
$\hat{a}^{(i)}$	The Speculator's i-th candidate action.
b	Speculation breadth (number of Speculator candidates per node).
$\textsc{Match}(\hat{a}, a)$	Predicate testing equivalence of $\hat{a}$ and a (exact string match in our case).
Speculative tree & budget	
$\mathcal{T}$	Speculative tree; root is the current confirmed state.
u	A node in $\mathcal{T}$ (a speculative state plus its pending Actor/Speculator calls).
$\bar{a}$	Pending (in-flight) action handle from an asynchronous call.
max_inflight_actor	Cap on concurrent Actor calls (controls effective tree depth).

Table 1: Notation used for MSSA

combine asynchronous I/O with speculative read-only tool calls held until confirmation; and Huang et al. (2025) co-design algorithm and system to speculate search-agent actions. All of these threads commit to either a single speculative chain or one layer of candidates, and none characterizes the joint breadth–depth allocation. MSSA generalizes breadth and depth into a single tree-based pipeline parameterized by $(b, \textit{max_inflight_actor})$, in the spirit of tree-structured verification (Miao et al., 2024) but at the action level, building on the classic principle of issuing future work eagerly under uncertainty.

3 Methodology

3.1 Setup

We consider an agent that interacts with a deterministic environment for T steps. At each step the agent must decide *which* external resource to invoke and *with what* input: a policy $\pi(s)$ maps the current state s to a pair (h, q), where the header h specifies the API or tool to call and the query q specifies its parameters or payload. Keeping h and q abstract makes the framework environment-agnostic. In our chess environment, each player's move is produced by an LLM call conditioned on the board: h_t encodes whose turn it is and q_t encodes the board state, so (h_t, q_t) together form the prompt header and query of the LLM request.

A slow **Actor** returns the ground-truth action $a_t \sim \text{Actor}(h_t, q_t)$, and a fast **Speculator** $\hat{g}$ returns b candidate actions $\{\hat{a}^{(1)}, \ldots, \hat{a}^{(b)}\}$ it believes the Actor is likely to take. The environment transitions deterministically via $s_{t+1} = f(s_t, a_t)$, and a domain-specific predicate $\textsc{Match}(\hat{a}, a)$ checks equivalence between a speculative action and the ground truth (exact string match, in our setting). Both calls are issued asynchronously and return pending handles that resolve later. Table 1 collects all notation used in this section and the next. In our agentic chess setting the Actor is itself an LLM, but the framework places no such restriction: in a fully agentic deployment the Actor can be any tool, API, or environment the agent has access to, as long as its output can be matched against the Speculator's guesses.

3.2 Pipeline Comparison

Figure 1 contrasts the three pipelines. In the *sequential* baseline, each Actor call begins only after the previous one resolves: in the chess setting, the LLM call for the next player's move cannot be issued until the previous player's LLM call has returned, so every move sits idle waiting for the one before it. In the *breadth-only* pipeline (Ye et al., 2025), the Speculator at s_t emits b candidate actions, each of which seeds one pre-launched Actor call from its hypothetical next state. Once the Speculator returns and these Actor calls are pre-launched, the system still waits for the *original* Actor at s_t to return its ground-truth action; the branch whose prediction matches a_t is then kept, and the rest are pruned or stopped. Because speculation extends only one step ahead, each new confirmed state restarts speculation from cold.

Our *multi-step* pipeline speculates *recursively*. We call any hypothetical, not-yet-confirmed future state a *speculative state*. Unlike breadth-only, a speculative state does not merely host a pre-launched Actor call: it also launches its *own* Speculator, whose b candidates create the next layer of speculative states, which in turn launch their own Speculators, and so on. While the root Actor is still

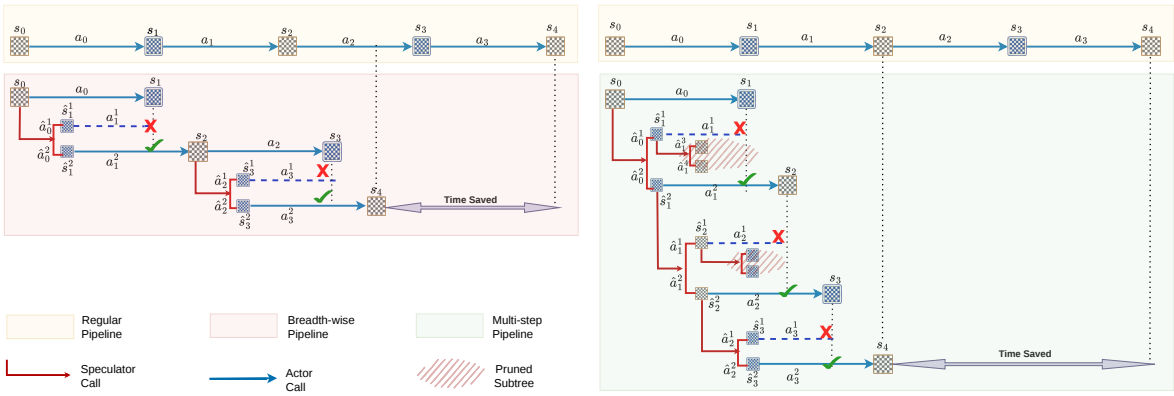

Figure 1: Three execution pipelines. **Sequential**: each Actor (blue) waits for the previous one. **Breadth-only**: a single layer of b speculative Actor calls is pre-launched; non-matching branches are pruned (hatched red). **Multi-step (ours)**: speculation builds a tree of pre-launched Actor/Speculator calls; matching descendants cascade-commit without network wait, yielding the time saved over the sequential baseline. Blue arrows are Actor calls, red arrows Speculator calls.

in flight, this process builds a tree of speculative states several levels deep, so that work for steps $t{+}1, t{+}2, \ldots$ is already in progress before step t is even confirmed. When the root Actor resolves, MSSA keeps the matching child and immediately checks whether that child's Actor has already resolved; if so, a cascade of consecutive confirmations proceeds with zero additional latency. We illustrate all three pipelines on a short chess rollout in Figure 1; a step-by-step walkthrough with the full notation is deferred to Appendix B.

3.3 Algorithm

Goal. The algorithm must roll out an episode of T actions but spend the time the slow Actor is blocked doing useful work for *future* steps. To do this it keeps a tree of speculative states. Each *node* of the tree holds one speculative state together with the two calls issued for it—an Actor call and a Speculator call. The *root* is always the current confirmed state. The tree forms because each Speculator call returns b guesses, and we turn *every* guess into a child node that itself launches a Speculator and an actor those children's guesses become grandchildren, and so on. The only thing that limits how far this grows is *max_inflight_actor*, a single integer bounding how many Actor calls may be in flight at once: Speculator calls are cheap and fire as soon as a node is created, but an Actor call beyond the budget is deferred until a slot frees. This budget is the primary knob in the depth–cost tradeoff.

Structure. Algorithm 1 is an event loop with two handlers. The **main loop** seeds the root and then repeatedly waits for *any* pending Actor or Speculator call anywhere in the tree to finish, dispatching it by type.

ONSPECULATORDONE$(u, \{\hat{a}^{(i)}\})$ *expands the tree one level under node* u. For each of the b guesses $\hat{a}^{(i)}$ it computes the hypothetical next state $s' = f(u.s, \hat{a}^{(i)})$ and its policy target $(h', q') = \pi(s')$, and "creates a child c" for it—a new node recording the guess, s', and (h', q'). If the Actor budget allows, it immediately pre-launches that child's own Actor and Speculator calls, which is what makes the speculation *recursive* and grows the tree deeper while the root Actor is still blocked.

ONACTORDONE(u, a^*) records the Actor result a^* on node u. If u is not the root, there is nothing to commit yet and it returns. If u *is* the root, the root's Actor has produced the ground-truth action, so the algorithm commits it ($s_{t+1} \leftarrow f(s_t, a^*)$, t advances) and looks for the *child of the root whose guessed action matches* a^* via MATCH. If no child matches (**Miss**), the speculative tree was wrong; it is discarded and a fresh root is started from the confirmed state—costing exactly the sequential baseline at that step. If a child matches (**Hit**), that child becomes the new root. The algorithm then enters the inner loop "*commit; advance root via match; tally Cascade*": as long as the new root's Actor call has *already resolved* (because it was pre-launched while the old root was blocked), the algorithm can commit that step too, advance the root again through its matching child—repeating until it reaches a root whose Actor is still pending or a mismatch ends the streak. Each cascaded step is a full Actor round-trip saved over both the

Algorithm 1 Multi-Step Speculative Actions (MSSA)

Require: s_0, T, f, π, $\hat{g}$; budget *max_inflight_actor*

Main loop

$\mathcal{T}.root \leftarrow$ MAKEROOT(s_0)
while $t < T$ **do**
  $(e, u) \leftarrow$ **await** any pending call in $\mathcal{T}$
  if e is Speculator **then** ONSPECULATORDONE$(u, e.preds)$
  else ONACTORDONE$(u, e.action)$
end while

function ONSPECULATORDONE$(u, \{\hat{a}^{(i)}\}_{i=1}^{b})$
  for $i = 1$ **to** b **do**
    $s' \leftarrow f(u.s, \hat{a}^{(i)})$; $(h', q') \leftarrow \pi(s')$
    create child c with $(\hat{a}^{(i)}, s', h', q')$
    if $|inflight_actor| < max_inflight_actor$ **then**
      $c.\bar{a}_{actor} \leftarrow$ Actor(h', q')
      $c.\bar{a}_{spec} \leftarrow \hat{g}(s', h', q')$
    end if
  end for
end function

function ONACTORDONE(u, a^*)
  $u.a^* \leftarrow a^*$; **if** $u \neq \mathcal{T}.root$ **return**
  repeat ▷ root resolved: commit and cascade
    $s_{t+1} \leftarrow f(s_t, a^*)$; $t += 1$
    $m \leftarrow$ child of root with MATCH$(c.\hat{a}, a^*)$
    if m is *None* **then** ▷ **Miss**
      $\mathcal{T}.root \leftarrow$ MAKEROOT(s_t); **return**
    end if
    $\mathcal{T}.root \leftarrow m$ ▷ **Hit**
    while $t < T$ **and** root.a^* resolved **do**
      commit; advance root via match; tally **Cascade**
    end while
  until $t \geq T$
end function

sequential and breadth-only baselines.

Losslessness. Like prior speculative-action work (Ye et al., 2025), MSSA only commits an action when the Actor itself has produced it; speculative branches are used solely to pre-warm Actor calls. The committed trajectory is therefore identical, action by action, to the sequential agent.

4 Experimental Setup

Models. We evaluate three Actor–Speculator pairs that span open-weight families and reasoning budgets: **(i)** GPT-OSS-120B as Actor with GPT-OSS-20B as Speculator; **(ii)** GPT-OSS-120B as both Actor and Speculator, with *medium* reasoning effort for the Actor and reasoning *disabled* for the Speculator; and **(iii)** Qwen3-235B (Actor) with Qwen3.6-35B (Speculator) (Yang et al., 2025). In every pairing the Actor is run with *medium* reasoning effort while the Speculator runs with reasoning *disabled*, so in pairing **(ii)** the two roles use the same model and differ only in reasoning configuration. Pricing per million tokens is reported in Appendix C.

Environment. We use the deterministic chess environment from TextArena (Guertler et al., 2025). Actions are UCI strings (e.g. `e2e4`), enabling exact-string matching. Each episode runs 20 steps for the main comparison (Section 5.1) and 15 steps for all subsequent analyses; in every case we replicate five to six trajectories per configuration.

Metrics. We report four metrics. **(i) Time saving** $\Delta T = (T_{seq} - T_{spec})/T_{seq}$, where T_{seq} is the wall-clock time to complete a full episode under the sequential baseline and T_{spec} is the wall-clock time to complete the *same* episode under speculative execution; a larger ΔT means a larger speedup. **(ii) API cost**: the total cost per episode in USD, computed from provider-reported token usage. **(iii) Hit rate** $H = \frac{1}{N}\sum_{t=1}^{N} \mathbf{1}\left[\exists\, i \in \{1, \ldots, b\} : \text{MATCH}(\hat{a}_t^{(i)}, a_t)\right]$, the fraction of the N committed steps at which *at least one* of the Speculator's b predictions $\hat{a}_t^{(i)}$ matched the Actor's ground-truth action a_t. A step is a *hit* if any of its b guesses was correct and a *miss* otherwise. **(iv) Cascaded hit rate**: the fraction of steps in the trajectory that are hits whose immediate predecessor was also a hit. We distinguish two kinds of hits. A *regular hit* is the first hit in a consecutive run—a hit at step t whose predecessor (step $t-1$) was a miss, or which is the first step. A *cascaded hit* is any hit whose predecessor was also a hit; these are exactly the steps that the warm pipeline commits with no cold restart. Cascaded hits are counted separately and are *not* included in the regular hit count, so the total number of hit steps is the sum of regular and cascaded hits. This separation isolates the contribution of depth speculation: only cascaded hits yield the back-to-back commits that breadth-only speculation cannot produce.

Infrastructure and provider pinning. All models are accessed via OpenRouter. Because OpenRouter dynamically routes requests across upstream providers with widely varying latencies, we *pin the provider* in every request to eliminate confounding. Rather than minimizing latency uniformly, we pin providers to deliberately reproduce the asymmetry the framework assumes—a slow, authoritative Actor and a fast Speculator: the Actor is pinned to a higher-latency provider and the

Model Pair	Time saving (%)		Cost ($/ep.)	
	MSSA	Brd. Δpp	MSSA	Brd.
Actor: GPT-120B Spec.: GPT-20B	**33.6**	13.5 (+20.1)	0.428	0.068
Actor: GPT-120B Spec.: GPT-120B	**36.0**	28.2 (+7.8)	0.269	0.036
Actor: Qwen3-235B Spec.: Qwen3.6-35B	**40.0**	19.4 (+20.6)	1.326	0.676

Table 2: MSSA vs. the breadth-only baseline (Brd.) across three Actor–Speculator model pairs, both run at b=2 and *max_inflight_actor*=25. Time saving is relative to sequential execution (higher is better); cost is USD per 20-step episode. Δpp is MSSA − breadth in percentage points.

Speculator to a low-latency one (typically a Groq-hosted endpoint), preferring low variance in both cases. This simulates the realistic deployment regime where the Actor is the dominant latency bottleneck and the Speculator resolves well ahead of it.

5 Results

5.1 MSSA v/s Breadth Pipeline Comparison

Table 2 compares MSSA with the breadth-only baseline (Ye et al., 2025) at b=2 and *max_inflight_actor*=25. Each reported time-saving and cost value is the mean over 3 episodes per configuration. MSSA wins on time saving across all three configurations. The largest margin is for the Qwen3-235B / Qwen3.6-35B pair: 40.0% time saving vs. 19.4% for breadth-only, a +20.6 pp improvement, with the GPT-120B / GPT-20B pair very close behind at +20.1 pp (33.6% vs. 13.5%). The third pair, GPT-120B / GPT-120B, shows the smallest gap (+7.8 pp: 36.0% vs. 28.2%). The gain is largest when the Actor–Speculator latency gap is largest and Speculator predictions are accurate: under those conditions, the speculative tree resolves several levels deep while the root Actor is still in flight, producing long cascades. When the same model serves both roles (GPT-120B / GPT-120B), though with different reasoning-level settings as mentioned previously, the latency gap shrinks and the depth advantage of MSSA shrinks accordingly. More information about the latencies of different models with their providers is provided in Appendix D

These gains are not free. MSSA pre-launches Actor calls at every tree node, so its cost is always higher than breadth-only's. The overhead is roughly 6–7× for the cheaper GPT-OSS pairs ($0.428 vs. $0.068 per episode for GPT-120B/20B; $0.269 vs. $0.036 for GPT-120B/120B) and roughly 2× for the most expensive Qwen3 pair ($1.326 vs. $0.676).

5.2 Effect of Breadth and Depth

To isolate the contributions of breadth and depth, we conduct two complementary ablations with the GPT-OSS-120B/20B pair: a breadth sweep over $b \in \{2, 3, 5, 7, 10, 15, 20\}$ at fixed *max_inflight_actor* $= 15$, and a budget sweep over *max_inflight_actor* $\in \{5, 10, 15, 20, 25, 30\}$ at fixed $b = 2$. Figure 2 reports the resulting time saving for both sweeps on a shared y-axis.

Effect of breadth. The breadth curve (blue) is flat at low b, jumps sharply between b=5 and b=7, and then exhibits clear diminishing returns: it peaks around b=15 and slightly declines beyond. Past the jump, each additional unit of breadth buys progressively less speedup, indicating a practical sweet spot in the range $b \in [10, 15]$ where widening speculation further no longer translates into additional parallelism gain.

Effect of Depth. The Actor budget controls the effective depth of the speculative tree: a larger budget lets more pre-launched Actor calls run concurrently, so the tree can grow deeper while the root Actor is in flight and support longer cascades. The budget curve (red) in Figure 2 shows time saving rising from 18.2% at a cap of 5 to 20.0% at 10 and 29.4% at 15, then flattening: 30.6% at 20, 29.3% at 25, and 30.3% at 30. Beyond a cap of roughly 15–20 the curve enters the shaded plateau region: increasing the budget further does not meaningfully improve time saving, and the ±1 std. dev. band widens, indicating noisier runs under heavier concurrency. The practical sweet spot is therefore around *max_inflight_actor* $\in [15, 20]$, where nearly all of the achievable speedup is captured without the extra API cost and rate-limiting pressure of launching more concurrent Actor calls than the cascade can productively exploit.

Hit composition. Figure 3 explains the jump. The regular hit rate is roughly flat across b: each step's first prediction either matches or it does not, and widening the search at the same step has only a secondary effect on first-step match probability

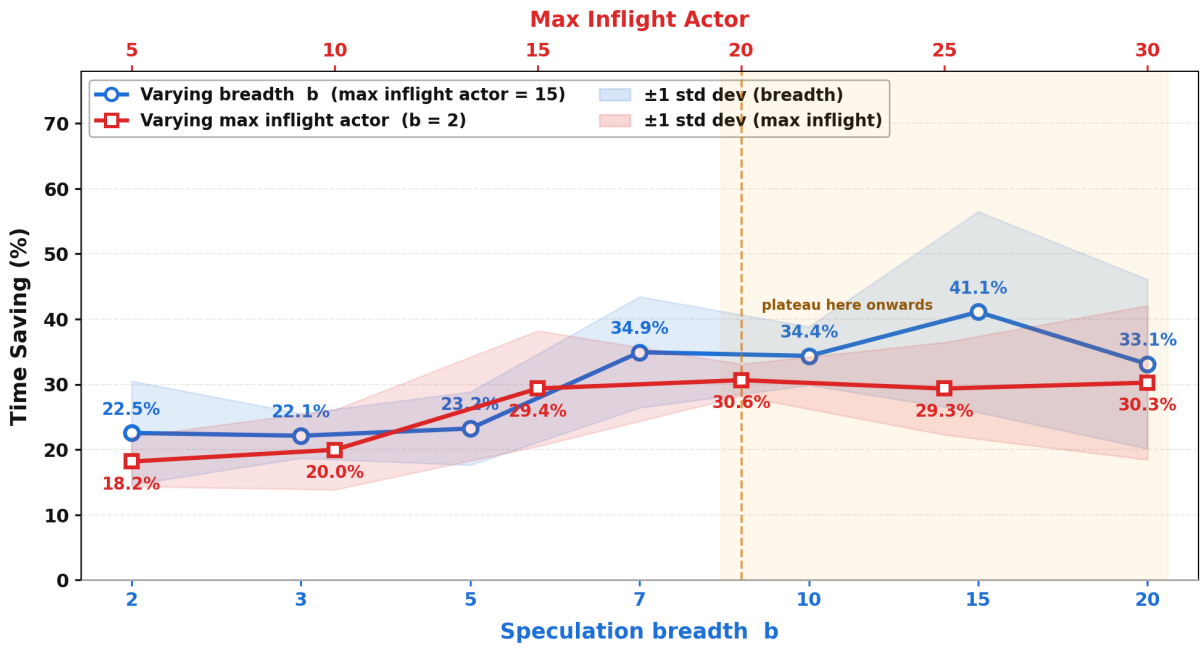

Figure 2: Time saving (%) under two independent ablations on the GPT-OSS-120B/20B pair, plotted on a shared y-axis. The **bottom x-axis (blue)** varies the speculation breadth b at fixed *max_inflight_actor* $= 15$; the **top x-axis (red)** varies *max_inflight_actor* at fixed $b = 2$. Shaded bands report ± 1 std. dev. across replicates, and the orange shaded region marks the plateau where additional Actor budget yields diminishing returns.

for this Speculator. The cascaded hit rate, in contrast, rises steeply with b and overtakes the regular hit rate at b=7, because wider speculation raises the probability that *descendant* Actor calls have already resolved by commit time, enabling longer cascades. This growth, however, does not continue indefinitely: beyond $b \approx 10$ the cascaded hit rate stagnates (Appendix E), so additional breadth no longer lengthens cascades. Pushing b even higher is in fact mildly counter-productive for time saving: under a fixed *max_inflight_actor* budget, a larger b creates more children per level and exhausts the Actor budget at a shallower depth, so the speculative tree cannot grow as deep and cascade chains shorten. This explains both the diminishing returns and the slight decline in time saving at large b.

Takeaway. Both breadth and depth lift time saving steeply at first and then plateau—around $b \in [10, 15]$ and *max_inflight_actor* $\in [15, 20]$, respectively—so MSSA's practical operating point lies at the joint sweet spot of these two plateaus.

6 Analysis

Why cascades grow with b, then stagnate. A cascade is simply a run of consecutive hits. As b increases, the Speculator emits more candidate actions per step, so the chance that one of them

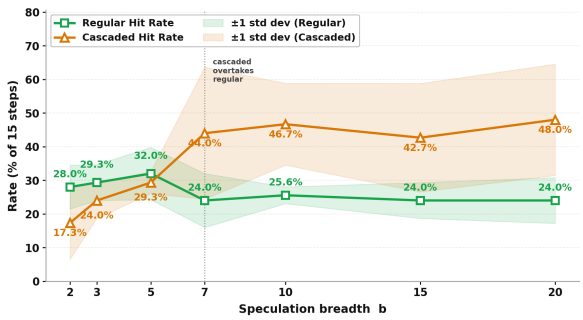

Figure 3: Regular hit rate and cascaded hit rate (fraction of 15 steps) vs. breadth b. The dotted line marks b=7, where cascaded hits overtake regular hits.

matches the Actor's true move goes up—the per-step hit probability rises. Because each step is now more likely to be a hit, the chance that *several steps in a row* are all hits rises even faster, and these consecutive hits are exactly what the cascade mechanism commits without waiting. This is why the cascaded hit rate climbs sharply with b. The effect cannot continue forever, however: once b is large enough to already cover the Speculator's plausible moves, further widening only adds redundant or low-probability candidates that the Actor would never take, so the per-step hit probability—and with it the cascade rate—stops improving and stagnates. The empirical distribution of cascade lengths

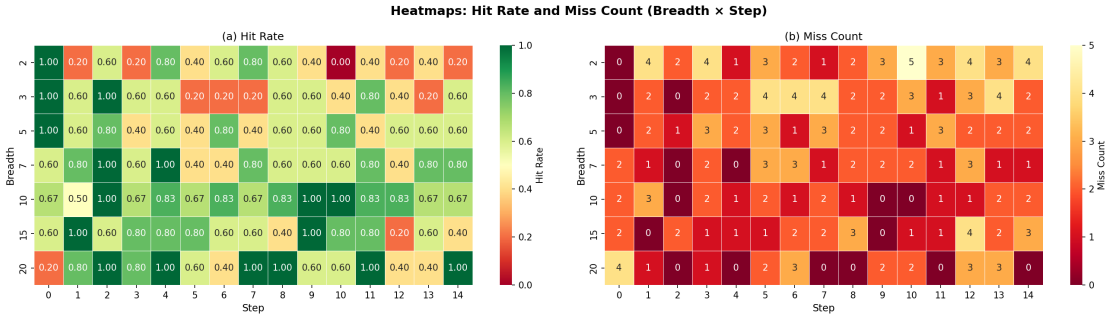

Figure 4: (a) Hit rate and (b) miss count across speculative breadth b (rows) and step index 0–14 (columns) for the GPT-OSS-120B / GPT-OSS-20B pair.

(Appendix E, Figure 6) confirms this: the distribution shifts rightward as b grows, with mean cascade length rising from 1.6 at b=2 to 3.9 at b=10, after which it no longer lengthens.

Predictability across step index. Hit rate also depends on *where* in the game a step occurs. The opening moves of a chess game come from a small, well-studied set of standard openings, so a fast Speculator can guess them reliably—early steps are easy to predict. As the game progresses the board drifts away from these familiar patterns, the number of reasonable moves grows, and the Actor's choice becomes harder to anticipate from the state alone, so hit rate naturally tends to fall over the course of a trajectory. Breadth counteracts this decline: with more candidates per step, even a hard-to-predict mid- or late-game position is likely to have *at least one* of the b guesses match the Actor. The per-step hit-rate heatmap in Figure 4 shows exactly this pattern—step 0 is near 1.0 even at small b, late steps collapse to 0.0–0.2 at b=2 but recover to 0.67–1.0 at b=10 and b=20—so wide speculation effectively flattens the step-wise hit-rate decline by providing broad coverage in the positions that are otherwise hardest to predict. A more detailed reading of the heatmap, including the step-0 timing artifact at large b, is provided in Appendix F.

7 Conclusion

Multi-Step Speculative Actions (MSSA) unifies breadth and depth into a single asynchronous, event-driven pipeline parameterized by b and *max_inflight_actor*. Across three Actor–Speculator pairs, MSSA improves time saving over the strongest breadth-only baseline by up to +20.6 percentage points, with cascaded hits—enabled by the depth dimension—as the dominant driver of these gains. Both axes plateau (around $b \in [10, 15]$ and *max_inflight_actor* $\in [15, 20]$), defining MSSA's practical operating point. The committed trajectory is identical to that of a sequential agent, so accuracy is never traded for speed.

Limitations

Single environment. We evaluate in only one environment (chess in TextArena), a deliberate scoping choice: our goal is to propose MSSA and analyze it deeply enough to make the breadth–depth dynamics interpretable rather than to maximize benchmark coverage. Applying MSSA to stochastic, partially-observed, or longer-horizon agentic tasks is a natural direction for follow-up work.

Inference under rate limiting. We access all models through OpenRouter, which frequently throttles concurrent requests. MSSA still beats the breadth-only pipeline under these conditions, suggesting our gains are conservative; It would be interesting to measure how much further multi-step speculation can go when this constraint is removed e.g. with models hosted on dedicated cloud hardware, run locally, or served through a powerful enterprise API without rate limiting.

Blind tree expansion. MSSA expands tree nodes blindly: every Speculator-returned action becomes a child whose Actor and Speculator calls are pre-launched, incurring substantial API cost. A natural next step is to expand *selectively* based on prediction confidence—e.g. verbalized self-assessment or Speculator logits—to retain most of the speedup at lower cost. Jointly learning b and *max_inflight_actor* online (Guan et al., 2025) are further directions.

References

Gregor Bachmann, Sotiris Anagnostidis, Albert Pumarola, Markos Georgopoulos, Artsiom Sanakoyeu, Yuming Du, Edgar Schönfeld, Ali Thabet, and Jonas Kohler. 2025. Judge decoding: Faster speculative sampling requires going beyond model alignment. *arXiv preprint arXiv:2501.19309*.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple llm inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

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

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

Leon Guertler, Bobby Cheng, Simon Yu, Bo Liu, Leshem Choshen, and Cheston Tan. 2025. Textarena. *arXiv preprint arXiv:2504.11442*.

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

Zixiao Huang, Wen Zeng, Tianyu Fu, Tengxuan Liu, Yizhou Sun, Ke Hong, Xinhao Yang, Chengchun Liu, Yan Li, Quanlu Zhang, and 1 others. 2025. Reducing latency of llm search agent via speculation-based algorithm-system co-design. *arXiv preprint arXiv:2511.20048*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Sehoon Kim, Karttikeya Mangalam, Suhong Moon, Jitendra Malik, Michael W Mahoney, Amir Gholami, and Kurt Keutzer. 2023. Speculative decoding with big little decoder. *Advances in Neural Information Processing Systems*, 36:39236–39256.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 28935–28948. PMLR.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2026. Eagle-3: Scaling up inference acceleration of large language models via training-time test. *Advances in Neural Information Processing Systems*, 38:136737–136756.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, and 1 others. 2024. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pages 52989–53046.

Jiageng Mao, Yuxi Qian, Junjie Ye, Hang Zhao, and Yue Wang. 2023. Gpt-driver: Learning to drive with gpt. *arXiv preprint arXiv:2310.01415*.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. Gaia: a benchmark for general ai assistants. In *International Conference on Learning Representations*, volume 2024, pages 9025–9049.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, and 1 others. 2024. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 932–949.

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

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*.

Chonghao Sima, Katrin Renz, Kashyap Chitta, Li Chen, Hanxue Zhang, Chengen Xie, Jens Beißwenger, Ping

Luo, Andreas Geiger, and Hongyang Li. 2024. Drivelm: Driving with graph visual question answering. In *European conference on computer vision*, pages 256–274. Springer.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. 2018. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*, 31.

Yifan Sui, Han Zhao, Rui Ma, Zhiyuan He, Hao Wang, Jianxun Li, and Yuqing Yang. 2026. Act while thinking: Accelerating llm agents via pattern-aware speculative tool execution. *arXiv preprint arXiv:2603.18897*.

Nadav Timor, Jonathan Mamou, Daniel Korat, Moshe Berchansky, Gaurav Jain, Oren Pereg, Moshe Wasserblat, and David Harel. 2025. Accelerating llm inference with lossless speculative decoding algorithms for heterogeneous vocabularies. *arXiv preprint arXiv:2502.05202*.

Tao Tu, Mike Schaekermann, Anil Palepu, Khaled Saab, Jan Freyberg, Ryutaro Tanno, Amy Wang, Brenna Li, Mohamed Amin, Yong Cheng, and 1 others. 2025. Towards conversational diagnostic artificial intelligence. *Nature*, 642(8067):442–450.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024a. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. 2024b. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. *Findings of the Association for Computational Linguistics: ACL 2024*, pages 7655–7671.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. 2025. {τ}-bench: A benchmark for \underline{T}ool-\underline{A}gent-\underline{U}ser interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Naimeng Ye, Arnav Ahuja, Georgios Liargkovas, Yunan Lu, Kostis Kaffes, and Tianyi Peng. 2025. Speculative actions: A lossless framework for faster agentic systems. *arXiv preprint arXiv:2510.04371*.

Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. 2024. Assistantbench: Can web agents solve realistic and time-consuming tasks? In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 8938–8968.

Yangyang Yu, Zhiyuan Yao, Haohang Li, Zhiyang Deng, Yuechen Jiang, Yupeng Cao, Zhi Chen, Jordan W Suchow, Zhenyu Cui, Rong Liu, and 1 others. 2024. Fincon: A synthesized llm multi-agent system with conceptual verbal reinforcement for enhanced financial decision making. *Advances in Neural Information Processing Systems*, 37:137010–137045.

A Models and Generation Parameters

We list every model used in our experiments below, with a link to its HuggingFace repository (model card and weights):

1. **GPT-OSS-120B** — `openai/gpt-oss-120b`
2. **GPT-OSS-20B** — `openai/gpt-oss-20b`
3. **Qwen3-235B** — `Qwen/Qwen3-235B-A22B`
4. **Qwen3.6-35B** — `Qwen/Qwen3.6-35B-A3B`

Generation parameters. The same parameters are used across all experimental configurations and were set in the YAML configs of the released codebase.

- **Actor:** `max_tokens = 30,000`; reasoning *enabled* (`disable_thinking = false`), `reasoning_effort = medium`.
- **Speculator:** `max_tokens = 10,000`; reasoning *disabled* (`disable_thinking = true`);
- **Sampling:** `temperature` and `top_p` are left at each provider's defaults; we do not override either.

B Pipeline Walkthrough

Here, we expand Section 3 with a step-by-step trace of the three pipelines in Figure 1, using the figure's notation: s_t is the confirmed state at step t, a_t the Actor's ground-truth action, $\hat{a}_t^i$ the Speculator's i-th candidate at step t, and $\hat{s}_t^i$ the hypothetical state reached by applying $\hat{a}_t^i$. Blue arrows denote Actor calls, red arrows Speculator calls, hatched red regions pruned subtrees, and the purple region the wall-clock time saved.

Sequential pipeline. Starting from s_0, the Actor is called to produce a_0, which advances the environment to s_1; the Actor is then called from s_1 to

produce a_1, yielding s_2, and so on through $a_3 \to s_4$. No call is ever issued before its predecessor resolves, so the total wall-clock time is the sum of four Actor latencies with no overlap.

Breadth-only pipeline. At s_0 the Actor call for a_0 is issued, and *simultaneously* the Speculator predicts two candidates $\hat{a}_0^1$ and $\hat{a}_0^2$, leading to hypothetical states $\hat{s}_1^1$ and $\hat{s}_1^2$. An Actor call is pre-launched from each. When the ground-truth Actor at s_0 returns a_0, the system keeps whichever branch matches: if $\hat{a}_0^2$ matches a_0, the subtree at $\hat{s}_1^1$ is pruned (hatched red) and the pre-launched Actor at $\hat{s}_1^2$ carries forward to s_2. Because the next step's Actor was already in flight, the cold inter-step gap is removed—but speculation extends only one level, so step $t{+}1$ begins with no pre-computation and the pattern restarts cold at every confirmed state.

Multi-step pipeline (ours). At s_0 the Actor and Speculator launch concurrently as before. When the Speculator at s_0 resolves with $\hat{a}_0^1, \hat{a}_0^2$, two children $\hat{s}_1^1, \hat{s}_1^2$ are created and *each* pre-launches both an Actor call and its own Speculator call. When the Speculator at $\hat{s}_1^2$ resolves with $\hat{a}_1^1, \hat{a}_1^2$, grandchildren $\hat{s}_2^1, \hat{s}_2^2$ are created and again pre-launch their Actor and Speculator calls. This recursion continues—bounded only by the inflight Actor budget—building a tree of depth d entirely *while the root Actor at s_0 is still waiting to resolve*. When the root Actor returns a_0 and $\hat{a}_0^2$ matches, the subtree at $\hat{s}_1^1$ is pruned and $\hat{s}_1^2$ becomes the new root. Crucially, the Actor at $\hat{s}_1^2$ may have *already* resolved during the root Actor's wait; if so, the algorithm commits it immediately and advances again. The same two-way match applies at each promoted root, so a streak of correct predictions ($\hat{a}_1^2$ matches, then $\hat{a}_2^2$ matches, . . .) commits several steps back-to-back with no network wait—the purple cascade region. A mismatch at any depth abandons the remaining subtree and restarts speculation from the newly confirmed state, costing exactly the sequential baseline at that step.

C Model Pricing

Model	Reasoning	In ($/M)	Out ($/M)
GPT-OSS-120B	Medium	0.127	0.554
GPT-OSS-20B	None	0.058	0.206
Qwen3-235B	Standard	0.240	2.274
Qwen3.6-35B	Standard	0.449	2.890

Table 3: Input/output token pricing for models evaluated in this work (per million tokens).

The Table 3 above provides the average cost incurred by each model for its respective reasoning setting when accessed through OpenRouter.

D Provider Latency Variance

OpenRouter serves the *same* model through multiple upstream providers, and as Figure 5 shows, these providers exhibit markedly different latencies—differing by more than an order of magnitude for an identical model. Because OpenRouter routes requests across these providers dynamically, an unpinned run gives no control over which provider serves the Actor versus the Speculator. This directly threatens the core assumption of our algorithm: that the Actor is slow and the Speculator is fast. Without pinning, the Speculator could be routed to a slow provider and the Actor to a fast one, collapsing or even inverting the latency gap the speculative tree relies on, and making measured speedups uninterpretable. We therefore pin the provider via the `provider.order` field in every request, deliberately selecting a higher-latency provider for the Actor and a low-latency one (typically Groq-hosted, also preferred for low variance) for the Speculator, so the slow-Actor / fast-Speculator regime the algorithm assumes is guaranteed and held fixed across all runs.

E Cascade Length Distribution

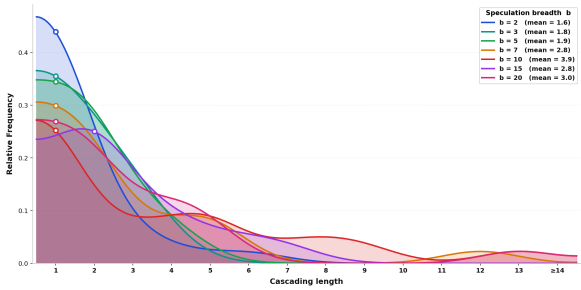

Figure 6: Distribution of cascade lengths across speculation breadths b (Gaussian-smoothed, σ=0.9).

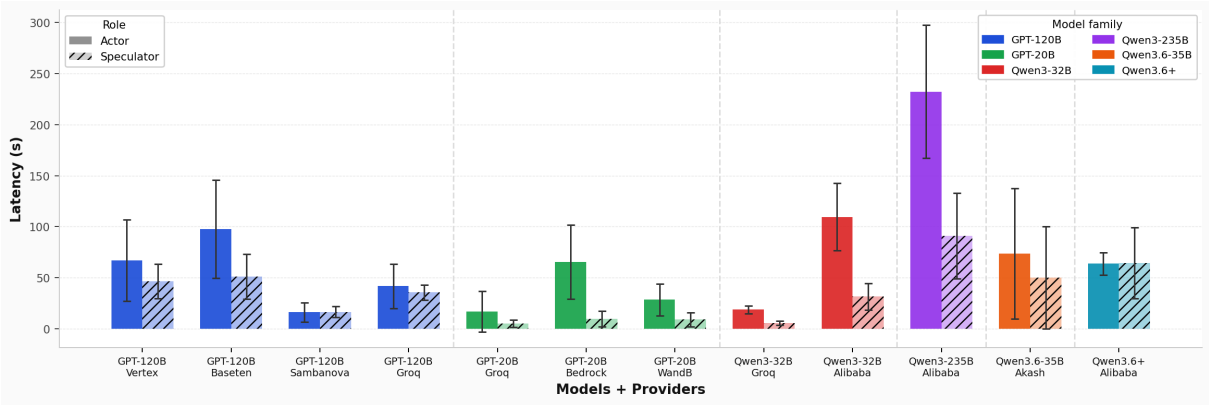

Figure 5: Mean latency and ± 1 std. dev. for each model and provider in Actor and Speculator roles.

We define the *cascade length* of a cascade as the number of steps committed back-to-back in a single warm-pipeline burst—that is, the number of consecutive hits the algorithm confirms without any cold restart, starting from the first hit in the run and continuing through every subsequent hit until a miss or a still-pending Actor ends the streak. A cascade length of 1 means an isolated hit (the next step was a miss or required a fresh wait), whereas a cascade length of, say, 5 means five consecutive steps were committed with no network wait between them. Longer cascades are precisely where the depth component of MSSA pays off, so the full distribution of cascade lengths—not just its mean—is informative: a distribution concentrated at length 1 indicates breadth-like behavior, while a distribution with substantial mass at larger lengths indicates that deep speculation is actively saving steps. Figure 6 plots this distribution for each speculation breadth b.

At b=2 the distribution is heavily right-skewed with mean cascade length 1.6: most hits are isolated. As b grows the distribution shifts right and its tail thickens. The mean cascade length increases to 1.9 at b=5, 2.8 at b=7 and b=15, and peaks at 3.9 for b=10. Non-trivial probability mass appears at cascade lengths 8–14 once $b \geq 10$, but the marginal benefit of widening further is small—consistent with the time-saving plateau in Figure 2.

F Per-Step Hit Heatmaps

Here, we provide further explanation about the per-step hit-rate heatmap shown in Figure 4. The dominant trend in Figure 4(a) is that increasing b raises the probability that *any* given step is a hit: reading down each column, the hit rate generally rises with breadth, and the late-game columns in particular brighten substantially as b grows. Concretely, late-game steps are hard for the Speculator at low b—at b=2, steps 10–14 frequently fall to 0.0–0.2 hit rate—but increasing b flattens this decline, with the same steps reaching 0.67–1.0 at b=10 and b=20. Broader coverage thus provides resilience to late-game positional diversity even when no single prediction is reliable.

One apparent exception is worth explaining. Step 0 (the opening move) achieves near-perfect hit rates at *small* b, but its hit rate is actually *lower* at large b than at small b—the opposite of the general trend. This is not because the opening becomes harder to predict; empirically, the Actor at step 0 returns very quickly, sometimes even *before* the Speculator, because the start position is not complex and requires little reasoning about the opening. When the ground-truth Actor resolves before the Speculator's predictions are available, the step is recorded as a miss regardless of how predictable the move was. At larger b this effect is amplified: the much larger volume of concurrent requests makes OpenRouter more likely to throttle or rate-limit the Speculator, delaying its predictions further and causing the fast step-0 Actor to return ahead of it more often. Step 0 therefore reflects a timing artifact of our inference setup rather than a genuine drop in opening-move predictability.

G Prompts

We use a single fixed prompt per role across all configurations and runs; only the reasoning-effort setting differs between the Actor and the Speculator (Section 4). Both roles share the same system prompt and the same environment observation; the Speculator additionally receives the instruction in Figure 7 appended to the observation.

Actor / Speculator system prompt.

```
You are a competitive game player. Make
sure you read the game instructions
carefully, and always follow the
required format. Reason extensively
and deeply, and then return your move.
Important: always return a valid move
and a valid move only, even if you are
uncertain.
```

Environment observation (user message). The TextArena chess environment supplies, at every step, an observation of the form:

```
[GAME] You are playing as {turn} in
a game of Chess. Make your moves in
UCI format enclosed in square brackets
(e.g., [e2e4]).
[GAME] The current board is:
{board}
[GAME] The valid moves are:
{valid_moves}.
```

where `{turn}` is the side to move, `{board}` is the rendered board (derived from the FEN string), and `{valid_moves}` is the list of legal UCI moves.

Speculator instruction. For the Speculator, the following is appended to the observation, with `{num_guesses}` set to the breadth b:

```
Reason very very succinctly about
the next move, return {num_guesses}
candidates for the next move, ordered
from most to least favorable. IMPORTANT
FORMAT: Your response must be a
comma-separated list of moves in
[UCI_MOVE] format, like: [e2e4], [e2e3],
[d2d4] (exact syntax with square
brackets and commas between moves!).
For example: [e2e4], [d2d4], [g1f3].
Ensure all moves are from the list
of valid moves. Even if uncertain,
return exactly {num_guesses} candidates.
Reason very very quickly, no need to
truly think about each of the candidates,
just return the most likely candidates
and return precisely {num_guesses} valid
moves - no more, no less.
```

Figure 7: Speculator instruction appended to the environment observation.

Retry prompt. If a model returns a malformed or illegal move, the following is appended and the call is re-issued (up to a fixed retry budget):

```
Attempt {attempt} failed, please
remember that you are acting as {role}
in the chess game, and you need to make
a valid move from the last valid moves
list. Return the move in the format
[UCI_MOVE], for example [e2e4].
```