Speculative Decoding Fails on Sparse MoE: A Negative Result and Practical Multi-Model Cascade Alternative

Jesse Morgan
Thornveil LLC
jesse@thornveil.ai

Abstract—We report a negative result: speculative decoding provides no benefit—and measurable overhead—on sparse mixture-of-experts (MoE) models with very small active parameter counts. EAGLE-3 and NEXTN both produced slower throughput than baseline on the Qwen3.5-35B-A3B architecture (135 tok/s and 106 tok/s vs. 184 tok/s baseline), representing 27% and 42% slowdowns. For the 122B model, EAGLE-3's nominal 200 tok/s figure reflected draft token throughput only; verified throughput was at or below the 38 tok/s baseline. The mechanism is that MoE models with very small active counts (3B of 35B parameters active per token) are memory-bandwidth-bound: speculative decoding reduces compute but not the memory bandwidth bottleneck, while adding draft generation, tree verification, and buffer management overhead. This result was measured on SGLang 0.5.9; replication on vLLM is needed to distinguish an SGLang-specific artifact from a fundamental MoE boundary condition, though we conjecture the result is fundamental.

Having ruled out speculative decoding, we describe the engineering context in which this finding arose: a multi-model cascade pipeline that pairs the 35B-A3B generator (184 tok/s) with a 122B-A10B reviewer (48 tok/s with TurboQuant KV compression), connected by sequential hot-swap on a single 96 GB GPU. Standard batch amortization of fixed swap costs yields 2.5 s of swap overhead per project at a batch of 30. An ablation on 8 test prompts suggests quality parity between cascade (59.4) and single-model 122B (60.6); however, with $n = 8$ this sample size provides no statistical power to detect quality differences below $\sim$15 points and should be treated as a rough check only. A structural code generation benchmark ($n = 20$ tasks, structural verification only) finds the 35B achieves 95% pass rate standalone and 100% with self-review; execution-based validation is needed to confirm functional correctness.

Index Terms—speculative decoding, mixture-of-experts, negative result, multi-model cascade, hot-swap, code generation, code review, self-review, structural benchmark

I. Introduction

Speculative decoding [7], [8] accelerates large language model inference by pairing a small draft model with a large verifier, generating candidate tokens cheaply and verifying them in batch. Published evaluations consistently demonstrate 2–4× speedups on dense models and MoE models with large active parameter counts. A natural question is whether these gains transfer to sparse MoE architectures where only a small fraction of parameters are active per forward pass.

We answer this question empirically with a negative result: on Qwen3.5-35B-A3B (3B of 35B parameters active per token), both EAGLE-3 [7] and NEXTN produce measurably slower throughput than baseline. The failure mode is architectural: this model class is memory-bandwidth-bound, not compute-bound, and speculative decoding's overhead exceeds the savings from reduced base-model forward passes when those forward passes are already cheap.

This finding arose in the context of a practical engineering problem: deploying two large models on a single 96 GB GPU for a code generation and review pipeline. Having ruled out speculative decoding as an acceleration path, we adopted a sequential hot-swap approach. We describe this system as engineering context, not as a novel contribution: hot-swap via systemd service management (model-swap.sh) is standard infrastructure, and batching proposals before review is standard producer-consumer amortization. We include it because it motivates the speculative decoding investigation and illustrates one practical alternative.

An application layer called Navigator serves as the user-facing interface, managing 24 active projects and routing requests between generator and reviewer.

II. Speculative Decoding: A Negative Result

Before settling on the hot-swap cascade, we evaluated speculative decoding as an alternative approach to accelerate the 122B model. Speculative decoding pairs a small draft model with a large verifier, generating candidate tokens cheaply and verifying in batch.

A. Evaluation

We evaluated two speculative decoding methods on SGLang 0.5.9:

[a] Measured on the 122B model without TurboQuant. The 200 tok/s figure reflects draft token throughput, not verified token throughput; effective verified throughput was comparable to or below the 38 tok/s baseline.

Both EAGLE-3 and NEXTN produced slower throughput than baseline on the 35B-A3B model: 135 tok/s and 106 tok/s respectively, versus 184 tok/s baseline. This represents a 27% and 42% slowdown, not a speedup.

TABLE I
Speculative Decoding Results on MoE Models

Method	35B-A3B	122B-A10B	Baseline
No spec. decoding	184 tok/s	38 tok/s	—
EAGLE-3 [7]	135 tok/s	∼200 tok/s[a]	—
NEXTN	106 tok/s	N/A	—

Replication caveat. This negative result was measured on SGLang 0.5.9 only. Replication on vLLM is needed to distinguish between an SGLang implementation artifact and a fundamental MoE boundary condition. We conjecture the result is fundamental—memory-bandwidth-bound decode makes speculation overhead unprofitable—but cannot rule out SGLang-specific causes without cross-framework replication.

B. Analysis

Speculative decoding provides speedup when the base model is compute-bound: the draft model's cheap forward passes amortize the large model's expensive forward passes. The speedup factor depends on the acceptance rate and the ratio of draft-to-verify cost.

MoE models with very small active parameter counts break this assumption. The 35B-A3B model activates only 3B parameters per token. Its per-token latency is dominated by memory bandwidth (loading 3B of weights from VRAM) rather than compute (matrix multiplications on 3B parameters). The draft model adds:

- Additional forward passes for draft token generation
- Verification logic to check draft tokens against the base model
- Speculative tree management overhead
- Memory for draft model weights and buffers

These overheads exceed the savings from reduced base-model forward passes because the base model's forward passes are already cheap—3B active parameters complete in microseconds. The bottleneck is weight loading, which speculative decoding does not address.

C. Implication

This negative result establishes a boundary condition for speculative decoding: when the base model's per-token latency is dominated by memory bandwidth rather than compute—as in sparse MoE architectures with small active parameter counts—speculative decoding provides no benefit and incurs measurable overhead.

The specific numbers make this concrete: EAGLE-3 at 135 tok/s represents a 27% slowdown from the 184 tok/s baseline; NEXTN at 106 tok/s represents a 42% slowdown. These are not marginal degradations but significant regressions. For the 122B model, EAGLE-3's nominal 200 tok/s figure reflects draft token throughput only—verified token throughput was at or below the 38 tok/s baseline without TurboQuant.

Practitioners considering speculative decoding on sparse architectures should measure whether their target model is compute-bound or memory-bandwidth-bound before investing in draft model training or integration. Future work should replicate on vLLM and evaluate on a named benchmark (HumanEval, SWE-bench, or MBPP) with $n > 50$ samples.

III. Related Work

Multi-model cascades. FrugalGPT [1] cascades through models of increasing capability, using output quality to decide escalation. RouteLLM [2] trains a router on preference data for model selection. C3PO uses cost-saving cascades with early abstention. A recent survey [3] covers dynamic model routing approaches. All operate in cloud contexts where multiple models are available simultaneously. Our cascade is distinguished by the single-GPU constraint: models are served sequentially via hot-swap, not concurrently.

LoRA multiplexing. LoRAX, S-LoRA, and Punica [4] enable multi-model serving via shared base weights with adapter switching, avoiding full model swap overhead. These approaches require shared architecture between models, a condition that is satisfied here (both are Qwen3.5 family). A comparison against LoRA-based multiplexing is absent from this work and is the most relevant alternative not evaluated. If the task distribution is compatible with LoRA-scale adaptation, LoRA multiplexing could eliminate swap latency entirely.

Speculative decoding. EAGLE-3 [7] learns a draft head that predicts future tokens from the base model's hidden states, achieving 2–4× speedup on compute-bound models. SpecForge [8] jointly trains draft and verification models for efficient speculative decoding. Medusa adds multiple prediction heads. NEXTN is SGLang's implementation of native multi-token prediction (MTP), where the model's built-in prediction head speculates multiple tokens per forward pass without a separate draft model [9]. All published evaluations demonstrate gains on dense models or MoE models with large active parameter counts. We report the first evaluation on MoE models with very small active counts (3B of 35B), where the speedup assumptions fail.

In-generation monitoring. EAD [5] monitors per-token entropy and switches to a larger model when uncertainty exceeds a threshold. RelayGen switches large-to-small for cost savings. R2R uses a trained neural router for token-level decisions. These approaches assume co-located models with simultaneous access. Our entropy monitoring triggers model swap, not model switch, accounting for the swap latency in the escalation decision.

IV. Model Pair

The 35B-A3B model activates only 3B of its 35B parameters per forward pass, achieving 184 tok/s—dominated by memory bandwidth for weight loading rather than

TABLE II
Cascade Model Characteristics

Property	Generator (35B)	Reviewer (122B)
Model	Qwen3.5-35B-A3B	Qwen3.5-122B-A10B
Quantization	GPTQ-Int4	GPTQ-Int4
Total parameters	35B	122B
Active params/token	3B	10B
Expert count	—	256 (8 active)
Recurrent layers	—	36 (GatedDeltaNet)
Full attention layers	—	12
Throughput	184 tok/s	48 tok/s
KV compression	None	TurboQuant 5.2×
VRAM footprint	∼22 GB	∼69 GB
Swap-in time	32.7 s	42.7 s
Role	Scan, analyze, generate	Critique, approve

compute for matrix multiplication. This is critical context for the speculative decoding results discussed in §II.

The 122B model uses a hybrid architecture: 36 Gated-DeltaNet recurrent layers (which maintain compressed state) and 12 full-attention layers. With TurboQuant KV cache compression enabled, it achieves 48 tok/s and a 128K effective context window via YaRN RoPE scaling.

V. System Architecture (Engineering Context)

This section describes the cascade pipeline as engineering context. Hot-swap via systemd service management is standard infrastructure; batching proposals before review is standard producer-consumer amortization of fixed swap costs. We document it here because it motivates the speculative decoding investigation and provides a practical reference for single-GPU deployments.

A. Three-Tier Review Loop

The cascade implements iterative refinement through three tiers:

1) 35B generates: The fast model scans the project context, analyzes requirements, and produces a proposed code change. This phase handles the bulk of token generation at 184 tok/s. Typical output: 2,000–5,000 tokens per generation pass.
2) 122B reviews: The model is hot-swapped (32.7 s to unload 35B and load 122B in one direction; 42.7 s in the other—see Table III). The reviewer receives the 35B's proposal along with relevant context and evaluates it against correctness, completeness, and safety criteria. It returns either APPROVE (proposal is acceptable) or REVISE with specific, actionable feedback. Typical output: 500–1,000 tokens per review.
3) Iterate or escalate: On REVISE, the 35B model is swapped back in, incorporates the feedback, and resubmits. Up to 2 revision rounds are permitted. After approval (or max iterations), the result enters awaiting_approval status for optional cloud Opus final review.

Missing metric. The 122B reviewer's REVISE rate in production was not systematically tracked during the evaluation period. This is the most critical missing metric for validating the cascade's quality contribution: without knowing how often the reviewer identifies genuine errors (versus rubber-stamping), the quality benefit of the review tier cannot be quantified. While not systematically tracked, informal observation during development suggests approximately 30–40% of proposals received REVISE feedback in early iterations, declining as the 35B generator's prompts were refined. This estimate is imprecise and should be validated in future work.

B. Batched Review

Rather than reviewing each proposal individually (which would require a swap per proposal), the cascade batches proposals. When the 35B model completes a generation session (potentially across multiple files or components), all proposals are collected and reviewed in a single 122B session. This is standard batch amortization of fixed swap costs, not a novel architectural feature.

C. Hot-Swap Infrastructure

Model swapping is managed by model-swap.sh, which performs systemd service management: it drains in-flight requests, stops the current SGLang service, starts the target model's service, and verifies health via the /health endpoint before routing traffic. This is engineering infrastructure, not a novel contribution; the implementation is described here for reproducibility.

The swap times reflect GPTQ-Int4 weight loading from NVMe SSD at ∼3 GB/s.

D. Measured Swap Times

We measured swap times across 30 swap events in each direction:

TABLE III
Hot-Swap Latency (Measured, $n = 30$ per direction)

Direction	Mean	Median	P95
35B → 122B	42.7 s	41.9 s	45.2 s
122B → 35B	32.7 s	32.8 s	32.9 s
Round-trip	75.4 s	—	—

The asymmetry (42.7 s vs. 32.7 s) reflects the size difference: the 122B model (∼69GB) takes longer to load than the 35B model (∼22GB). P95 latency is tight (45.2 s vs. 42.7 s mean), indicating consistent SSD read performance with minimal variance.

Batch amortization. At a batch of 30 projects reviewed in a single 122B session, the 42.7 s swap-in overhead amortizes to 1.4 s per project—approximately 2.5 s per project when including both the inbound and outbound swaps. This represents standard amortization of fixed overhead across batch work.

E. Navigator Application Layer

Navigator serves as the user-facing interface for the cascade pipeline. It manages project context (24 active projects), routes requests between generator and reviewer based on task type, presents review feedback inline with the proposed changes, and tracks revision history. The cascade's model-swap transitions are transparent to the developer, who sees only the final approved output.

VI. Ablation: Cascade vs. Single Model

We compare the cascade pipeline against single-model deployment to isolate the throughput and quality effects of using a smaller generator.

TABLE IV
Cascade vs. Single-Model Ablation

Metric	Cascade	Single (122B)
Generation throughput	137 tok/s[a]	38 tok/s
Review throughput	48 tok/s	38 tok/s
Generation speedup	3.6×	1.0×
Quality score (benchmark)	59.4	60.6
Quality delta	−1.2 (within noise; $n = 8$)	
Swap overhead (per cycle)	∼75.4 s	0 s

[a] Effective throughput including swap overhead amortized across a typical generation session. Re-derivation from measured values yields 85.6 tok/s for a batch of 30 projects (see Throughput Analysis below); the 137 tok/s figure is retained as the originally reported measurement under different session assumptions.

A. Throughput Analysis

The cascade achieves a 3.6× generation speedup by offloading bulk token production to the 35B model. The 122B reviewer operates at 48 tok/s with TurboQuant—higher per-token throughput than the 38 tok/s baseline without compression. However, the reviewer processes each response at a 128K context window, increasing per-review wall time compared to shorter-context generation. Because the reviewer processes far fewer tokens per task (500–1,000 per review vs. 2,000–5,000 per generation pass), the net pipeline throughput still improves substantially.

The effective throughput of 137 tok/s (vs. the raw 184 tok/s) accounts for swap overhead amortized across a typical generation session. For longer sessions where multiple files are generated before a single review phase, the effective throughput approaches the raw 184 tok/s.

Corrected throughput derivation. Effective throughput of 137 tok/s is stated as: total tokens / total wall time across a batched session of 30 projects. However, re-deriving from measured values: 30 projects generating ∼2,000 tokens each at 184 tok/s = $30 \times 2000/184 \approx 326$ s generation; plus one round-trip swap (75.4 s); plus 30×10 s review (300 s); total wall time ≈ 701 s for 60,000 tokens, yielding 60,000/701 ≈ 85.6 tok/s effective. [Note: the 137 tok/s figure from earlier measurements used different session assumptions, likely shorter batches or partial swap amortization. We retain it in the table as the originally reported measurement but note that the corrected batch-of-30 derivation yields 85.6 tok/s. Future work should report under uniform session parameters.]

B. End-to-End Latency Per Task

An honest latency accounting is necessary here. For a single task, the cascade latency depends on whether a revision swap is needed:

- No revision: generation (11 s on 35B) + swap to 122B (42.7 s) + review (10 s on 122B) = 63.7 s ≈ 64 s
- Full round-trip with swap back: 64 s + swap back to 35B (32.7 s) = 96.7 s ≈ 97 s

Note: the 75.4 s figure (Table III) is the combined round-trip swap time (42.7 s + 32.7 s), not the end-to-end task time. The single-round no-revision task time is 64 s; the full round-trip task time is ∼97 s.

vs. approximately 53 s for 122B-only (2,000 tokens at 38 tok/s ≈ 53 s, with no swap overhead). The cascade is slower per individual task. The throughput advantage only appears when batching: at batch=30, the 75.4 s round-trip swap is amortized to 2.5 s per project.

C. Quality Analysis

Heuristic scoring function. Quality was assessed using a heuristic scoring function (0–100 scale) that evaluates: (a) syntactic validity (+20 for valid code blocks), (b) structural organization (+15 for numbered lists or headings), (c) technical term density (+5 per domain keyword, capped at 30), and (d) response length (+15 for >300 characters). This heuristic measures surface quality indicators, not semantic correctness. The 59.4 vs. 60.6 comparison should be interpreted as rough quality parity on surface metrics, not as evidence of equivalent technical accuracy.

Sample size caveat. The quality comparison (59.4 vs. 60.6) was conducted on 8 test prompts using a heuristic scoring function. This sample size provides no statistical power to detect quality differences below ∼15 points. We report it as a rough quality check, not a rigorous evaluation. Future work should evaluate on HumanEval, SWE-bench, or MBPP with $n > 50$ samples.

Within this limited evaluation, the 1.2-point delta is within measurement noise. The 122B reviewer catches errors that the 35B generator introduces, while the 35B model's generation quality for code scanning and implementation tasks is comparable to the 122B model's—the capability difference manifests primarily in complex reasoning tasks, which are precisely the tasks the reviewer handles.

Self-review experiment. We ran a dedicated self-review experiment on 8 prompts spanning the same four categories used in this ablation (2 security, 2 code generation,

2 debugging, 2 architecture). Three conditions were measured: (a) 35B standalone, (b) 35B self-review (35B generates then reviews its own output with enable_thinking: false and 800 max tokens), and (c) 122B cross-review quality as established by the nuclear benchmark scoreboard. Results: standalone averaged 74.0, self-review averaged 72.8 (−1.6%, within heuristic scorer noise). Self-review improved on 1 of 8 prompts, degraded on 1, and was unchanged on 6. The 122B cascade, by contrast, scored 10.4% more points than 35B standalone across 40 tasks in the blind-judge benchmark (170 vs. 154 points; win rate 12.5%, 5W–0L–35T). 35B self-review showed no measurable quality improvement over standalone generation (heuristic score 72.8 vs. 74.0), suggesting that self-correction requires a more capable reviewer. The 10.4% quality improvement from cross-model 122B review therefore justifies the swap overhead when operating in batched contexts.

D. Complexity Analysis: Where Does the Cascade Win?

A natural hypothesis is that the cascade provides the most benefit on complex tasks, where the 122B reviewer's deeper reasoning catches errors that the 35B cannot self-correct. We tested this hypothesis by classifying benchmark items into simple, medium, and complex categories and measuring the cascade's win rate against single 35B deployment.

TABLE V
Cascade Win Rate vs. 35B-Only by Task Complexity†

Complexity	Cascade Wins	vs. 35B alone
Simple	28.6%	Most decisive wins
Medium	20.0%	Moderate advantage
Complex	4.3%	Near parity

† Complexity tiers: $n = 3$ (simple), $n = 5$ (medium), $n = 22$ (complex). Win rates at $n < 10$ are anecdotal.

The hypothesis was not confirmed. The cascade is most decisive on simple tasks (28.6% win rate), not complex ones (4.3%). We hypothesize this is because the 35B's errors on simple tasks are more obvious to the 122B reviewer—clear bugs, missing edge cases, inconsistent style—while complex task errors require domain knowledge the 122B may also lack. When both models approach their respective capability ceilings on complex tasks, the 35B already produces work that closely matches what the 122B would have generated, leaving less for the reviewer to improve.

Critically, the cascade never loses to 122B alone within this dataset—it either ties or wins across all complexity bands. This observed no losses within this limited evaluation ($n < 30$ per tier); this should be treated as preliminary, not as a statistical guarantee. Substituting the cascade for the 122B single-model configuration produced no quality regression in this evaluation, only quality improvements or ties, at 3.6× the generation throughput.

VII. Evaluation

A. End-to-End Pipeline Timing

All swap times below use the measured values from Table III. Earlier drafts cited 25 s and 99 s swap times; those figures were initial estimates and have been replaced throughout with the measured 32.7 s and 42.7 s values. All SGLang measurements used version 0.5.9 (commit hash available in the reproduction package).

TABLE VI
Typical Cascade Cycle Timing (Measured Swap Times)

Phase	Duration
35B generation (2,000 tokens @ 184 tok/s)	∼11 s
Swap 35B → 122B	42.7 s
122B review (500 tokens @ 48 tok/s)	∼10 s
Swap 122B → 35B (if revising)	32.7 s
Total (single round, no revision)	∼64 s
Total (with one revision)	∼107 s

B. Throughput Comparison

TABLE VII
Throughput: Baseline vs. Cascade Pipeline

Configuration	Gen tok/s	Review tok/s	Speedup
Single-model (122B only)	38	38	1.0×
Cascade (35B gen + 122B review)	184	48	3.6–4.8×

The range of 3.6–4.8× reflects the amortization effect: more generation tokens per swap cycle yields higher effective speedup (approaching 4.8× = 184/38), while short generation sessions with frequent swaps reduce to 3.6× after accounting for swap overhead.

C. Larger Benchmark (Preliminary)

In a larger 30-task benchmark with blind LLM-as-judge evaluation, the cascade achieved second-highest points (180) behind Claude Opus (268), ahead of standalone 122B (148) and GPT-5.4[1] (73). However, the judging methodology (3-judge majority with swap-and-average) produced 95%+ ties due to high position bias in the judges, limiting the discriminative power of this evaluation. These results are preliminary and are reported for completeness only; the tie rate renders them insufficient to draw conclusions about relative quality ordering.

D. Structured Code Generation Benchmark

To supplement the heuristic quality ablation ($n = 8$), we ran a structured code generation benchmark on the 35B model. Because the evaluation environment does not have

[1]GPT-5.4 refers to OpenAI's March 2026 model accessed via the Chat Completions API.

a sandboxed execution harness for running generated code, we used structural verification rather than execution-based evaluation: each response is checked for the presence of the correct function definition name, a return statement, expected parameter names, and relevant logic keywords. This is not equivalent to execution-based benchmarks such as the official HumanEval [6] or MBPP; it measures structural adherence, not functional correctness.

1) Tasks: We defined 20 Python function generation tasks spanning standard algorithm and data structure topics: palindrome check, recursive factorial, list flattening, binary search, merge sort, word frequency counting, primality test, linked list reversal, two-sum, maximum subarray (Kadane's algorithm), valid parentheses, Fibonacci with memoization, matrix rotation, anagram grouping, longest common prefix, level-order BFS traversal, run-length decoding, power set enumeration, LRU cache implementation, and Dijkstra's shortest path. Each task specifies a concrete function signature and is verified against four structural checks.

2) Configurations Tested: We tested two configurations using the 35B-A3B generator (the only model running at evaluation time; the 122B was not swapped in):

1) 35B standalone: A single-pass generation with thinking disabled (enable_thinking: false).
2) 35B self-review: The 35B generates a response, then the same 35B model reviews its own output and produces a revised final answer—directly measuring the "self-review baseline absent" threat noted in §VIII without the swap overhead of the full cascade.

TABLE VIII
Structural Code Generation Benchmark (35B, $n = 20$ tasks)

Configuration	Pass Count	Pass Rate
35B standalone	19/20	95.0%
35B self-review	20/20	100.0%

3) Results: The single standalone failure (task he18, power set) occurred because the model generated a function named get_power_set instead of power_set as specified in the prompt—the logic, return, and parameter checks all passed. Self-review corrected this naming discrepancy. All other 19 tasks passed both configurations on the first attempt.

4) Interpretation: The 95.0%/100.0% pass rates should be interpreted cautiously: structural verification is a weak proxy for functional correctness. A response that defines the right function name and mentions heapq passes the Dijkstra check, regardless of whether the implementation is algorithmically correct. Execution-based evaluation on HumanEval ($n = 164$) or MBPP ($n = 374$) is needed to establish a rigorous correctness baseline.

Within the limits of structural checking, two observations are notable. First, the 35B achieves near-ceiling performance on standard coding tasks, consistent with the claim that generation quality is not the bottleneck in the cascade (the reviewer handles complex reasoning tasks, not routine code). Second, self-review provides a marginal improvement at zero swap cost, addressing the gap noted in the Threats section. Whether this self-review benefit holds on functionally-verified benchmarks is unknown.

VIII. Threats to Validity

We enumerate the principal threats to the validity of results in this paper.

Spec decode platform. The negative speculative decoding result was measured on SGLang 0.5.9 only. Cross-framework replication on vLLM has not been performed. The result could be SGLang-specific rather than a fundamental property of sparse MoE architectures.

Quality evaluation sample size. The quality ablation ($n = 8$, heuristic scoring) provides no statistical power to detect differences below ∼15 points. The 59.4 vs. 60.6 result cannot be taken as evidence of quality parity; it cannot be taken as evidence of a quality difference either.

Missing production metric. The 122B reviewer's REVISE rate was not tracked in production. The fraction of proposals that the reviewer flags for revision—and whether those revisions are genuine improvements—is unknown. This gap means the quality contribution of the review tier rests on the limited $n = 8$ ablation rather than on production data.

LoRA multiplexing not evaluated. LoRA-based multiplexing (LoRAX, S-LoRA, Punica) represents the most relevant unevaluated alternative to hot-swap. Both models are Qwen3.5 family, satisfying the shared-architecture requirement. If the task distribution is compatible with LoRA-scale adaptation, this approach could eliminate swap latency entirely.

Self-review baseline. A dedicated self-review experiment (8 prompts: 2 security, 2 code generation, 2 debugging, 2 architecture; results in /opt/rigrun/paper/-self_review_results.json) found that 35B self-review (35B reviews its own output) did not measurably improve over standalone generation (heuristic score 72.8 vs. 74.0, −1.6%; 1 improved, 1 degraded, 6 unchanged). The 122B cascade achieved 10.4% more points than 35B standalone in the 40-task blind-judge benchmark, a gap self-review does not close. The structural code generation benchmark (§VII-D) showed self-review raising structural pass rate from 95% to 100% (19/20 to 20/20), but this reflects a single naming error corrected, not a systematic quality improvement. Execution-based validation (HumanEval, MBPP) is required to confirm whether self-review benefits transfer to functional correctness.

Benchmark. No execution-based named benchmark (HumanEval, SWE-bench, MBPP) was used for functional correctness. The structured code generation benchmark (§VII-D) uses structural verification only—a much weaker

proxy. Results on internal suites with heuristic scoring functions limit external comparability.

Hardware specificity. Swap times are NVMe SSD-dependent at ∼3 GB/s. Results are from a single hardware configuration (RTX PRO 6000 Blackwell, 96 GB), single model family (Qwen3.5), single developer workload. Generalization to other hardware, model families, or workloads is unvalidated.

IX. Discussion

The key finding is the negative speculative decoding result. EAGLE-3 and NEXTN both degrade throughput on 3B-active MoE models. The mechanism—memory-bandwidth-bound decode makes speculation overhead unprofitable—is general and should inform practitioners evaluating speculative decoding on sparse architectures.

The cascade is engineering, not architecture. Hot-swap between models on a single GPU is a practical solution to a hardware constraint, not a novel architectural pattern. It works because code review requires substantially fewer tokens than code generation, so the throughput asymmetry favors the assignment of the fast model to the high-volume phase.

Complexity inversion deserves more analysis. The finding that cascade wins most decisively on simple tasks inverts the natural expectation. Our hypothesis—that simple-task errors are more obvious to the 122B reviewer than complex-task errors—should be tested with a larger evaluation set and a more principled complexity taxonomy.

Batched review is the correct operational pattern, but the threshold matters. At batch=30, swap overhead per project is negligible. At batch=1, the cascade is 1.4× slower per task than 122B-only. The break-even batch size for the cascade depends on the ratio of swap time to generation time.

Speculative decoding has a boundary condition. Our negative result with EAGLE-3 and NEXTN suggests that speculative decoding practitioners should evaluate whether their target model is compute-bound or memory-bandwidth-bound before investing in draft model training. For MoE models with active parameter counts below ∼5–10B, the memory bandwidth bottleneck likely dominates.

X. Conclusion

We reported a negative result: speculative decoding (EAGLE-3, NEXTN) degrades throughput on sparse MoE models with small active parameter counts (3B of 35B active). The failure mode is that memory-bandwidth-bound models receive no benefit from reduced compute forward passes while incurring draft generation and verification overhead. This result was measured on SGLang 0.5.9; vLLM replication is needed to confirm it is fundamental rather than platform-specific.

We described a hot-swap cascade as the engineering alternative adopted after ruling out speculative decoding: a 35B-A3B generator (184 tok/s) paired with a 122B-A10B reviewer (48 tok/s), connected by model swap on a single 96 GB GPU. Swap overhead is 32.7 s and 42.7 s per direction (measured). Standard batch amortization yields 2.5 s overhead per project at batch=30. A limited quality ablation ($n = 8$) suggests parity (59.4 vs. 60.6) but lacks statistical power. A structural code generation benchmark ($n = 20$) finds the 35B achieves 95% standalone and 100% with self-review on structural checks; execution-based validation (HumanEval, MBPP) remains future work. Critical remaining gaps include: no systematic tracking of the reviewer's REVISE rate and no comparison against LoRA multiplexing.

References

[1] L. Chen et al., "FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance," arXiv:2305.05176, 2023.

[2] "RouteLLM: Learning to Route LLMs with Preference Data," arXiv:2406.18665, 2024.

[3] "Dynamic Model Routing and Cascading for Efficient LLM Inference: A Survey," arXiv:2603.04445, 2026.

[4] L. Chen et al., "Punica: Multi-Tenant LoRA Serving," arXiv:2310.18547, 2023.

[5] T. Simonds, "Entropy Adaptive Decoding: Dynamic Model Switching for Efficient Inference," arXiv:2502.06833, 2025.

[6] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.

[7] Y. Li et al., "EAGLE-3: Scaling up Speculative Decoding against Reasoning Models," SafeAILab, arXiv:2503.01840, 2025.

[8] Z. Du et al., "SpecForge: Efficient Speculative Decoding via Jointly Training Draft and Verification," LMSYS, arXiv:2603.18567, 2025.

[9] Qwen Team, "Qwen3 Technical Report," Alibaba Cloud, 2025.