vSKIPPER: TRANSLATING DYNAMIC LAYER SKIPPING INTO LLM SERVING GAINS

Anonymous authors
Paper under double-blind review

ABSTRACT

Dynamic layer skipping reduces LLM computation by allowing each token to execute only a subset of the model's layers. However, existing skippers rely on specialized generation loops and do not integrate with modern serving engines. As a result, fewer executed layers do not necessarily translate into lower serving latency: FlexiDepth skips 8 of Llama-3-8B's 32 layers on average, yet its standard generation loop decodes 14.6–21.0% more slowly than the base model. We present vSKIPPER, a virtualization layer that makes dynamic layer skippers pluggable in serving engines while preserving continuous batching, fixed-shape batches, paged KV caching, and captured decode graphs. At each routed layer, vSKIPPER groups tokens by the skipper's decision and uses routed execution only when predicted to be profitable. We implement vSKIPPER in SGLang and evaluate the released FlexiDepth checkpoint against upstream SGLang under identical prompts, arrivals, output lengths, and launch settings. At the knee of upstream's load curve, vSKIPPER reduces mean end-to-end latency by 36.8% on GSM8K and 13.6% on BBH. Under saturation, it increases request throughput by 11.3% and 7.4%. Serving adds no statistically resolved quality loss beyond the checkpoint's own. Across synthetic skip policies, two Qwen3 skippers, and three GPUs, we demonstrate reuse without workload-specific tuning. To our knowledge, vSKIPPER is the first system to realize serving-efficiency gains from per-token interior layer skipping within a modern LLM serving engine. The code is open-sourced as an SGLang fork at `https://anonymous.4open.science/r/sglang-vskipper-7BCE`.

1 INTRODUCTION

Not every token requires the full depth of an LLM. Dynamic layer-skipping methods exploit this observation by choosing, for each token, which layers to execute (Luo et al., 2025; He et al., 2025; Yang et al., 2025). FlexiDepth (Luo et al., 2025), for example, adds lightweight per-token routers to a pretrained model and skips 20–30% of its layers. Such methods reduce model computation without replacing the underlying transformer backbone. However, translating saved computation into batched serving gains is a separate execution problem.

Modern LLM serving engines rely on regular batched execution, continuous batching, paged KV-cache management, and captured GPU graphs (Yu et al., 2022; Kwon et al., 2023; Zheng et al., 2024). These optimizations largely assume that tokens in a batch traverse every layer. Dynamic skipping breaks this regularity: at a given layer, some tokens execute the full computation while others bypass it. Released skippers therefore commonly use model-specific research execution paths and report FLOP savings (Luo et al., 2025), rather than gains in modern serving engines. The missing piece is an interface between layer skipping and the engine's batching, cache management, and execution.

We present vSKIPPER, a serving virtualization layer for per-token interior layer skipping. vSKIPPER makes a dynamic skipper a pluggable component of the serving engine through a reusable interface. The skipping policy decides where each token skips, while vSKIPPER preserves the engine's batching, cache, and graph interfaces. We realize mixed-depth execution through a RUN/*Project-Only* interface. At each routed layer, vSKIPPER packs rows sharing a decision into cohorts and scatters their outputs back to the original batch positions. Project-Only rows bypass the layer's attention and FFN while still writing the KV state required by future tokens' attention computation. Since routing introduces overhead, vSKIPPER performs *mode switching* only when its estimated savings exceed the cost.

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

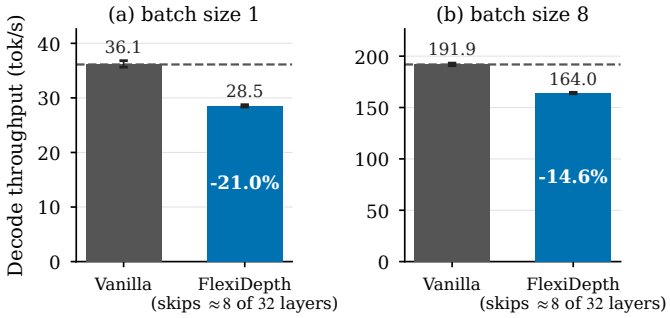


Figure 1: Saved FLOPs are not saved time: decode throughput of the FlexiDepth checkpoint against its base model in a standard generation loop, with fixed batches of 1 and 8 requests each generating 256 tokens (mean of three runs; whiskers show the range).

We implement vSKIPPER in SGLang and evaluate it against upstream SGLang using identical launch settings, prompts, arrivals and per-request output lengths. At the knee of upstream's load curve, mean end-to-end latency falls by 36.8% on GSM8K and 13.6% on BBH. Under saturation, request throughput rises by 11.3% and 7.4%. Serving introduces no statistically resolved quality loss beyond that of the checkpoint, and under load we find no resolved quality difference from routing every pass at the knee. To our knowledge, vSKIPPER is the first serving system to realize serving-efficiency gains from per-token interior whole-layer skipping within a modern LLM serving engine.

Our contributions are:

1. **A virtualization layer for skippers.** We introduce a whole-layer RUN/Project-Only interface that makes per-token skip policies pluggable components of a modern serving engine without redesigning its scheduler or memory manager (§3).
2. **Mixed-depth execution.** We develop route-guided cohort execution for heterogeneous per-token paths within captured graphs (§3.1), establish the invariants required for correct dynamic-depth execution (§3.2), and introduce profitability-aware mode switching (§3.3).
3. **A controlled evaluation.** We compare vSKIPPER with upstream SGLang under matched work and launch settings, and separate checkpoint quality from serving effects. We then demonstrate how the same implementation generalizes beyond FlexiDepth across nine synthetic skip policies, two Qwen3 skippers in addition to Llama-3-8B, three workloads, and A100, H100, and RTX A6000 GPUs, without workload-specific tuning (§4).

2 Background and Motivation

Serving engines assume fixed-depth execution. Continuous-batching engines such as Orca (Yu et al., 2022), vLLM (Kwon et al., 2023), and SGLang (Zheng et al., 2024) organize kernels, metadata, page tables, and captured graphs around the assumption that every row in a batch traverses every layer. Dynamic layer skipping breaks this assumption because tokens follow different layer paths.

Saved FLOPs are not saved time. FlexiDepth reports that its implementation "does not lead to improved throughput on the existing GPU hardware" (Luo et al., 2025). We reproduce this gap using the public checkpoint over 100 prompts with 256 greedy output tokens. Despite skipping 8 of 32 layers on average, decode throughput falls by 21.0% at batch size 1 and 14.6% at batch size 8 (Figure 1). At batch size 1, routing overhead offsets the saved work. At batch size 8, the batch still traverses the union of its tokens' required layers, while splitting it sacrifices batch width.

What a serving abstraction must provide. Turning dynamic depth into serving gains requires preserving batch efficiency, maintaining correct per-layer state across divergent paths, and routing only when the savings exceed the overhead. For autoregressive attention, even a row that skips a layer's main computation must still produce the KV state required by future tokens. The benefit also depends on phase: skipping reduces attention memory traffic during memory-bound decode and avoids arithmetic during compute-bound prefill.

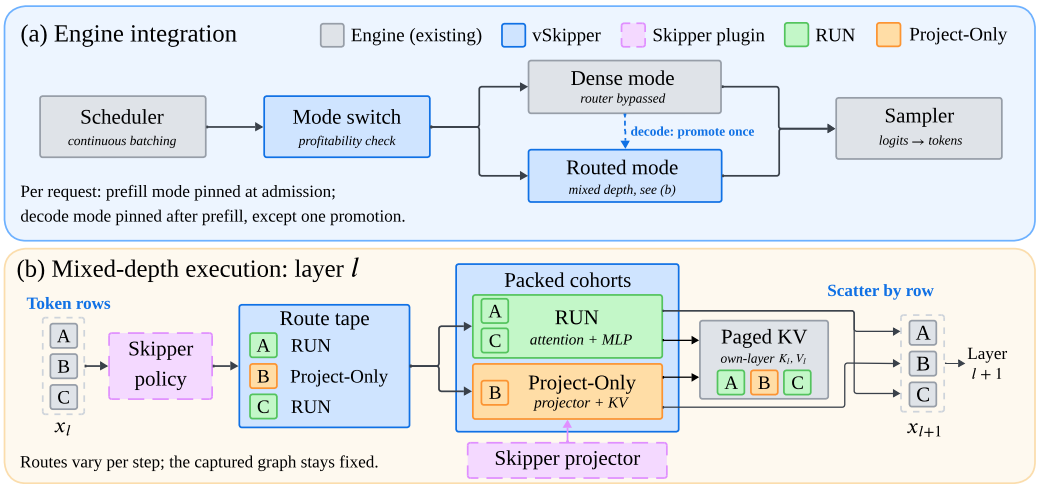


Figure 2: Overview of vSKIPPER. (a) The mode switch selects dense or routed execution per request and phase based on predicted profitability (§3.3). (b) Routed layers record skipper decisions, pack rows into RUN and Project-Only cohorts, and scatter outputs back to the batch (§3.1). Project-Only rows still write their own-layer KV (§3.2).

3 Design and Implementation

vSKIPPER separates the skipping policy from execution and engine state (Figure 2). A skipper implements the *skipper interface*: it supplies per-token RUN or Project-Only decisions and the projector applied to skipped rows. We realize these decisions through *mixed-depth execution*, which turns them into route data, packed cohorts, and KV-complete cache state (§3.1). Two invariants keep this execution correct under dynamic depth (§3.2), and the *mode switch* decides when routing is worth its cost (§3.3). The engine remains compatible with its serving optimizations, including continuous batching, admission control, paged KV management and prefix reuse (§3.4).

Vocabulary. A *row* holds one token's hidden state, one per sequence in decode and several per sequence in prefill. Each request uses *dense* or *routed* mode, chosen separately for prefill and decode. Dense mode is the unmodified engine's forward pass and runs every layer without consulting the router. Routed mode takes RUN or Project-Only per row at each routed layer. If a batch mixes modes, it uses the routed path with dense rows taking RUN. V denotes a batch's resident KV tokens.

3.1 Route-Guided Cohort Execution

Mixed-depth execution runs inside the engine's existing forward passes. Decode steps run as in a standard engine, replaying a single captured graph over the batch. Inside the graph, as Figure 2 shows, rows in the same batch can take different actions chosen by the policy from the current rows, and the RUN and Project-Only rows compute their updates separately before scattering them back to their rows. The next routed layer evaluates the updated states and may route differently.

Route tape. A layer's policy decisions are written into a device-resident *route tape*, a per-row mask for each routed layer. Since the captured graph consumes the tape as data, routes can change from step to step without host synchronization or graph recapture.

Project-Only path. A skipped token cannot simply bypass a layer: future tokens attending to it expect that layer's KV to exist (Invariant 1, §3.2). Skipped layers are therefore traversed cheaply, with the skipper's own skip semantics. A skipped row writes only the layer's key and value projections to the KV cache, dropping its attention output and running the plugged projector instead of the MLP. For FlexiDepth, the projector is the checkpoint's adapter, mixed by the router's weight $w \in [0, 1]$. The skip avoids the row's attention KV read and, when compute-bound, its bypassed arithmetic.

Packed cohorts. From the tape, each routed layer derives its two groups of rows, the *cohorts*, and their sizes as device scalars. We treat each cohort as one contiguous operand, folding the row gather

Algorithm 1 One routed layer of a decode graph captured at batch size C. The cohort counts n_r, n_p and router weights w live on the device, so the layer runs without host synchronization.

Require: the layer's route tape, hidden states X, residual Y, layer and projector weights, tile table
Ensure: Y updated for all rows and the layer's KV cache complete

1: **Maps.** From the route tape build the index maps $\mathrm{run}[0{:}n_r]$ and $\mathrm{proj}[0{:}n_p]$, mapping each cohort position to its batch row, and the counts n_r, n_p as device scalars.
2: **Attention.** Compute and write every valid row's own-layer KV through the layer's normal projection and cache-write path. The decode kernel takes the RUN mask: a Project-Only row reads no KV at this layer and emits zero.
3: **Tiles.** For each of the four count-bounded GEMMs (gate–up and down, per cohort) select the tile configuration from the device's tuned tile table with captured batch size C.
4: **RUN cohort.** $G \leftarrow \mathrm{GEMM}_{n_r}(X[\mathrm{run}], W_{\mathrm{gate,up}})$ with the gather folded into the operand load; $A \leftarrow \mathrm{silu}(G_{\mathrm{gate}}) \odot G_{\mathrm{up}}$ over n_r rows; $Y[\mathrm{run}[m]]$ += $w\,\mathrm{GEMM}_{n_r}(A, W_{\mathrm{down}})[m]$ with the scatter and the router weight w folded into the epilogue.
5: **Project-Only cohort.** The same three steps over proj and n_p, with the projector's weights and route weight $1 - w$.
6: **Grid exit.** Every GEMM launches a static grid sized for C rows, and a tile starting at or beyond the device-side cohort count exits immediately, so one captured graph serves every cohort size.

into the operand load and the weighted scatter of results into the epilogue, so the cohort GEMMs are bounded by cohort size rather than batch size. Attention instead keeps its launch shape and masks skipped rows' KV reads, leaving the graph capture-safe (Algorithm 1, Appendix B).

Captured execution. We capture both modes at fixed batch sizes, up to a device-derived maximum (e.g., $1{,}024$ rows on an A100), so the mode switch always finds a captured graph. Within the routed graph, each routed layer runs the mixed-cohort path of Algorithm 1, and a device-side conditional CUDA-graph node adds an all-RUN fast path for the frequent steps where no row skips the layer.

Prefill. The same interface and cohorts govern prefill, with one row per prompt token and its own switching rule (§3.3). Its variable-size inputs keep prefill outside capture, so cohorts use standard cuBLAS GEMMs with the same gather and scatter (Appendix B). In this compute-bound phase, a skipped row saves arithmetic directly and writes each layer's KV as in decode.

3.2 Invariants for Correct Mixed-Depth Execution

vSkipper enforces two invariants that preserve correctness when depth varies per token.

Invariant 1: every layer writes its own KV. *Every cached token has keys and values computed by each layer's own projection weights at all L layers, including the skipped layers.* Batched attention, paged KV caches (Kwon et al., 2023) and prefix reuse all assume that position t of layer ℓ's cache holds the outputs of that layer's own key and value projections, rather than copied or missing state. Early-exit methods can violate this and must recompute, keep per-depth caches (Elhoushi et al., 2024), or copy state across depths (Liu et al., 2025). The Project-Only path preserves it.

Invariant 2: row metadata is never stale. *Any structure that maps rows to memory is recomputed from the current row set wherever that set can change.* This covers attention metadata, KV page mappings and every gather or scatter map derived from routes. Under dynamic depth these maps change per layer and step, and reusing a stale map can make requests silently read other requests' KV pages. vSkipper therefore derives every route-dependent map on the device from the current tape.

3.3 Profitability-Aware Mode Switching

Routed mode has a fixed per-step cost, so it is *profitable* only when its savings exceed that cost. The mode switch is a host-side profitability check that selects dense or routed mode separately for prefill and decode. Prefill mode is fixed at admission, while decode mode is set after prefill and may be promoted once from dense to routed if load crosses the enter threshold, after which it never reverts. Because this promotion can change computation mid-generation, we evaluate the resulting *served hybrid* directly (§4.3). The *always-route* setting disables the switch.

Decode. Below the device's *ridge point*, where a kernel becomes compute-bound in the roofline model (Williams et al., 2009), decode is limited by weight bandwidth, and a skipped row saves only its KV read. Let L_r be the number of routed layers, s the fraction of decisions projecting, b the bytes of KV per token and layer, and τ routed mode's fixed per-step cost. Routing is profitable when

$$V \;>\; V^* = \frac{\tau \cdot \mathrm{BW}}{s\, L_r\, b} \;\approx\; 158\text{k resident tokens on an A100}$$

($\tau = 2.67$ ms, $s = 0.5$, $L_r = 16$, $b = 4$ KB; Appendix D). The served *switching thresholds* form a hysteresis band: routing is enabled at $V \geq 200$k and disabled for new decode requests at $V \leq 160$k. Within this range, the current switching state is retained. The rule reproduces the served exit threshold within 2%. Every other deployment's thresholds follow from the rule with its device and model constants (Appendix D).

Prefill. Prefill uses routed mode for batches with at least $1{,}536$ prompt tokens and an estimated Project-Only share above 0.35, jointly sufficient to cover the fixed cost. The share is probed every 64 admitted batches. These constants are fixed across workloads (Appendix D).

3.4 Implementation

We implement vSkipper as a fork of SGLang (Zheng et al., 2024), leaving unchanged its multiprocess architecture, admission control and memory management. The scheduler additionally records each request's mode, which also keys prefix reuse. Routed-layer hooks support the Llama-3 and Qwen3 families. The main skipper is the released FlexiDepth checkpoint for Llama-3-8B (Luo et al., 2025), routing the last 16 of 32 layers, and we train and serve two Qwen3 skippers (Appendix K).

Under the skipper interface, we implement policies of two kinds: a trained gate (FlexiDepth and the Qwen3 skippers) and a seeded hash of request and token (RandomSkip). A new policy can reuse an existing projector and its execution path; only a new projector requires its own loading and execution code and checkpoint tensors. At startup, the runtime rejects any policy whose actions fall outside the whole-layer $\{$RUN, Project-Only$\}$ set. Appendix A illustrates the interface with a static-depth policy.

4 Evaluation

We present the experimental setup (§4.1) and then evaluate vSkipper on three questions. **(Q1)** Does it turn a skipper's serving deficit into a gain at matched work (§4.2)? **(Q2)** Does it add any quality loss beyond the skipper's own (§4.3)? **(Q3)** Is the abstraction general across skippers, models, workloads and hardware (§4.4)?

4.1 Experimental Setup

Arms. We define an *arm* as one measured configuration and a *cell* as one (workload, offered rate) pair. The baseline arm is upstream SGLang, the code our fork derives from, served from its own checkout. The vSkipper arms are the served *hybrid* and always-route settings (§3.3). All arms share one launch profile (fp16 for Llama, bf16 for Qwen, static memory fraction 0.8, Triton attention), capture decode graphs at the same batch sizes, and keep SGLang's radix prefix cache enabled, with vSkipper maintaining mode-consistent prefix reuse (Appendix J). We check every arm's reported execution settings for undeclared differences (Appendix C).

Load. We set load through the offered arrival rate rather than admission control: seeded Poisson arrivals, no client-side concurrency cap, and every request served to completion. For each workload we measure Q^*, the last offered rate at which the baseline's output token rate still grows under its own natural generation (Appendix M). We call $0.95 \times Q^*$ the *knee*, $0.75 \times Q^*$ *below the knee* and $1.25 \times Q^*$ *overload*, evaluating arms at those absolute rates.

Workloads. We evaluate on three standard tasks with complementary serving profiles (Table 1): GSM8K mathematical reasoning (Cobbe et al., 2021), pairing long few-shot prompts with short outputs; BBH (BIG-Bench Hard) (Suzgun et al., 2022), whose chain-of-thought generations result in long outputs and whose shared few-shot prefixes are largely served from the prefix cache; and CoQA conversational QA (Reddy et al., 2019), whose long passages and very short answers (43% end within five tokens) concentrate decode work in a few long requests. We serve each workload as a

Table 1: The three workloads: serving protocol, requests per performance cell, and documents per quality cell. Output tokens and E2E are upstream's mean values at its knee.

Workload	protocol	requests	documents	output tokens	Q^* (req/s)	upstream E2E (s)
GSM8K	5-shot, chat template	3,600	1,319	116	13	26.9
BBH	3-shot chain-of-thought	4,000	6,511	255	34	23.9
CoQA	raw completion	4,000	500	164	25	13.1

frozen suite of token-id prompts, padding the GSM8K and CoQA load with training-split prompts and scoring only their held-out documents.

Matched work. The arms generate the same number of output tokens for each request and differ only in how they execute that work. Under natural stopping, the checkpoint's output lengths differ from the base model's (Appendix G), which would confound execution speed with generation behavior. We therefore record each request's output length from the baseline's natural generation at the cell's rate and replay it in both arms with `ignore_eos`. Prompts, arrival schedule, output lengths and launch profile are identical across arms, and a per-request gate rejects any cell whose token counts differ. This *matched-work lane* measures how fast each engine executes the same work. Its complement, the *natural lane*, lets each stack generate to its own end (Appendix G).

Testbed. Each measured pair runs on one NVIDIA A100-SXM4-80GB of a four-GPU node with no other tenant, and the two arms alternate order across repetitions. Hardware transfer to an H100 and an RTX A6000 follows the same protocol (§4.4, Appendix L), with machine details in Appendix M.

Metrics. Performance experiments use SGLang's standard serving benchmark module, `sglang.benchmark.serving`, to replay the same seeded Poisson arrival schedule for both arms and collect metrics, with no client-side concurrency cap. We report end-to-end latency (E2E), time to first token (TTFT), and time per output token (TPOT), where TPOT is a request's mean inter-token interval. All latency means and percentiles are over requests. At the cell level, *makespan* is the time from injection start until the final request completes, and *drain time* is the time from the end of the injection window until completion. Whole-cell tokens per second therefore depends on both the arrival schedule and drain, rather than service capacity alone. We instead measure saturated throughput as in FaaScale (Yu et al., 2026): output tokens (TPS) and completed requests (RPS) per second within the injection window at $1.25 \times Q^*$ (Appendix F). Below saturation, we compare latency at equal offered load.

Quality. Task quality is scored with `lm-eval` (Gao et al., 2024), a unified framework for evaluating language models, on each workload's held-out documents. For GSM8K, we use a composite of its two answer filters (Appendix E.2). To attribute quality changes between the checkpoint and our serving, we use a 2×2 design (Table 5, left): the base model and the checkpoint run natively in PyTorch (arms A and B) and are served on upstream and on vSKIPPER with always-route (arms C and D), all on the same documents. The paired difference-in-differences (DiD), $(D-C) - (B-A)$, with a 95% interval, isolates what serving adds beyond the checkpoint's own cost.

Statistics. We repeat each main cell six times, running both arms within every repetition, and report the mean of the per-repetition differences with a t-based 95% interval. A difference is *resolved* when its interval excludes zero, and *unresolved* otherwise, evidence of neither an effect nor equivalence. Experiments with fewer repetitions state n per table, and single-repetition serving cells, having no interval, are never called resolved.

4.2 SERVING COMPARISON (Q1)

We serve the released FlexiDepth checkpoint on Llama-3-8B, testing whether vSKIPPER turns its native serving deficit (§2) into a gain at matched work. Table 2 reports the percentage change of each metric against upstream, per workload and load point. Mean end-to-end latency is lower on every row, with the 95% interval excluding zero on eight of nine: -4.6 to -8.0% below the knee, -36.8% at the GSM8K knee, and -13.6% at the BBH knee.

The gains come from different phases, which Table 3 isolates by routing one phase at a time. GSM8K's gain comes from prefill routing, which alone gives -35.6% at the knee (n=1; Appendix H). Shorter prefill passes also reduce concurrent decode rows' inter-token time because SGLang runs prefill and

Table 2: Matched-work serving performance of vSKIPPER against upstream SGLang: mean percentage change with 95% intervals over six paired repetitions per row. Negative is better. Makespan changes only through the drain after the shared injection window.

	E2E		TPOT		TTFT		
Rate	mean	p99	mean	p99	mean	p99	makespan
GSM8K							
$0.75 \times Q^*$	-6.6 ± 1.5	-9.7 ± 2.8	-6.8 ± 1.6	-10.0 ± 3.7	-7.1 ± 1.2	-4.4 ± 4.8	-0.5 ± 0.1
$0.95 \times Q^*$	-36.8 ± 2.0	-37.3 ± 2.5	-35.6 ± 1.8	-15.3 ± 3.8	-66.0 ± 6.8	-74.5 ± 3.0	-1.4 ± 0.1
$1.25 \times Q^*$	-25.7 ± 0.7	-21.2 ± 0.6	-10.5 ± 0.2	-18.9 ± 1.7	-41.5 ± 1.1	-40.7 ± 0.9	-7.2 ± 0.2
BBH							
$0.75 \times Q^*$	-4.6 ± 0.5	-8.2 ± 1.0	-4.4 ± 0.6	-10.7 ± 2.7	+4.4 ± 17.7	+12.0 ± 37.5	-4.7 ± 0.5
$0.95 \times Q^*$	-13.6 ± 1.2	-11.4 ± 0.3	-14.6 ± 1.4	-22.4 ± 1.5	-4.9 ± 8.7	+6.8 ± 31.3	-7.1 ± 0.2
$1.25 \times Q^*$	-8.5 ± 1.0	-8.9 ± 0.9	-8.8 ± 0.7	-10.4 ± 2.0	-5.6 ± 14.6	-16.4 ± 7.1	-6.3 ± 0.7
CoQA							
$0.75 \times Q^*$	-8.0 ± 0.6	-10.1 ± 0.4	+1.1 ± 5.9	-2.0 ± 18.4	+1.6 ± 6.8	+5.4 ± 30.0	-7.8 ± 0.3
$0.95 \times Q^*$	-4.0 ± 4.4	-9.2 ± 0.5	+1.0 ± 15.1	+1.0 ± 24.0	-1.2 ± 5.9	+0.8 ± 5.2	-7.0 ± 0.4
$1.25 \times Q^*$	-2.7 ± 0.2	-9.1 ± 0.2	-0.9 ± 0.2	-1.7 ± 0.6	+0.1 ± 0.6	+0.3 ± 0.7	-7.3 ± 0.1

Table 3: GSM8K mechanism ablation: mean E2E and makespan change (%) against upstream. The served arm uses n=6 repetitions and each ablation arm n=1. All workloads appear in Appendix H.

	served (n=6)		decode only		prefill only		always route		no promotion	
rate	E2E	makespan	E2E	makespan	E2E	makespan	E2E	makespan	E2E	makespan
0.75×	-6.6 ± 1.5	-0.5 ± 0.1	+0.4	+0.0	-8.6	-0.7	-5.5	-1.0	-6.7	-0.5
0.95×	-36.8 ± 2.0	-1.4 ± 0.1	-14.3	-0.9	-35.6	-1.2	-46.5	-1.9	-32.6	-1.2
1.25×	-25.7 ± 0.7	-7.2 ± 0.2	-7.8	-2.6	-19.9	-5.4	-26.3	-7.4	-24.5	-7.0

decode as separate passes on one stream. Decode routing drives BBH, matching vSKIPPER at the knee (-16.0%), with prefill contributing nothing, and produces CoQA's smaller gain. Always-route improves further at the GSM8K knee but becomes slower than upstream at the CoQA knee ($+5.5\%$), while disabling promotion reduces the GSM8K knee gain to -32.6%.

Near saturation, queueing amplifies execution savings into large latency gains (Appendix F). At the same offered load, requests wait less: time to first token falls by 66.0% at the GSM8K knee, where shorter prefill passes clear the admission queue. Under saturation at $1.25 \times Q^*$, vSKIPPER raises RPS by 11.3% on GSM8K and 7.4% on BBH. CoQA, bound by prefill that vSKIPPER tends to avoid routing, shows no resolved change (Table 4).

CoQA is tail-heavy: 2.0% of the base model's generations run past $4{,}096$ tokens and carry 94% of a cell's output tokens. vSKIPPER finishes this tail faster, improving makespan by 7.0–7.8% at every rate; the gain sits in the long requests (Appendix I).

At matched work, vSKIPPER thus converts FlexiDepth's serving deficit into lower mean and p99 end-to-end latency across the grid, largest at and beyond the knee on GSM8K and BBH.

4.3 TASK QUALITY (Q2)

Serving adds no resolved quality loss beyond the checkpoint's own. Table 5 (left) reports the 2×2: the DiD is -0.23 ± 0.68 pp on GSM8K, $+0.45 \pm 0.55$ pp on BBH and $+0.07 \pm 0.12$ pp on CoQA, each interval containing zero. The checkpoint's own change, $B-A$, is the skipper's quality–compute trade-off. The same serving interface can therefore support checkpoints with different trade-offs.

Table 5 (right) evaluates the served hybrid under load at the knee. It shows no resolved difference from always-route: -0.08 ± 1.46 pp on GSM8K, -0.22 ± 0.38 pp on BBH, and -0.05 ± 1.76 pp on CoQA. Intervals are computed per document with n=1, and other load rates appear in Appendix E.3.

4.4 GENERALITY (Q3)

The framework generalizes across skippers, model families, workloads, and hardware. New skippers, models, and devices reuse the same interface and execution framework without scheduler or memory-manager redesign or per-workload tuning, aside from architecture-specific projections.

Table 4: Throughput under saturation at $1.25 \times Q^*$: mean change (%) with 95% intervals against upstream in TPS and RPS inside the injection window; positive is better. GSM8K, BBH, CoQA: Llama-3-8B on the A100 (n=6); other columns: GSM8K (n=3).

	GSM8K	BBH	CoQA	Qwen3-4B	Qwen3-8B	H100	RTX A6000
TPS	$+10.4 \pm 0.3$	$+4.1 \pm 0.4$	-1.0 ± 1.6	-0.2 ± 0.3	$+2.5 \pm 0.2$	$+18.5 \pm 3.0$	$+9.1 \pm 2.1$
RPS	$+11.3 \pm 0.2$	$+7.4 \pm 0.3$	-1.0 ± 1.7	-0.2 ± 0.4	$+2.8 \pm 0.1$	$+19.3 \pm 3.4$	$+9.3 \pm 1.7$

Table 5: Task quality (%; GSM8K composite exact match, BBH exact match, CoQA F1). Left: base (A) and checkpoint (B) in PyTorch, served on upstream (C) and always-route vSKIPPER (D), with the DiD. Right: served at the knee; routed: share of decode rows, hybrid / always-route.

	PyTorch versus served (`lm-eval`)							Served at the knee			
Task	A	B	C	D	$(B-A)$	$(D-C)$	DiD (pp)	upstream	hybrid	always-route	routed (%)
GSM8K	77.41	74.15	77.79	74.30	-3.26	-3.49	-0.23 $\pm$ 0.68	77.63	74.00	74.07	90 / 100
CoQA	78.31	78.99	78.31	79.07	+0.68	+0.75	+0.07 $\pm$ 0.12	76.74	77.68	77.73	71 / 100
BBH	68.04	59.65	67.93	59.99	-8.39	-7.94	+0.45 $\pm$ 0.55	67.87	59.47	59.68	99 / 100

Skippers. Alongside FlexiDepth, we define a deterministic mock skipper (RandomSkip) that implements the interface by routing a fixed fraction r of rows around a fixed fraction d of routed layers, giving skip ratio $s = r\,d$. Figure 3 maps the nine arms' mean E2E change under three switch settings: always route (a), FlexiDepth's thresholds for every arm (b), and rule-derived thresholds per policy (c). Without the switch, unprofitable arms cost up to $+23.8\%$. With shared thresholds, routing becomes profitable between $s = \frac{3}{16}$ and $s = \frac{1}{4}$, where the rule places the crossover (Appendix D), and gains reach -49.1% at r=d=75%. Per-policy thresholds keep unprofitable arms dense while retaining gains for profitable ones.

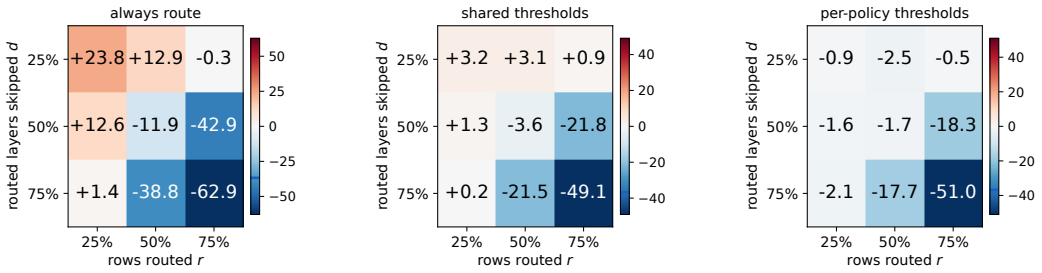
Models. We train Qwen3 skippers with only the first of FlexiDepth's two training stages (Appendix K). They use the same RUN/Project-Only interface and execution path as FlexiDepth on Llama-3-8B. The only model-specific change is a hook that applies Qwen3's per-head query and key normalization when a skipped layer writes its KV, as Qwen3's own attention does; the switching thresholds follow from the rule in §3.3. Table 6 pairs latency and quality at each model's knee. Qwen3-4B, whose always-route decode skip of 0.119 is too low for routing to be profitable at these loads, shows no resolved change in mean E2E at any load ($+2.7 \pm 2.8\%$ at the knee). Qwen3-8B, at skip 0.416, lowers mean E2E by $6.6 \pm 0.3\%$ with no resolved quality loss beyond the checkpoint's own (Appendix K). Across three Qwen3-8B checkpoints, serving gains grow with the skip rate, and along the lower-penalty run, continued training raised skipping and accuracy together, suggesting skipper training can further improve this quality–efficiency trade-off.

Hardware. We run the same implementation on an H100 and an RTX A6000 without workload-specific tuning. Each device uses rule-derived thresholds and its own knee, while only device-specific kernel tiles and, on the A6000, the decode-graph range change (Table 7). At the knees, mean E2E falls by $39.4 \pm 0.2\%$ on the H100 and $30.0 \pm 4.2\%$ on the A6000 (GSM8K, n=3; Appendix L).

5 Related Work

Layer skipping. Prior work reduces transformer depth by selecting layers per token (Luo et al., 2025; Yang et al., 2025), skipping sublayers (He et al., 2025), using fixed or temporally scheduled layer subsets (Liu et al., 2024; Rajabzadeh et al., 2026), exiting early (Schuster et al., 2022; Elhoushi et al., 2024; Chen et al., 2024), or removing layers (Men et al., 2025). Other methods constrain the policy to simplify execution (Corro et al., 2023; Raposo et al., 2024). These works primarily decide *what* to skip, and released FlexiDepth, AdaSkip, and LoRA-Drop implementations use specialized PyTorch paths. vSKIPPER instead enables layer skipping in modern serving engines by providing a RUN/Project-Only interface.

Serving dynamic computation. Modern engines (Yu et al., 2022; Kwon et al., 2023; Zheng et al., 2024; Agrawal et al., 2024) schedule at token granularity but execute each batch through the full model. DREX (Liu et al., 2025) and Apparate (Dai et al., 2024) serve early exits, where each decision removes a suffix of layers; DREX rebatches by depth and transfers state across depths. Multi-adapter



(a) always route (b) shared thresholds (c) per-policy thresholds

Figure 3: Mean E2E change (%) against upstream for the nine RandomSkip arms at the GSM8K knee (n=1), by rows routed r and routed layers skipped d, under the three switch settings (a–c).

Table 6: Cross-model comparison on GSM8K at each model's knee. Matched-work mean E2E uses n=6 for Llama and n=3 for Qwen. Served accuracy uses one repetition. Skip is the always-route Project-Only decode share.

		mean E2E (s)			accuracy (%)		
Model	skip	upstream	vSKIPPER	change (%)	upstream	vSKIPPER	change (pp)
Llama-3-8B (FlexiDepth)	0.504	26.9	17.0	-36.8 ± 2.0	77.6	74.0	-3.64 ± 2.22
Qwen3-4B	0.119	26.7	27.4	$+2.7 \pm 2.8$	80.7	77.7	-2.96 ± 2.15
Qwen3-8B	0.416	71.6	66.8	-6.6 ± 0.3	80.4	75.9	-4.47 ± 2.33

Table 7: Hardware transfer: the GSM8K matched-work comparison on an H100 ($Q^* = 14$ req/s) and an RTX A6000 ($Q^* = 6$ req/s), as mean percentage change with 95% intervals over three paired repetitions against upstream on the same device.

			E2E		TPOT		TTFT		
Device	Rate	n	mean	p99	mean	p99	mean	p99	makespan
H100	$0.75 \times Q^*$	3	-6.0 ± 3.2	-8.8 ± 5.2	-6.3 ± 3.5	-9.8 ± 9.1	-5.2 ± 0.2	-9.8 ± 6.4	-0.5 ± 0.3
H100	$0.95 \times Q^*$	3	-39.4 ± 0.2	-34.2 ± 9.6	-31.4 ± 7.6	-11.5 ± 7.9	-91.7 ± 4.8	-93.0 ± 2.2	-2.3 ± 2.2
H100	$1.25 \times Q^*$	3	-35.8 ± 3.4	-32.1 ± 3.0	-15.7 ± 1.5	-24.4 ± 9.3	-51.8 ± 3.8	-50.8 ± 2.4	-9.8 ± 1.1
RTX A6000	$0.75 \times Q^*$	3	-1.4 ± 2.3	-7.3 ± 2.4	-1.4 ± 2.5	-9.3 ± 2.5	-8.9 ± 2.3	-16.4 ± 8.9	$+0.3 \pm 0.7$
RTX A6000	$0.95 \times Q^*$	3	-30.0 ± 4.2	-27.7 ± 4.6	-23.0 ± 5.0	-12.0 ± 6.0	-81.1 ± 2.8	-62.9 ± 4.7	-2.2 ± 0.2
RTX A6000	$1.25 \times Q^*$	3	-27.1 ± 4.2	-26.6 ± 4.4	-9.9 ± 1.3	-15.8 ± 5.5	-34.8 ± 5.5	-33.3 ± 5.4	-6.7 ± 1.1

LoRA serving (Hu et al., 2022; Chen et al., 2023; Sheng et al., 2024) separate customized model adapter from batched execution, but retain fixed layer traversal. MoE systems (Shazeer et al., 2017) route tokens across experts within a layer, whereas vSKIPPER routes across depth while keeping the batch and captured graph fixed and preserving per-layer KV state.

6 Conclusion and Future Work

We present vSKIPPER, a serving virtualization layer that translates per-token interior layer skipping into gains in a modern LLM serving engine. Its RUN/Project-Only interface, mixed-depth cohort execution, and profitability-aware switching preserve the batching, cache, and graph optimizations of SGLang. vSKIPPER converts FlexiDepth's saved computation into lower latency and higher saturated throughput without statistically resolved serving-induced quality loss, and the framework extends across additional policies, Qwen3 skippers, workloads, and three GPUs without workload-specific tuning. Three directions could further broaden vSKIPPER's scope. (1) Extend execution support across additional serving engines and attention backends. (2) Generalize the interface to sublayer skippers such as AdaSkip and more model families. (3) Support distributed serving. Separate prefill and decode modes naturally fit disaggregation (Zhong et al., 2024), while tensor and pipeline parallelism require routing-aware execution across devices. Pipeline parallelism is particularly promising: assigning stages using profiled skip rates could balance dynamic work and translate layer skipping into pipeline-level gains.

REPRODUCIBILITY STATEMENT

The main code repository is available anonymously at `https://anonymous.4open.science/r/sglang-vskipper-7BCE/README.md`: the VSKIPPER implementation over SGLang, the training scripts for the Qwen3 skippers, and the experiment scripts that reproduce the results, from serving cells to measurement-derived tables, figures and numerical summaries. The two Qwen3 skippers we trained are released at `https://anonymous-hf.com/a/afdrdkb4ckoj/`.

The supplementary data pack contains frozen analysis inputs, per-document quality records, analysis source and reference outputs. Its results-only entrypoint regenerates and checks the numerical results without a GPU or manuscript source. The architecture schematic is an explanatory illustration.

GPU reproduction uses the repository scripts with the model, dataset and checkpoint assets specified in Appendix M and the code guides. The serving cells take about 75 GPU-hours across the three devices, under USD 100 at the recorded prices. Each Qwen3 skipper trains in 40 to 60 A100-hours. Appendix M gives the revisions, launch profile, Q^* sweeps, training setup and cost breakdown. Appendix C lists the checks each cell passed; Appendix A gives a policy-plugin example.

AI DISCLOSURE

In this work, we used generative AI tools to implement methods according to the authors' design, with the code reviewed by the authors, to interpret results, and to design or provide feedback on the research methodology or experiments. We have not used generative AI tools to develop theoretical models or conceptual frameworks, to propose or refine hypotheses, to generate synthetic datasets, to clean or reformat datasets, or to assist with translation, and formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, and qualitative or thematic data analysis are not applicable to this work. Additionally, we used generative AI tools to create and edit software code, to tabulate and visualize measured results, to build the reproduction artifact, to identify relevant literature and format references, and to draft parts of the paper and edit it for readability. We have reviewed all AI-assisted work. The authors designed the system, the switching rule and the evaluation protocol, and made every design, experiment and methodology decision. AI-drafted code was also verified on GPU by the correctness and execution checks of Appendix C and by the paired quality comparison of § 4.3. Every number in the paper is generated from the measured results by the artifact's scripts, which reproduce all 124 reference outputs. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023. URL `https://arxiv.org/abs/2308.16369`.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3.

Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving, 2023. URL `https://arxiv.org/abs/2310.18547`.

Yanxi Chen, Xuchen Pan, Yaliang Li, Bolin Ding, and Jingren Zhou. Ee-llm: large-scale training and inference of early-exit large language models with 3d parallelism. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL `https://arxiv.org/abs/2110.14168`.

Luciano Del Corro, Allie Del Giorno, Sahaj Agarwal, Bin Yu, Ahmed Awadallah, and Subhabrata Mukherjee. Skipdecode: Autoregressive skip decoding with batching and caching for efficient llm inference, 2023. URL https://arxiv.org/abs/2307.02628.

Yinwei Dai, Rui Pan, Anand Iyer, Kai Li, and Ravi Netravali. Apparate: Rethinking early exits to tame latency-throughput tensions in ml serving. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, pp. 607–623, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi: 10.1145/3694715.3695963. URL https://doi.org/10.1145/3694715.3695963.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, Ahmed Aly, Beidi Chen, and Carole-Jean Wu. LayerSkip: Enabling early exit inference and self-speculative decoding. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12622–12642, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.681. URL https://aclanthology.org/2024.acl-long.681/.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL https://zenodo.org/records/12608602.

Zhuomin He, Yizhen Yao, Pengfei Zuo, Bin Gao, Qinya Li, Zhenzhe Zheng, and Fan Wu. Adaskip: adaptive sublayer skipping for accelerating long-context llm inference. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'25/IAAI'25/EAAI'25. AAAI Press, 2025. ISBN 978-1-57735-897-8. doi: 10.1609/aaai.v39i22.34579. URL https://doi.org/10.1609/aaai.v39i22.34579.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Xuting Liu, Daniel Alexander, Siva Kesava Reddy Kakarla, Behnaz Arzani, and Vincent Liu. Dynamic rebatching for efficient early-exit inference with drex, 2025. URL https://arxiv.org/abs/2512.15705.

Yijin Liu, Fandong Meng, and Jie Zhou. Accelerating inference in large language models with a unified layer skipping strategy, 2024. URL https://arxiv.org/abs/2404.06954.

Xuan Luo, Weizhi Wang, and Xifeng Yan. Adaptive layer-skipping in pre-trained LLMs. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Gu0XSax2YS.

Xin Men, Mingyu Xu, Qingyu Zhang, Qianhao Yuan, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. ShortGPT: Layers in large language models are more redundant than you expect. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 20192–20204, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.1035. URL https://aclanthology.org/2025.findings-acl.1035/.

Hossein Rajabzadeh, Maryam Dialameh, Chul B. Park, Il-Min Kim, and Hyock Ju Kwon. Lora-drop: Temporal lora decoding for efficient llm inference, 2026. URL https://arxiv.org/abs/2601.02569.

David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. Mixture-of-depths: Dynamically allocating compute in transformer-based language models, 2024. URL https://arxiv.org/abs/2404.02258.

Siva Reddy, Danqi Chen, and Christopher D. Manning. CoQA: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics*, 7:249–266, 2019. doi: 10.1162/tacl_a_00266. URL https://aclanthology.org/Q19-1016/.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Q. Tran, Yi Tay, and Donald Metzler. Confident adaptive language modeling. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. URL https://arxiv.org/abs/1701.06538.

Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-lora: Serving thousands of concurrent lora adapters, 2024. URL https://arxiv.org/abs/2311.03285.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. Challenging big-bench tasks and whether chain-of-thought can solve them, 2022. URL https://arxiv.org/abs/2210.09261.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Ning Yang, Fangxin Liu, Junjie Wang, Tao Yang, Kan Liu, Haibing Guan, and Li Jiang. Dash: Input-aware dynamic layer skipping for efficient llm inference with markov decision policies, 2025. URL https://arxiv.org/abs/2505.17420.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Zirui Wang, Yue Cheng, Wei Wang, Ao Wang, and Ruichuan Chen. Faascale: Unlocking fast LLM scaling for serverless inference. In *Ninth Conference on Machine Learning and Systems*, 2026. URL https://openreview.net/forum?id=jgL8LuOVyT.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3.

A A MINIMAL SKIPPER PLUGIN

A policy plugin returns one action per row without executing either branch. The projector specifies the skip-path computation, while the runtime also writes the layer's own KV (Invariant 1). The example selects a static layer subset, following the unified layer-skipping strategy (Liu et al., 2024), and reuses the released checkpoint's projector. Place the definition in the skipper module, which already imports the adapter types, registration function and PyTorch.

```
class StaticDepthSkipper(FullGraphSkipperAdapter):
   name = "static_depth"
   supported_actions = frozenset((LogicalAction.RUN,
                           LogicalAction.PROJECT_ONLY))
   requires_router_weights = False
   projector_kind = FLEXIDEPTH_PROJECTOR
   payload_semantics = "binary_static_depth"

   def __init__(self, ratio):
      self.ratio = float(ratio)
      if not 0.0 <= self.ratio <= 1.0:
         raise ValueError("ratio must be in [0, 1]")

   def prepare_batch(self, *, hidden_states, valid_rows,
                     route_layer_order, **_):
      # Choose a tail within the engine's existing routed set.
      take = int(len(route_layer_order) * self.ratio)
      layers = frozenset(
         route_layer_order[len(route_layer_order) - take:])
      project_mask = (~valid_rows).view(-1, 1)
      run_mask = torch.ones_like(project_mask)
      # Preallocate device payloads; padding rows always RUN.
      dtype = hidden_states.dtype
      return (layers, run_mask, run_mask.to(dtype),
            project_mask, project_mask.to(dtype))

   def route(self, hidden_states, *, router, forced_action=None,
           layer_id=None, batch_state=None):
      if forced_action is not None:
         raise ValueError("forced routes are unsupported")
      if layer_id is None or batch_state is None:
         raise ValueError("layer and prepared batch required")
      layers, run, run_w, project, project_w = batch_state
      mask, weights = ((project, project_w) if layer_id in layers
                  else (run, run_w))
      return FullGraphActionBatch(
         adapter_name=self.name,
         supported_actions=self.supported_actions,
         branch_weights=weights, explicit_run_mask=mask,
         payload_semantics=self.payload_semantics)

   def attestation(self):
      return {"name": self.name, "decision": "static_tail_fraction",
            "static_depth_ratio": self.ratio,
            "requires_router_weights": self.requires_router_weights,
            "projector_kind": self.projector_kind,
            "supported_actions": ["run", "project_only"]}

def build_static_depth(arm):
   return StaticDepthSkipper(arm["static_depth_ratio"])

register_skipper("static_depth", build_static_depth)
```

Register the serving configuration in the design module:

```
ARMS["integrated_staticdepth"] = {
   "skipper": "static_depth", "phases": "both",
   "regime_switch": True, "static_depth_ratio": 0.5,
}
```

The explicit mask and branch weights have shape `[rows, 1]`; no threshold or forced action accompanies the mask. At ratio zero every routed layer runs; at ratio one every valid row projects throughout the routed set.

The example reuses FlexiDepth's projector execution. To integrate a different projector, implement its model-loading and executor paths, register its metadata and checkpoint tensors, and select it from the policy.

B Route-Guided Cohort Execution at the Kernel Level

Algorithm 1 gives one routed decode layer. There are B resident rows in a graph captured for batch size $C \geq B$; C is the graph's capacity. The released router supplies a weight $w_i \in [0, 1]$ and selects RUN when $w_i > 0.5$; Qwen uses a straight-through hard gate. Exact device-side counts determine the cohort sizes, without a capacity fraction or host synchronization.

Prefill uses the same pack and scatter with cuBLAS over the packed rows. Captured decode uses count-bounded Triton GEMMs with a device-tuned tile table (Appendix L). Their accumulation order differs from cuBLAS; Table 5 evaluates end-task quality through the routed execution path.

C Validation Gates

The quality comparison. For each task the paired per-document difference-in-differences $(D - C) - (B - A)$ has a two-sided 95% t interval, reported as is.

Attestation. The runtime *attests* what actually executed through its info endpoint: route and cohort counters, passes per mode, and every switch decision. The gates below read this attestation rather than the launch configuration, so a misconfigured mechanism cannot pass silently.

Serving gates. Each is enforced in code and can refuse a cell. Grouped by what they protect:

- *Identity of what is served.* Served design: the design read from the info endpoint after boot equals the intended arm's, field by field.
- *Comparability of the two arms.* Execution difference: the two servers' reported execution (graph capture sizes, KV capacity, chunk size, backends) matches a declared configuration. Cross-arm identity: resolved configurations may differ only in an allow-listed set. This covers the runtime's own settings and the KV budget consumed by its weights: $3{,}840$ tokens, or 480 MB at 128 KB per Llama-3-8B token. That is 0.98% of the A100's $393{,}319$-token pool and 2.07% of the RTX A6000's $185{,}766$. The gate refuses a larger KV pool on the served arm.
- *Validity of a cell.* Work identity: both arms generated the same number of tokens per request, from raw server-reported output ids. Accounting: submitted = started = completed = reported, zero errors, no filtered tail. Declared cell size: the exact suite, duration derived as requests over rate. Zero-empty: no empty generation in a natural-lane or quality cell, except the cases of Appendix E. Protocol identity: served arms receive the same token ids as the PyTorch arms.
- *Mechanism evidence.* Skipping executed: the attestation read before and after the cell must show routed passes on the phases the arm declares, with per-mode prefill counters summing to the scheduler's pass count. A cell whose declared mechanism never executed is refused.

Every arm is a named entry in one module, and an undeclared difference in the arms' executed settings fails the cell.

D The Switching Thresholds from the Roofline

Prefill floor. A coarse sweep over 1k–3k-token prefill passes put routed mode's crossover near 1.5k tokens, so $1{,}536$ was fixed as the floor; it is applied to an admission round's prompt tokens, before prefix-cache hits. Like a chunked-prefill size (Agrawal et al., 2023), this setting is applied unchanged to every workload, rate, arm and device.

Decode rule. For a weight-stationary decode GEMM over M rows with fp16 weights, FLOPs are $2MKN$ and weight bytes $2KN$. The arithmetic intensity, the FLOPs per byte moved, is therefore M. The device's ridge point becomes a row count, $M^* =$ peak FLOP/s$/$bandwidth ≈ 161 on the A100. Below M^* the weights are streamed whether or not a row skips. What a skipped row saves at a routed layer is that layer's attention read of its own context, $b = 4$ KB per resident token for Llama-3-8B. Over $L_r = 16$ routed layers with a fraction s of decisions projecting, a step holding V resident tokens removes $s\,L_r\,b\,V$ bytes, i.e. $s\,L_r\,b\,V/\text{BW}$ of time, against routed mode's fixed per-step cost $\tau = 2.67$ ms. This cost is the measured intercept difference of the two modes' per-step latency lines on the A100's 16 routed layers. It is calibrated once and reused as a per-routed-layer coefficient on every device and model (scaled by $L_r/16$ for Qwen); only the device bandwidth and

the model and policy constants change. The H100, A6000 and Qwen results therefore test the rule's transfer without per-device retuning. Routed mode is profitable when

$$V \;>\; V^* = \frac{\tau \cdot \mathrm{BW}}{s\, L_r\, b} \;=\; \frac{2.67\,\mathrm{ms} \times 1.94\,\mathrm{TB/s}}{0.5 \times 16 \times 4\,\mathrm{KB}} \;\approx\; 158\mathrm{k\ resident\ tokens},$$

with the checkpoint's declared decode skip ratio $s = 0.5$ (the always-route arm attests 0.504 at the GSM8K knee). The rule estimates the break-even point using the unweighted decision ratio as a proxy for its context-weighted counterpart. Above M^*, skipped rows also remove compute, making the estimate conservative.

The served thresholds and the rule. The A100 thresholds (exit 160k, enter 200k) were measured with a two-mode row sweep at 1k context. Routed mode ran 7% slower per step at 128 rows (131k tokens) and 4% faster at 256 rows (262k). Linear interpolation places the crossover near 214k; the enter threshold was set just below it and the exit 20% lower. The rule gives 158k, within 2% of the exit. Rows alone would misplace the thresholds, since the saving scales with rows $\times$ context. For another device or model, the rule gives predicted thresholds: exit at V^*, enter at $1.25\,V^*$. This sets the H100, A6000 and Qwen thresholds (Appendices L and K). From when V crosses the enter threshold until it falls to the exit threshold, requests that start decoding are pinned to routed mode, and the crossing promotes the dense-decoding requests already running.

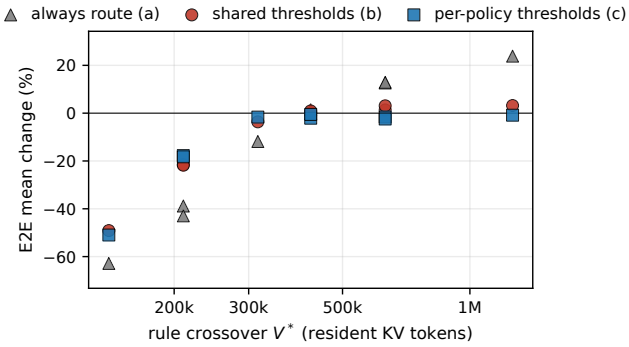


Figure 4: The rule's crossover V^* per RandomSkip arm against its measured mean E2E change under the three switch settings of Figure 3.

The two rules on this grid. The grid's skip ratios are $s = r\,d \in \{\frac{1}{16}, \frac{1}{8}, \frac{3}{16}, \frac{1}{4}, \frac{3}{8}, \frac{9}{16}\}$. At upstream's estimated knee occupancy of 411k tokens the decode rule splits the grid between $s = \frac{3}{16}$ and $s = \frac{1}{4}$; the prefill share threshold sits at 0.35. One point, $50\% \times 50\%$ ($s = \frac{1}{4}$), lies between them and routes only in decode (-3.6%; Figure 4). The three winners are the arms whose prefill passes are routed under the thresholds: their TTFT moves by -61.2 to -78.2%, the others' by -33.6 to $+44.2\%$. With always route (Figure 3a) the change runs from $+23.8\%$ at $25\% \times 25\%$ to -62.9% at $75\% \times 75\%$. This is the policy's return under routed mode. Switching rules reduce the cost of unprofitable policies (Table 8).

Table 8: The nine RandomSkip arms (n=1 each): the rule's crossover V^* and own thresholds, and per switch setting the cell's estimated p90 occupancy (running requests $\times$ mean context), the share of decode passes and rows served in routed mode, and the mean E2E change (%).

				always route (Fig. 3a)	shared thresholds (Fig. 3b)			per-policy thresholds (Fig. 3c)		
r	d	V^*	own thresholds	E2E Δ	occupancy	routed	E2E Δ	occupancy	routed	E2E Δ
25 %	25 %	1261k	1260k/1580k	+23.8	408k	95/91%	+3.2	407k	0/0%	-0.9
25 %	50 %	631k	630k/790k	+12.6	408k	95/91%	+1.3	408k	0/0%	-1.6
25 %	75 %	420k	420k/530k	+1.4	408k	95/91%	+0.2	407k	0/0%	-2.1
50 %	25 %	631k	630k/790k	+12.9	407k	95/91%	+3.1	408k	0/0%	-2.5
50 %	50 %	315k	320k/390k	-11.9	407k	95/91%	-3.6	408k	92/76%	-1.7
50 %	75 %	210k	210k/260k	-38.8	404k	93/88%	-21.5	407k	93/87%	-17.7
75 %	25 %	420k	420k/530k	-0.3	407k	95/91%	+0.9	408k	0/0%	-0.5
75 %	50 %	210k	210k/260k	-42.9	405k	93/87%	-21.8	406k	93/88%	-18.3
75 %	75 %	140k	140k/180k	-62.9	263k	93/87%	-49.1	273k	93/87%	-51.0

E FAITHFULNESS AT THE TOKEN LEVEL

E.1 FIRST-TOKEN AGREEMENT

Routed mode reproduces the checkpoint's first-token choice on both tokenized inputs in Table 9. They represent CoQA `lm-eval` document 80 and serving request `coqa:train:856:15`. We compare the top-5 first-token log-probabilities under native PyTorch, upstream SGLang and routed mode.

Table 9: First generated token and its log-probability on the two prompts whose generations are empty. Each served arm matches its PyTorch reference within 0.01 nats.

prompt	base, PyTorch (A)	upstream SGLang (C)	checkpoint, own code (B)	vSKIPPER routed (D)
doc 80	`Island` -0.489	`Island` -0.489	`<eot>` -0.882	`<eot>` -0.872
856:15	`Le` -0.047	`Le` -0.047	`<eot>` -1.033	`<eot>` -1.033

The checkpoint itself selects `<|eot_id|>` on both prompts. Each served arm matches its native reference within 0.01 nats: the immediate stop reflects the checkpoint's first choice.

All four quality arms receive identical token ids. This controls for a tokenization difference: `lm-eval`'s HF backend omits BOS, while the server's text path prepends it. On doc 80, adding BOS changes the checkpoint's first choice from `Island` to `<eot>`. The serving gate records `coqa:train:856:15` as one immediate-stop exception.

E.2 GSM8K ANSWER EXTRACTION

`lm-eval` ships two GSM8K filters, and on this checkpoint they disagree by 11 pp: flexible extraction puts the checkpoint's cost at -5.53 pp, strict-match at $+5.31$. Each fails on a different arm.

Flexible extraction takes the *last* number in the generation, and the checkpoint appends a confidence epilogue after its answer marker (`#### 594\nConfidence: 95%`), so the filter reads 95. The epilogue is the checkpoint's: under its own code the checkpoint emits it on 2.9% of held-out documents, and on 3.0% when served by vSKIPPER, while the base model never does.

Strict-match matches `#### <number>` and returns `[invalid]` when the marker is missing or followed by anything else, such as `#### $21`; the base model under a chat template trips it on a fifth of its documents. On the $1{,}319$ held-out documents:

arm	strict-match unusable	flexible extraction misreads
A base model, `lm-eval` backend	271 (20.5%)	0 (0.0%)
C base model, upstream SGLang	274 (20.8%)	0 (0.0%)
B checkpoint, its own code	121 (9.2%)	38 (2.9%)
D checkpoint, served by vSKIPPER	121 (9.2%)	39 (3.0%)

The *composite* applies one rule to every arm: use strict-match when it parses, otherwise flexible extraction. Both filters come from `lm-eval`. The rule preserves the two base-model scores and correctly reads the checkpoint's confidence epilogues. Its checkpoint cost is -3.26 pp, between the two single-filter values.

The two diagnosed format failures are disjoint: strict-match reads the marked answers whose epilogues mislead flexible extraction. Disagreements where strict-match parses incorrectly and flexible extraction is correct affect 0.00% of upstream documents at every rate, at most 0.53% of hybrid documents and 0.61% of always-route documents. The single-filter comparisons give DiD values of -0.38 ± 0.74 pp with flexible extraction and $+0.15 \pm 0.66$ with strict-match. Both agree within their intervals with the composite result, -0.23 ± 0.68 in Table 5.

E.3 THE SERVED CONFIGURATION ACROSS LOADS

Table 10 scores the three served systems at every rate of the load grid; Table 5 (right) is its knee row. Below the knee, where these cells keep GSM8K and CoQA decode in dense mode, the hybrid shows no resolved difference from upstream (-0.38 ± 1.39 pp on GSM8K). At and above the knee, where

it routes 89.5 and 98.7% of GSM8K decode rows, it shows none from always-route. On BBH, the hybrid serves 99.0% of decode rows through routed mode at the knee. Every cell is one repetition with a per-document interval.

Table 10: Task quality of the three served systems across the load grid, one repetition per cell, same metrics as Table 5. "Routed" is the share of decode rows served in routed mode (vSKIPPER / always-route).

Task	Load	upstream	vSKIPPER	always-route	routed (%)	vs. upstream (pp)	vs. always-route (pp)
GSM8K	$0.75 \times Q^*$	77.63	77.26	74.00	0 / 100	-0.38 ± 1.39	$+3.26 \pm 2.34$
GSM8K	$0.95 \times Q^*$	77.63	74.00	74.07	90 / 100	-3.64 ± 2.22	-0.08 ± 1.46
GSM8K	$1.25 \times Q^*$	77.79	74.00	73.92	99 / 100	-3.79 ± 2.30	$+0.08 \pm 0.65$
CoQA	$0.75 \times Q^*$	76.74	76.74	77.66	0 / 100	$+0.00 \pm 0.00$	-0.92 ± 1.94
CoQA	$0.95 \times Q^*$	76.74	77.68	77.73	71 / 100	$+0.94 \pm 0.82$	-0.05 ± 1.76
CoQA	$1.25 \times Q^*$	76.71	79.65	77.66	99 / 100	$+2.94 \pm 1.55$	$+1.99 \pm 1.45$
BBH	$0.75 \times Q^*$	67.93	59.35	59.58	98 / 100	-8.59 ± 1.06	-0.23 ± 0.52
BBH	$0.95 \times Q^*$	67.87	59.47	59.68	99 / 100	-8.40 ± 1.07	-0.22 ± 0.38
BBH	$1.25 \times Q^*$	67.98	59.45	59.59	100 / 100	-8.52 ± 1.07	-0.14 ± 0.26

F LATENCY AND MAKESPAN ACROSS LOAD

The benchmark is open-loop with fixed work and a common injection window. Each arm's makespan includes that window and its final drain. While the server keeps up with arrivals, execution savings appear as lower latency: smaller concurrent batches, shorter token intervals and less queueing. Makespan moves only through the drain, the last and longest requests to finish, so it reflects the tail of each cell (Figure 5).

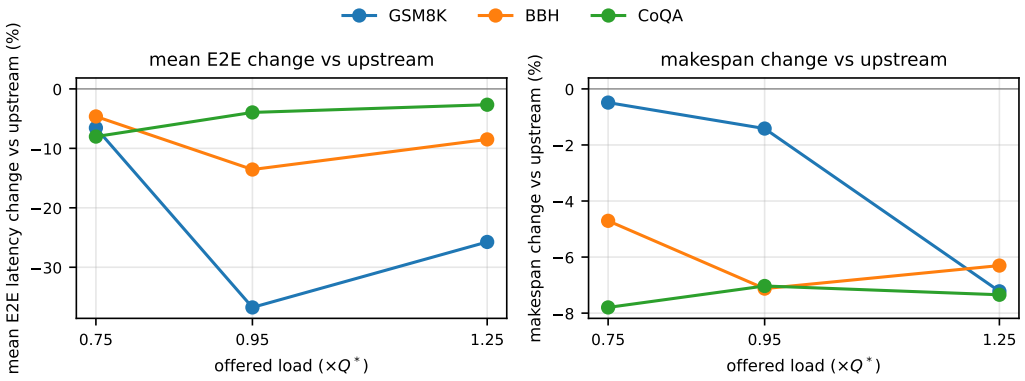


Figure 5: Mean E2E change (left) and makespan change (right) against offered load, per workload, from the cells of Table 2.

The E2E reduction is roughly uniform across output lengths at and below the knee on GSM8K and BBH. Binning by pinned output tokens gives per-bin spreads of 0.7 and 1.3 pp on GSM8K and 0.9 and 3.6 pp on BBH. It widens to 8.0 pp on GSM8K in overload and to 7.8–11.1 pp on CoQA, where a request's place in the queue sets its gain.

Mean TTFT falls on GSM8K at every rate, where prefill routing shortens the prefill passes (Table 2). On BBH and CoQA it shows no resolved difference from upstream: prefill routes few passes on CoQA (Table 16), and on BBH prefill-only routing stays near upstream (Table 13).

Throughput under saturation. Table 4 counts the output tokens emitted and the requests completed between the first and the last arrival of each $1.25 \times Q^*$ cell, the injection window both arms share; the drain is excluded. Both arms are saturated there: in every repetition of every column they complete only 0.50–0.80 of the requests that arrive inside the window, and their backlog grows through it. On GSM8K and CoQA the running batch stays full while the waiting queue grows, and TPS and RPS

agree within one percentage point. On BBH, whose shared few-shot prefixes keep each request cheap in KV, the backlog accumulates in the running batch; in-flight requests grow through the window, and the RPS change ($+7.4\%$) exceeds the TPS change ($+4.1\%$).

G Natural Stopping and the Output Lengths

The matched-work lane replays output lengths recorded from the base model's natural generation. Here the base model on upstream SGLang and the FlexiDepth checkpoint on vSkipper each generate to their own stop. Both receive the same open-loop arrivals, with one repetition per cell (Table 11). A generation reaching its remaining context window is counted as a context-limit hit.

Table 11: The two stacks under natural stopping, one repetition per cell: mean output tokens per request, context-limit hits, mean E2E latency and makespan. A third row serves the checkpoint in the always-route configuration.

Workload	Load ($\times Q^*$)	Stack	out tokens	limit hits	E2E s	makespan s
GSM8K	0.75	Base + upstream	116	3	6.1	437
		FlexiDepth + vSkipper	126	5	6.4	440
		FlexiDepth always-route	299	82	18.8	632
GSM8K	0.95	Base + upstream	116	3	22.4	388
		FlexiDepth + vSkipper	330	99	38.2	668
		FlexiDepth always-route	302	84	32.4	612
GSM8K	1.25	Base + upstream	116	3	57.2	388
		FlexiDepth + vSkipper	271	69	61.6	563
		FlexiDepth always-route	297	81	64.6	598
BBH	0.75	Base + upstream	257	42	14.0	349
		FlexiDepth + vSkipper	394	114	24.4	548
		FlexiDepth always-route	408	121	25.8	565
BBH	0.95	Base + upstream	255	41	21.9	329
		FlexiDepth + vSkipper	405	120	31.9	551
		FlexiDepth always-route	420	128	33.8	576
BBH	1.25	Base + upstream	262	45	36.3	336
		FlexiDepth + vSkipper	405	121	44.7	545
		FlexiDepth always-route	413	125	46.0	565
CoQA	0.75	Base + upstream	166	82	8.9	563
		FlexiDepth + vSkipper	156	76	7.5	497
		FlexiDepth always-route	83	38	3.9	393
CoQA	0.95	Base + upstream	164	81	11.9	546
		FlexiDepth + vSkipper	125	59	8.8	410
		FlexiDepth always-route	86	40	7.3	362
CoQA	1.25	Base + upstream	166	82	25.1	541
		FlexiDepth + vSkipper	161	78	24.0	478
		FlexiDepth always-route	84	39	20.4	348

Matched work measures the runtime; natural stopping characterizes each model–runtime stack, whose generated work differs. At the GSM8K knee the checkpoint generates 330 output tokens per request against the base model's 116 ($+185\%$). It reaches the context limit on 99 requests against 3; on BBH it generates 53–59% more. The long requests carry most of a cell's tokens, so each stack's latency and makespan follow its generated work. They rise on GSM8K and BBH. On CoQA the checkpoint generates similar or less work (-3 to -24% output tokens), and its stack finishes sooner (E2E -4 to -26%). These are stack-level differences, not runtime effects.

The matched-work cells freeze the base model's per-request lengths, separating the serving engine's execution cost from changes in generated work. In this natural lane, requests run to EOS or their remaining context window ($6,171$–$7,988$ tokens), with context-limit hits counted explicitly. Task quality is measured separately under `lm-eval`'s per-task generation limits.

G.1 Native Checkpoint Repetition Analysis

Under the checkpoint authors' native PyTorch execution, FlexiDepth increases the incidence of runaway generation over its base model on the same prompts (Table 12). Both the base model and checkpoint generate greedily from frozen token ids within an 8192-token window. On a seeded random sample of 500 prompts of the GSM8K knee cell, the base model repeats on none and the checkpoint on $2.2 \pm 1.3\%$, each a degenerate repetition of the answer line.

The propensity is prompt-specific. The checkpoint repeats on 20.3% of the 143 served-arm repeaters, against 2.0% of the controls. The sampled groups stop once repetition is detected.

Under serving, which prompts repeat varies with load and stack (Table 11). Sampling penalties, such as repetition and frequency penalties, are a standard mitigation. The evaluation keeps greedy decoding under the third-party `lm-eval` protocol, and tuning these parameters is left to future work.

Table 12: Native repetition attribution on the GSM8K knee cell's prompts (PyTorch, base model and checkpoint). A repetition is a degenerate repeated answer or a generation that reached the window (cap).

Model	Prompt group	prompts	repeats	%	cap	mean	p90
Base, batch 16	random sample	500	0	0.0	0	108	163
	of which served-arm repeaters	15	0	0.0	0	100	153
	of which never repeated	485	0	0.0	0	109	164
FlexiDepth, batch 16	random sample	500	11	2.2	0	155	213
	of which served-arm repeaters	15	4	26.7	0	364	1024
	of which never repeated	485	7	1.4	0	148	209
Base, batch 1	served-arm repeaters	143	0	0.0	0	107	153
	upstream repeaters	3	2	66.7	2	4656	6956
	controls	100	0	0.0	0	102	145
FlexiDepth, batch 1	served-arm repeaters	143	29	20.3	29	1583	7278
	upstream repeaters	3	0	0.0	0	194	215
	controls	100	2	2.0	2	273	196
FlexiDepth, batch 16	served-arm repeaters	143	29	20.3	29	1518	7009
	upstream repeaters	3	0	0.0	0	188	222
	controls	100	3	3.0	3	330	196

H Mechanism Ablation

Table 13: Mechanism ablation: mean E2E and makespan change (%) against upstream at the main output lengths. The served arm is the n=6 main result; each ablation arm is one repetition, and negative is better.

	served (n=6)		decode only		prefill only		always route		no promotion	
rate	E2E	makespan	E2E	makespan	E2E	makespan	E2E	makespan	E2E	makespan
GSM8K										
0.75×	-6.6 ± 1.5	-0.5 ± 0.1	+0.4	+0.0	-8.6	-0.7	-5.5	-1.0	-6.7	-0.5
0.95×	-36.8 ± 2.0	-1.4 ± 0.1	-14.3	-0.9	-35.6	-1.2	-46.5	-1.9	-32.6	-1.2
1.25×	-25.7 ± 0.7	-7.2 ± 0.2	-7.8	-2.6	-19.9	-5.4	-26.3	-7.4	-24.5	-7.0
BBH										
0.75×	-4.6 ± 0.5	-4.7 ± 0.5	-3.7	-4.7	-0.3	+0.1	-3.2	-5.7	+2.0	-2.8
0.95×	-13.6 ± 1.2	-7.1 ± 0.2	-16.0	-6.2	+0.5	+0.1	-13.0	-6.7	-9.9	-3.8
1.25×	-8.5 ± 1.0	-6.3 ± 0.7	-9.7	-5.9	-0.2	+0.1	-9.0	-6.1	-6.3	-3.3
CoQA										
0.75×	-8.0 ± 0.6	-7.8 ± 0.3	-8.0	-7.8	+0.8	+0.4	-1.3	-5.1	+3.4	+0.5
0.95×	-4.0 ± 4.4	-7.0 ± 0.4	-2.6	-7.3	-0.9	+0.3	+5.5	-6.1	+4.9	+5.3
1.25×	-2.7 ± 0.2	-7.3 ± 0.1	-2.7	-8.1	+0.1	+0.1	+0.0	-7.1	-1.1	-4.0

Four arms isolate the mechanisms of §3.1 and §3.3 on all three workloads at the three main rates, one repetition each, paired against upstream at the main output lengths (Table 13). *Decode only*: routed mode on the decode passes the decode thresholds select, prefill dense. *Prefill only*: routed mode on the prefill rounds the admission rule selects, decode dense. *Always-route*: both phases without the mode switch. *No promotion*: the served hybrid without the one-way promotion into routed decode (§3.3). All routed arms share the cohort execution path (Appendix B).

GSM8K. Prefill routing provides the gain: its knee-cell E2E change, -35.6%, is close to the served arm's -36.8%. Decode-only remains inactive below the knee. Always-route is faster at the knee in this single repetition; the CoQA paragraph shows what the switch gains in exchange.

BBH. Decode routing provides the gain across the load grid; prefill-only stays near upstream. The served arm is faster than always-route at and below the knee. The switch avoids routed-mode costs on passes it keeps dense.

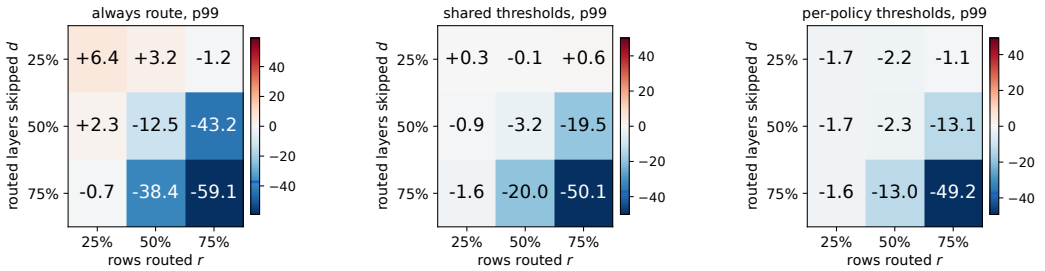
CoQA. The gain comes from finishing the long-output tail sooner: the served and decode-only arms improve makespan by 7.0–8.1%. The switch matters most at the knee, where always-route is 9.5 pp behind the served arm in mean E2E.

H.1 Kernel Headroom

Routed mode's GEMMs use count-bounded Triton kernels so captured execution can consume row counts held on the device. Outside capture, the same rows go through cuBLAS. The tile tuner times cuBLAS on the same packed rows beside each candidate. On the A100, the served kernels take a median of 1.18× cuBLAS's time (0.82–1.79× over the 20 operation/count-range shapes). On the RTX A6000, the median is 1.13× (0.77–1.53×). The serving gains already include these count-bounded kernels. The comparisons identify target shapes for further kernel optimization.

I Tail Latencies

Table 14 gives the p50 and p99 companions of Table 2; Figure 6 the p99 companion of the Random-Skip maps.



(a) always route (b) shared thresholds (c) per-policy thresholds

Figure 6: p99 E2E change against upstream for the nine RandomSkip arms under the three switch settings, the same cells as Figure 3.

Table 14: p50 and p99 companions of Table 2: mean percentage change with 95% intervals over the same six paired repetitions.

			E2E		TPOT		TTFT	
Workload	Rate	n	p50	p99	p50	p99	p50	p99
BBH	$0.75 \times Q^*$	6	-3.0 ± 0.5	-8.2 ± 1.0	-5.0 ± 0.8	-10.7 ± 2.7	-0.0 ± 2.8	+12.0 ± 37.5
BBH	$0.95 \times Q^*$	6	-15.6 ± 1.7	-11.4 ± 0.3	-13.8 ± 0.8	-22.4 ± 1.5	-10.3 ± 2.4	+6.8 ± 31.3
BBH	$1.25 \times Q^*$	6	-10.4 ± 1.1	-8.9 ± 0.9	-10.1 ± 1.2	-10.4 ± 2.0	+0.7 ± 11.6	-16.4 ± 7.1
CoQA	$0.75 \times Q^*$	6	+1.8 ± 2.6	-10.1 ± 0.4	+2.1 ± 2.1	-2.0 ± 18.4	-0.6 ± 2.7	+5.4 ± 30.0
CoQA	$0.95 \times Q^*$	6	+2.5 ± 9.9	-9.2 ± 0.5	+4.2 ± 11.3	+1.0 ± 24.0	-2.6 ± 7.8	+0.8 ± 5.2
CoQA	$1.25 \times Q^*$	6	-0.3 ± 0.4	-9.1 ± 0.2	-2.5 ± 0.6	-1.7 ± 0.6	+0.3 ± 0.8	+0.3 ± 0.7
GSM8K	$0.75 \times Q^*$	6	-6.3 ± 1.5	-9.7 ± 2.8	-6.3 ± 1.6	-10.0 ± 3.7	-6.3 ± 0.6	-4.4 ± 4.8
GSM8K	$0.95 \times Q^*$	6	-38.4 ± 1.9	-37.3 ± 2.5	-43.6 ± 2.1	-15.3 ± 3.8	-53.8 ± 11.6	-74.5 ± 3.0
GSM8K	$1.25 \times Q^*$	6	-26.4 ± 0.7	-21.2 ± 0.6	-10.3 ± 0.2	-18.9 ± 1.7	-41.8 ± 1.3	-40.7 ± 0.9

J Absolute Baseline Numbers

Table 15 gives the upstream arm's absolute values behind the relative rows of Table 2, averaged over the same repetitions. The artifact separately records the served arm's absolute means directly. The injected arrival rate matches the offered rate within 2% on every cell. Makespan exceeds the injection window by the drain, the tail of the longest pinned requests; on CoQA the drain is 61–76% of the makespan at every load, so its gain appears in makespan.

Decode-graph coverage. Both arms capture decode graphs up to 1,024 rows. The load samples show how often a decode batch exceeded SGLang's default A100 capture setting of 256 rows, which the default configuration would have run without a graph. For the baseline it is 57% at the GSM8K

knee, 39% at the BBH knee and 5% at the CoQA knee; for the served arm, 20, 40 and 6%. Samples exceed 1,024 rows only at BBH overload, on 23% (baseline) and 23% (served) of samples; there upstream runs uncaptured and vSKIPPER replays its captured graphs in capture-sized chunks.

Prefix cache. The radix prefix cache is enabled on both arms. The cached share of prefill tokens, upstream / vSKIPPER, is 1.7 / 1.4% on GSM8K, 91.0 / 90.2% on BBH (shared few-shot prefixes) and 1.1 / 1.1% on CoQA, within 0.8 pp between the arms. vSKIPPER keeps separate dense and routed namespaces keyed by a request's prefill mode, so a prompt reuses only KV computed under its own prefill mode, and a request whose decode mode differs from its prefill mode does not insert its generated KV.

Table 15: Upstream SGLang absolutes per cell, means over the repetitions; drain is makespan minus the injection window.

Workload	$\times Q^*$	TTFT (ms)	TPOT (ms)	E2E (s)	makespan (s)	drain (s)	tok/s	req/s injected	req/s offered
GSM8K	0.75	197	52.8	6.1	437	62	955	9.6	9.75
GSM8K	0.95	894	238.3	26.9	390	94	1087	12.2	12.35
GSM8K	1.25	29017	274.2	58.8	377	152	1106	16.0	16.25
BBH	0.75	132	64.4	14.4	354	195	2943	25.2	25.50
BBH	0.95	250	118.2	23.9	330	205	3111	31.9	32.30
BBH	1.25	1251	200.5	38.7	323	228	3180	42.0	42.50
CoQA	0.75	211	138.5	8.8	561	345	1169	18.5	18.75
CoQA	0.95	493	796.7	13.1	543	373	1193	23.5	23.75
CoQA	1.25	10204	1809.5	30.8	534	405	1240	30.9	31.25

Table 16: Pass share per phase: the fraction of the served arm's prefill and decode passes that ran in routed mode (a routed pass may also carry dense requests), from each cell's attestation. Means use the six repetition logs.

Workload	$\times Q^*$	n	prefill passes routed (%)	decode passes routed (%)
GSM8K	0.75	6	24	0.0
GSM8K	0.95	6	42	92.9
GSM8K	1.25	6	94	99.0
BBH	0.75	6	37	95.9
BBH	0.95	6	54	98.7
BBH	1.25	6	67	99.2
CoQA	0.75	6	0	69.8
CoQA	0.95	6	1	97.1
CoQA	1.25	6	0	99.8

K TWO QWEN3 SKIPPERS: THE PROFITABILITY BOUNDARY

Two skippers we trained test where the method starts to be profitable. Both are alignment-only straight-through-gate routers and projectors on Qwen3 with thinking off, routing layers 18–35 of 36. Each is served by the same runtime with the thresholds the rule derives from its declared skip (Appendix D); training and checkpoint selection are in Appendix M. We report Qwen3-4B at step 20,000 and three Qwen3-8B checkpoints. Table 17 identifies each checkpoint, its decode thresholds and its attested always-route decode skip. The thresholds use skip-rate estimates from a 32-request training-time routing probe: 0.25 for 4B, and 0.445, 0.485, and 0.520 for the three 8B checkpoints. These probe estimates are used only for pre-serving threshold calibration.

Each follows the main matched-work protocol: its own upstream Q^* (17 and 14 req/s), its own output lengths, GSM8K at the three rates, three paired repetitions, bf16. In both Qwen arms the Project-Only KV path applies Qwen3's per-head q/k norms and projects all rows. The Qwen natural lane runs under an 8,192-token output budget (Qwen3's context window is 40,960 tokens); the quality suites use `lm-eval`'s per-task limits.

Serving. Higher skipping translates into larger serving gains in the three-checkpoint comparison. The highest-skipping Qwen3-8B checkpoint lowers mean E2E the most at every load, with each interval excluding zero (Table 17). Qwen3-4B shows no resolved change in mean E2E at any load. The table also reports its single-repetition run under the shared Llama thresholds.

Table 17: Qwen3 checkpoints against upstream: matched-work percentage changes with 95% intervals (A100, bf16, GSM8K; n=3, except the Llama-threshold rows). Group headings give the checkpoint, exit/enter thresholds and always-route decode skip at the knee.

		E2E		TPOT		TTFT		
Rate	n	mean	p99	mean	p99	mean	p99	makespan
Qwen3-4B, step 20k; thresholds 320k/390k, skip 0.119								
$0.75 \times Q^*$	3	+1.2 ± 1.9	+1.1 ± 2.3	+1.3 ± 1.9	+2.7 ± 1.9	-0.2 ± 7.6	+0.3 ± 58.9	+0.1 ± 0.4
$0.95 \times Q^*$	3	+2.7 ± 2.8	+0.9 ± 2.3	+1.0 ± 1.1	+0.8 ± 0.7	+9.3 ± 11.2	+10.4 ± 10.2	+4.5 ± 0.7
$1.25 \times Q^*$	3	+0.6 ± 1.0	+0.7 ± 1.8	-0.0 ± 0.6	-0.2 ± 2.8	+1.1 ± 1.3	+1.4 ± 2.4	+3.1 ± 0.7
Qwen3-8B, penalty 2×10^{-4}, step 10k; thresholds 180k/220k, skip 0.345								
$0.75 \times Q^*$	3	-6.4 ± 2.1	-4.5 ± 2.1	-2.3 ± 0.5	-1.1 ± 3.1	-19.5 ± 4.7	-18.7 ± 5.1	+1.1 ± 0.2
$0.95 \times Q^*$	3	-3.8 ± 1.7	-3.7 ± 0.9	-2.1 ± 0.2	-1.1 ± 6.2	-4.9 ± 2.8	-4.9 ± 0.9	+0.6 ± 0.7
$1.25 \times Q^*$	3	-3.0 ± 0.2	-3.0 ± 0.7	-2.3 ± 0.3	-1.9 ± 1.4	-3.2 ± 0.2	-3.3 ± 0.4	+0.3 ± 0.4
Qwen3-8B, penalty 2×10^{-4}, step 18.75k; thresholds 160k/200k, skip 0.350								
$0.75 \times Q^*$	3	-6.5 ± 4.6	-4.4 ± 3.7	-2.3 ± 1.6	-1.3 ± 3.6	-19.7 ± 13.8	-20.9 ± 8.7	+0.7 ± 0.9
$0.95 \times Q^*$	3	-5.0 ± 2.2	-4.2 ± 1.8	-2.5 ± 0.7	-3.0 ± 4.1	-6.7 ± 3.3	-6.3 ± 2.4	-0.2 ± 1.1
$1.25 \times Q^*$	3	-3.5 ± 0.9	-3.3 ± 0.7	-2.5 ± 0.2	-1.2 ± 2.3	-3.9 ± 1.2	-3.7 ± 0.6	+0.3 ± 0.6
Qwen3-8B, penalty 4×10^{-4}, step 15k (served); thresholds 150k/190k, skip 0.416								
$0.75 \times Q^*$	3	-10.3 ± 1.8	-6.3 ± 0.8	-3.4 ± 0.4	-2.9 ± 7.2	-32.1 ± 8.4	-31.5 ± 5.3	-0.5 ± 1.0
$0.95 \times Q^*$	3	-6.6 ± 0.3	-6.0 ± 0.3	-3.1 ± 0.2	-2.4 ± 3.5	-9.1 ± 0.3	-8.7 ± 0.7	-0.8 ± 0.3
$1.25 \times Q^*$	3	-4.9 ± 0.3	-4.8 ± 0.9	-3.3 ± 0.4	-2.0 ± 0.8	-5.5 ± 0.4	-5.4 ± 1.1	-1.0 ± 1.2
Qwen3-4B, shared Llama thresholds (n=1)								
$0.75 \times Q^*$	1	+0.9	+3.6	+0.9	-1.0	+1.8	+4.0	+0.0
$0.95 \times Q^*$	1	+2.3	+0.7	+0.4	+1.6	+8.5	+9.4	+3.8
$1.25 \times Q^*$	1	+1.2	+1.4	+0.3	+1.1	+1.8	+2.4	+3.5

Quality and tuning. Training improves skipping and accuracy together along the 2×10^{-4} run: the later checkpoint skips more and scores higher both natively and when served (Tables 17 and 18). FlexiDepth's released Llama-3-8B checkpoint was trained by its authors in two stages, alignment and then annealing. Our Qwen3 skippers use only the alignment stage, stopped before one epoch, with a short search over the router penalty, the weight of the training term that rewards skipping; a larger penalty trades accuracy for skipping (Appendix M). They are proofs of concept and do not show the best trade-off that full two-stage training can reach on Qwen3. These results motivate further hyperparameter tuning of the skipper's quality–efficiency trade-off, which we leave to future work.

Table 18 repeats the 2×2 design of Table 5 for all four checkpoints. All four difference-in-differences intervals contain zero at one run per arm, with no resolved loss beyond the checkpoint's own in the routed comparison.

The separately measured hybrids route 86.5% (4B) and 99.3% (8B) of decode rows at the knee and score $+1.52 \pm 2.49$ and $+4.25 \pm 2.68$ pp against always-route.

Table 18: The Qwen3 checkpoints' quality on GSM8K (composite exact match, 1,319 held-out documents, bf16): the 2×2 as in Table 5 (left), and the served hybrid at its knee cell, as in Table 5 (right), with routed shares ordered hybrid / always-route.

Model	A	B	C	D	$(B-A)$	$(D-C)$	DiD (pp)
Qwen3-4B	79.45	76.80	80.14	75.74	-2.65	-4.40	-1.74 ± 1.98
Qwen3-8B 2×10^{-4} 10k	80.97	75.44	81.12	75.28	-5.53	-5.84	-0.30 ± 1.84
Qwen3-8B 2×10^{-4} 18.75k	80.97	77.71	81.12	78.01	-3.26	-3.11	+0.15 ± 1.80
Qwen3-8B 4×10^{-4} 15k (served)	80.97	70.96	81.12	70.51	-10.01	-10.61	-0.61 ± 2.11

Model	upstream	vSKIPPER (served)	always-route	routed (%)	vs. upstream (pp)	vs. always-route (pp)
Qwen3-4B	80.67	77.71	76.19	87 / 100	-2.96 ± 2.15	$+1.52 \pm 2.49$
Qwen3-8B 2×10^{-4} 10k	80.36	76.65	75.21	99 / 100	-3.71 ± 2.23	$+1.44 \pm 2.60$
Qwen3-8B 2×10^{-4} 18.75k	80.36	80.06	78.39	99 / 100	-0.30 ± 2.27	$+1.67 \pm 2.41$
Qwen3-8B 4×10^{-4} 15k (served)	80.36	75.89	71.65	99 / 100	-4.47 ± 2.33	$+4.25 \pm 2.68$

L HARDWARE TRANSFER

The protocol of Table 2 was repeated on two other devices with the same code (the A6000 build adds only its decode-graph range and tuned tile table), launch profile, suite, gates and prefill floor. Each device gets its own knee, its own output lengths, its own decode-graph range, and three paired repetitions against upstream on that device (Table 7). The thresholds come from the rule with the bandwidth in Table 19; no workload-dependent configuration search was run on any device.

H100 (80 GB HBM3). Its upstream Q^* is 14 req/s (Table 20), with 270k/340k-token thresholds. It uses the A100's tile table unchanged. End-to-end latency falls on every row with every interval excluding zero, most at the knee (-39.4%). The sign transfers across the devices, each measured on its own load grid.

RTX A6000 (48 GB GDDR6, 768 GB/s). By default, upstream captures decode graphs up to 32 rows on this card. VSKIPPER captures up to 256, the occupancy this card reaches at its knee, and the upstream control is launched with `--cuda-graph-max-bs 256`. Its Q^* is 6 req/s, with 60k/80k-token thresholds. The device provides 101 KB of shared memory per block; the A100's cohort tiles require 110 KB. The tuner selects tiles within the device's shared-memory limit, holding block-K fixed and requiring bit-identical outputs to the A100 tiles. It replaced 19 of 20 tile keys (Appendix H.1); the same tuner supports new devices. Mean end-to-end latency falls on every row, with the interval excluding zero at the knee and in overload (Table 7).

Table 19: Bandwidth and compute inputs to the rule, estimated crossovers and served decode thresholds. Estimates use $V^* = \tau\,\mathrm{BW}/(sL_rb)$ with $\tau = 2.67$ ms, $s = 0.5$, $L_r = 16$ and $b = 4{,}096$ bytes.

Device	BW (GB/s)	TFLOPS	V^*	exit / enter	Q^* (req/s)
A100-SXM4-80GB	1,935	312	157.7k	160k / 200k	13
H100 80 GB HBM3	3,350	990	273.0k	270k / 340k	14
RTX A6000 48 GB GDDR6	768	155	62.6k	60k / 80k	6

Table 20: The upstream Q^* sweeps: contiguous integer offered rates, Q^*, achieved output tok/s at Q^* and at the highest rate, and the growth left above Q^* (computed from unrounded values).

Workload	rates (req/s)	Q^*	tok/s at Q^*	tok/s at top	growth above Q^* (%)
GSM8K	8–18	13	1079	1080	+0.1
BBH	20–36	34	3121	3106	-0.5
CoQA	16–33	25	1216	1217	+0.0
GSM8K (H100)	12–22	14	1226	1228	+0.1
GSM8K (RTX A6000)	2–10	6	493	492	-0.2

M REPRODUCIBILITY

Models. Base: For Llama-3-8B, `NousResearch/Meta-Llama-3-8B-Instruct` at revision `53346005fb0e`. Skipper: the released `xuan-luo/FlexiDepth-Llama-3-8B-Instruct` (routes the last 16 of 32 layers), served as published. Qwen3-4B: `Qwen/Qwen3-4B` at `1cfa9a72`; Qwen3-8B: `Qwen/Qwen3-8B` at `d117af2f`; both with thinking disabled through the chat template. Their skippers are released with the artifact: the served checkpoints as full exports, the two other Qwen3-8B checkpoints as router deltas, each with its probe and gate records.

Training of the Qwen3 skippers. Both use alignment-only FlexiDepth training: the router and projector are trained over a frozen base with a straight-through hard gate. Training uses `allenai/tulu-3-sft-mixture`, sequence length 2,048, lr 10^{-4}, global batch 32 and at most one epoch (27,643 steps). Checkpoints are saved every 1,250 steps with a 100-document scoring probe and a 32-prompt routing probe that reports the chat-template skip share.

Training varies the router penalty using the probe's skip share and score. The effective coefficient is penalty divided by accumulation, with accumulation fixed at 4. Qwen3-4B uses $10^{-5}/4$ through step 20,000. Qwen3-8B uses $10^{-4}/4$ for 7,500 steps, then $2 \times 10^{-4}/4$ or $4 \times 10^{-4}/4$ to step 20,000.

The routing probe's skip shares of the reported checkpoints are 0.445 and 0.485 at steps $10{,}000$ and $18{,}750$ of the 2×10^{-4} run and 0.520 at step $15{,}000$ of the 4×10^{-4} run, which is served.

Both ran on a CloudLab node with four A100-SXM4-40GB GPUs. The trainer and export scripts are in `vskipper/training/qwen3-flexidepth/`, alongside the input files and gate records.

Engine. Baseline: upstream SGLang at commit `602c8615a1`, a separate checkout. VSKIPPER: the same commit plus the runtime. Both use the shared launch profile and graph-capture settings of §4.1. Appendix C specifies the boot-configuration and execution-difference checks. The switch's constants live in one module and are attested in every cell.

Q^* sweeps. Table 20 gives the upstream rate sweeps used to measure Q^* (§4.1). A rate counts as growth when it and the next rate both exceed the running maximum by at least 1%. The Qwen Q^* values use the same rule.

Cost of a rerun. The measured cells sum to about 34 GPU-hours on the A100: main cells 12.5, ablations 5.2, RandomSkip 7.4 and Qwen 8.9. The H100 cells take 1.8 GPU-hours and the RTX A6000 cells 4.2. Rate sweeps, length-recording runs, quality cells and restarts roughly double the A100 figure. At the rental prices in Table 21, the serving evidence reruns for under USD 100; the A100 price covers four GPUs. The Qwen3-4B skipper trains in about 40 A100-40GB GPU-hours; each Qwen3-8B run, trained to step $20{,}000$, in about 60.

Table 21: The rented Vast.ai machines behind the serving cells, by host id (instance numbers change on every rental); driver as reported by the node, prices as rented, DLPerf as Vast.ai reports it for the whole machine.

Host	CPU	GPU	Price	Driver	DLPerf	Location
458442	2× AMD EPYC 7713	4× A100-SXM4-80GB (NVLink)	$4.60/h	595.71	469.1	Japan
638511	AMD EPYC 9454 (24 vCPU)	1× H100 80 GB HBM3 (SXM)	$2.09/h	595.71	314.6	Germany
314882	2× Intel Xeon E5-2699 v4	1× RTX A6000 48 GB	$0.49/h	580.173	54.5	Virginia, US

Artifacts. The package contains experiment specifications, checked cell records, analysis scripts, table and figure generators, and the tile tuner with each device's tuned tile table.