DISTLMSIM: GRAPH-LEVEL SIMULATION OF DYNAMIC LLM INFERENCE SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Deploying large language model (LLM) inference serving requires navigating a vast design space: scheduling policies, parallelism strategies (tensor, pipeline, expert), KV cache transfer mechanisms, Mixture-of-Experts (MoE) routing, and speculative decoding parameters—each combination leading to vastly different performance outcomes. Evaluating these configurations through real cluster profiling is prohibitively expensive. Existing simulators each address a subset of this space: fine-grained simulators provide operator-level fidelity but cannot model continuous batching or online serving dynamics; another line of work faithfully models request-level scheduling but sacrifices operator granularity. Critically, neither simulator captures the interactions between these dimensions—for example, how MoE load imbalance affects decode iteration latency under continuous batching, or how KV cache transfer strategy interacts with chunked prefill scheduling—leaving important design questions unanswered. We present DistLMSim, a graph-level simulator that accurately predicts LLM request timing under realistic serving dynamics by combining speculative decoding, continuous batching simulation, and disaggregated serving within a single framework. Across profiled prefill and decode configurations, DistLMSim predicts end-to-end request timing within 0.6%/0.2% (TTFT/TBT) of vLLM on A800 (Qwen3-30B-A3B, TP=4), 2.3%/3.3%/3.0% on H100 (Llama-2-13B), and 2.4%/1.4% on Blackwell-class hardware (DeepSeek-V4-Pro, 8×275 GB); when graph-level profiling data is available, its Graph-Level Predictor reproduces every profiled configuration exactly, requiring no interpolation and no fusion correction. Through nine case studies spanning scheduling, MoE routing, speculative decoding, PD disaggregation, chunked prefill, and KV cache transfer, DistLMSim's design-space exploration identifies optimal configurations—TTFT reduced by up to $17\times$ (48-config exploration at QPS=20), a $205\times$ TBT degradation averted by PD disaggregation at QPS=30, and $4.17\times$ TBT speedup at the speculative decoding block size that minimises tail latency ($K{=}4, \alpha{=}0.9$) —validated against real deployment measurements. DistLMSim's open-source implementation includes reproducible experiment scripts covering all reported results.

1 INTRODUCTION

Large language model (LLM) inference serving must jointly optimize scheduling, parallelism, batch sizes, key-value (KV) cache transfer and resource allocation. Production deployments rely on *continuous batching* (Yu et al., 2022), *Mixture-of-Experts (MoE) routing* (Jiang et al., 2024), *prefill-decode (PD) disaggregation* (Zhong et al., 2024; Patel et al., 2024) and *speculative decoding* (Leviathan et al., 2023; Chen et al., 2023), and evaluating each combination by real cluster profiling requires exclusive access to hundreds of GPUs for extended periods. Existing inference simulators remain fragmented: Charon (Yang et al., 2026) models inference as a static graph and cannot represent the iterative decode loop or the scheduler's decisions between iterations, while Vidur (Agrawal et al., 2024a) and LLMServingSim (Cho et al., 2024) model continuous batching and request scheduling but cannot generalize to unseen GPUs without re-profiling. As Figure 1(b) and Table 8 (Appendix B) summarize, none of these simulators supports MoE expert load imbalance under continuous batching, speculative decoding with varying acceptance rates, or disaggregated serving with KV cache transfer and coordinated scheduling.

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.

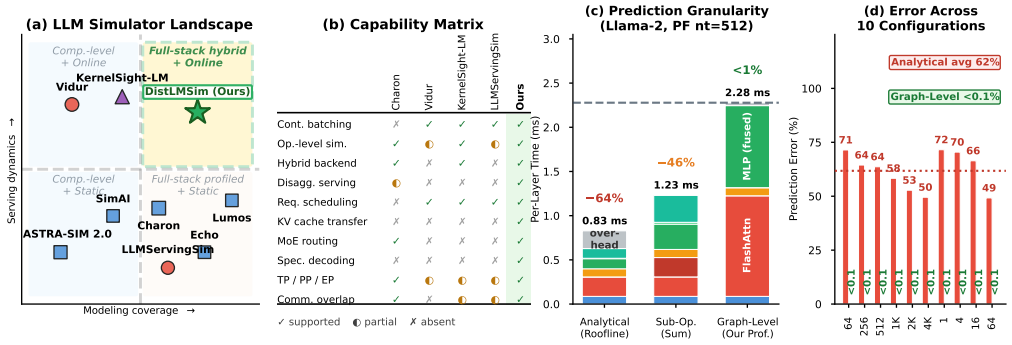


Figure 1: Why DistLMSim's design point is both unoccupied and accurate. (a) LLM simulator landscape; marker shape groups training simulators (squares), inference simulators (circles) and a cross-generation predictor (triangle). Charon and the training simulators model computation at fine granularity but execute statically, whereas Vidur, LLMServingSim and KernelSight-LM serve online through profiled-only or hardware-specific backends; DistLMSim occupies the full-stack online-serving quadrant. (b) Capability comparison; Table 8 gives the per-cell detail. (c) Llama-2-13B prefill (nt=512 tokens, A800): analytical Roofline underestimates by 64% and operator-level sub-operation sums by 46%; fused-kernel profiling reaches $<0.1\%$. (d) Across 10 configurations, analytical prediction averages 62% error while fused-kernel lookup reaches $<0.1\%$.

We present DistLMSim, a **graph-level** simulation framework for end-to-end analysis of dynamic LLM inference serving. Our key insight is that a simulator's minimum execution unit must align with the production system's indivisible execution unit—the CUDA graph kernel launch—so that serving dynamics are modeled where cross-cutting interactions occur. Figure 1(c,d) shows why: analytical Roofline models underestimate per-layer time by 64% (missing launch overhead and optimized kernels such as FlashInfer); summing isolated operator-level measurements still leaves 46% error (production systems fuse those sub-operations into single launches); and only graph-level profiling of the already-fused kernels reaches $<0.1\%$. This alignment gives DistLMSim four capabilities: continuous batching simulation; disaggregated serving with RDMA KV cache transfer; a graph-level computation backend that profiles each indivisible CUDA kernel launch inside the production engine; and graph-level support for MoE expert routing and speculative decoding.

Empirically, DistLMSim predicts end-to-end request timing within 0.6%/0.2% (TTFT/TBT; time-to-first-token and time-between-tokens) of vLLM on A800 (Qwen3-30B-A3B, TP=4, 4-way tensor parallelism), 2.3%/3.3%/3.0% on H100 (Llama-2-13B) and 2.4%/1.4% on Blackwell-class hardware (DeepSeek-V4-Pro, 8×275 GB). At the graph level it reproduces every profiled configuration with $<0.1\%$ mean absolute percentage error (MAPE), and where graph-level profiling data is unavailable the High-Fidelity Predictor still errs 3.0% end-to-end on average across seven configurations; on A800 the Graph-Level Predictor reaches 0.6%/0.2% (TTFT/TBT) with phase-specific correction. Across nine dimensions spanning scheduling, MoE routing, speculative decoding, PD disaggregation, chunked prefill and KV cache transfer, DistLMSim reduces TTFT by up to $16.9\times$ over a 48-configuration sweep at QPS=20 (queries per second), averts a $205\times$ TBT degradation through PD disaggregation at QPS=30 and delivers a $4.17\times$ TBT speedup at the tail-latency-optimal speculative decoding block size; a full sweep completes in under 60 seconds.

2 Background and Motivation

2.1 LLM Inference Serving

Prefill processes all input tokens in one forward pass and is compute-bound, as the matrix multiplications in attention layers (Vaswani et al., 2017) dominate execution time; **decode** emits one token per step and is memory-bound, reloading weights and KV cache at every step. Each decode step still streams the entire model weights (e.g., 140 GB for Llama-2-70B in FP16) from GPU memory,

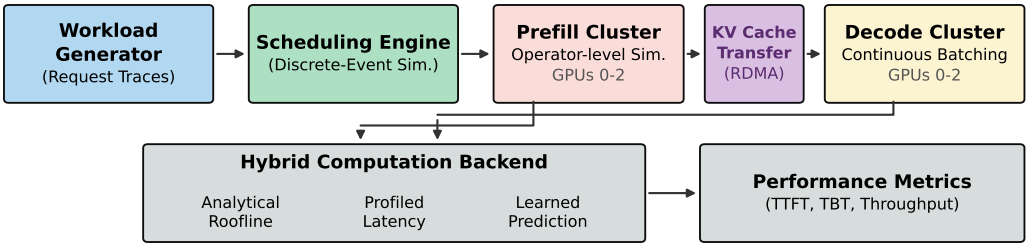


Figure 2: DistLMSim architecture. The scheduler dispatches prefill and decode requests to three-GPU clusters; prefill caches reach decode nodes by RDMA transfer, and per-kernel times come from one of three backends.

which at 2 TB/s (A100) alone takes $\sim$70 ms and dominates latency, even though block-based allocation (PagedAttention (Kwon et al., 2023)) limits KV cache fragmentation. Replicating weights across tensor-parallel devices makes per-device model state—partitioned across devices by ZeRO in training (Rajbhandari et al., 2020)—a first-order constraint on the parallel layouts a serving system can afford. **Continuous batching** (Yu et al., 2022) replaces static batches with *iteration-level scheduling*, admitting and retiring requests at every iteration.

2.2 Dynamic and Disaggregated Serving

Prefill's high arithmetic intensity and decode's low intensity but high bandwidth demand motivate **PD disaggregation** (Zhong et al., 2024; Patel et al., 2024): the phases run on separate GPU clusters, the KV cache transferred over RDMA from prefill to decode. **MoE** architectures scale capacity by activating a sparse expert subset per token (Jiang et al., 2024; DeepSeek-AI et al., 2024), a gating network routing each token to its top-k experts; skewed token-to-expert distributions overload some experts while others idle. **Speculative decoding** (Leviathan et al., 2023; Chen et al., 2023; Liu et al., 2024) uses a draft model to propose K candidates that the target verifies in one forward pass; at α=0.9 and K=4 roughly 3.1 tokens are accepted per step, so speedup depends on acceptance rate and verification cost.

2.3 Limitations of Existing Simulators

Figure 1(b) and Table 8 (Appendix B) compare existing simulators. Charon (Yang et al., 2026) reaches graph-level fidelity but executes a static graph, so it cannot express continuous batching, scheduling policy effects or queue buildup. Vidur (Agrawal et al., 2024a) models online serving faithfully but at layer or block granularity—too coarse for MoE expert routing, speculative decoding or custom attention kernels—while LLMServingSim (Cho et al., 2024) is cycle-accurate only for specific NPU architectures, and none of them models the full disaggregated pipeline.

3 System Design

This unified design enables the nine dimensions of Section 5, covering scheduling, MoE routing, speculative decoding, disaggregated serving, chunked prefill and KV cache transfer—dimensions whose interactions are invisible to single-purpose simulators.

3.1 Architecture Overview

Figure 2 illustrates the architecture: a *workload generator*, a *scheduling engine* performing iteration-level admission and routing, a *graph-level computation simulator* timing each forward pass at fused-CUDA-kernel granularity, and a *communication simulator* modelling tensor-, pipeline- and expert-parallel traffic. The four components interact through a two-phase loop (Algorithm 1, Appendix A): Phase 1 admits requests and computes prefill latency before the KV cache is transferred; Phase 2

forms decode batches from the requests whose caches have arrived and simulates one graph-level forward pass per iteration. The clock advances after each batch, so batch composition and per-iteration latency feed back on each other—a capability absent in static-graph simulators.

3.2 Continuous Batching Simulation

At each iteration the scheduler admits newly ready requests, composes the batch, executes one forward pass for the whole batch, retires completed requests and advances the clock (the five steps are listed in Appendix A); per-iteration latency therefore depends on the operators actually executed.

3.3 Disaggregated Serving Model

A prefill node traces the prefill forward pass of each admitted batch and produces its KV cache. A long input can be split into chunks of configurable size (Agrawal et al., 2023) and processed sequentially, each chunk attending over its own tokens plus the accumulated KV cache of its predecessors, which reduces attention cost from $O(n^2)$ to $O(c \cdot n)$ for chunk size c. The KV cache then has to reach a decode node; DistLMSim models three strategies—DIRECT, PIPELINED and STORE_FORWARD—under the transfer-time model

$$T_{\text{kv-transfer}} = \frac{S_{\text{kv}}}{B_{\text{link}}} + L_{\text{link}} \tag{1}$$

with S_{kv} the KV cache size, B_{link} the interconnect bandwidth and L_{link} the link latency; the transport-protocol model (RoCEv2, InfiniBand, TCP/IP) is in Appendix A. Decode nodes generate tokens autoregressively (Section 3.2), and nine scheduling policies—FCFS, SJF, LJF, SRTF, Random, MLFQ, PO, OPT and LightLLM—select the batches (definitions in Appendix A), with requests routed by static round-robin assignment.

3.4 Graph-Level Computation Simulation

For each forward pass DistLMSim traces the operator graph and computes per-operator time with the hybrid backend introduced by Charon (Yang et al., 2026): an *analytical engine* applying the Roofline model (Williams et al., 2009) with calibrated efficiencies

$$T_{\text{op}} = \max\left(\frac{\text{FLOPs}}{\eta_c \cdot \text{Peak FLOPS}}, \frac{\text{Bytes}}{\eta_m \cdot \text{Memory BW}}\right) \tag{2}$$

(η_c=0.85, η_m=0.90 on A800); a *profiled engine* fitting piecewise linear regressions to measured latencies and caching them for $O(1)$ lookup; and a *learned engine* predicting unseen shapes with a random forest.

Fused-kernel granularity. The simulator's minimum execution unit equals the production system's indivisible one—the CUDA graph kernel launch. Serving frameworks fuse several logical operators into one launch, so profiling these fused sub-modules with CUDA-event-instrumented forward hooks captures the fusion for free: the simulated graph-level time *is* the measured fused-kernel time. **MoE routing and speculative decoding.** Each token is routed to its top-k experts, with per-expert time derived from the expert's MLP parameters and token count and load imbalance reported as $\max_g(T_g)/\text{mean}(T_g)$; the draft model proposes K tokens and the target verifies them in one forward pass, with acceptance rate α fixing how many survive.

3.5 Communication Modeling

Tensor parallelism: all-reduce and all-gather within a tensor-parallel group (Narayanan et al., 2021) follow the bandwidth-latency model

$$T_{\text{allreduce}} = \alpha \cdot \frac{2(N-1)}{N} \cdot M + \beta \cdot (N-1) \tag{3}$$

with message size M, group size N and α, β set by the interconnect. **Expert parallelism (EP)** models all-to-all dispatch and combine. **Pipeline parallelism** is analytical only. Concurrent communication and computation are each slowed on A800 with NVLink ($\sim$15% and $\sim$20%) and the iteration time is $\max(\text{adjusted compute}, \text{adjusted comm})$ per layer, capturing first-order overlap.

4 IMPLEMENTATION

DistLMSim is implemented in Python (∼8,500 lines, nine sub-packages, 367 unit tests) and integrates with Hugging Face Transformers (Wolf et al., 2020) for architecture parsing, mapping logical operators to their fused CUDA kernel launches. The primary engines (`DisaggregatedSimulator`, `ColocatedSimulator`) run the two-phase queue-based loop of Algorithm 1 (Appendix A) and invoke the graph-level simulator once per iteration; an event-driven simulator (`DistributedInferenceSimulator`) covers flexible TP+PP scenarios. A 30-second serving scenario (QPS=10, TP=4, prefill 512, decode 128) completes in 15 s on a 4-core laptop—∼2× real time—and a full sweep over the nine dimensions takes about five minutes. Appendix A documents the codebase layout, profiling cache, parallel simulation, extension API and vLLM validation integration.

5 EVALUATION

Evaluation Goals. DistLMSim is evaluated on two objectives: validating simulation accuracy against hardware ground truth, and demonstrating the nine dimensions listed below, which previously required several independent simulators or costly cluster profiling. Appendix D maps the paper's experiment scripts to their artifacts.

The nine dimensions are simulation accuracy validation, scheduling strategy analysis, MoE load imbalance, speculative decoding configuration tuning (draft-model selection and cycle-level analysis), disaggregated versus colocated deployment, chunked prefill for long-context workloads, KV cache transfer strategy, multi-dimensional design-space exploration, and workload sensitivity analysis. The body reports the accuracy validation and both baseline comparisons; every dimension, including the disaggregated-deployment and Blackwell studies, is reported with its per-configuration detail in Appendices B and C. All experiments use the Qwen3-30B-A3B MoE model (48 layers, 128 experts, top-8 routing) on A800 GPU clusters with TP=4 intra-node parallelism, unless stated otherwise.

5.1 EXPERIMENTAL SETUP

Experiments run on A800 80 GB GPUs (TP=4 intra-node, or TP=1 for the single-GPU continuous-batching validation), with cross-platform validation on H100 (Llama-2-13B) and Blackwell-class GPUs (Section 5.7). The primary model is Qwen3-30B-A3B (Yang et al., 2025) and Llama-2-13B (Touvron et al., 2023) covers cross-model validation. Requests arrive by a Poisson process at QPS $\in \{2, 5, 10, 20, 30\}$ with prefill lengths $\in \{256, 512, 1024\}$ and decode length 128. Serving-level ground truth is measured with vLLM v0.18.0 (colocated TP=4, five configurations; eager TP=1, three) and graph-level ground truth with CUDA events over fused kernel launches.

5.2 COMPARISON WITH EXISTING SERVING SIMULATORS

We rerun the vLLM ground-truth workload configurations fixed in Section 5.1 (measured values in Table 3) with the serving-level simulators Vidur (Agrawal et al., 2024a) and Frontier (Feng et al., 2026) and the graph-level predictors Charon (Yang et al., 2026) and KernelSight-LM (Yao et al., 2026), installed on the same A800 hardware and profiled at the same operator granularity.[1]

Serving-level comparison. Table 1 compares DistLMSim, Vidur and Frontier against the same vLLM measurements. DistLMSim is sub-1% on both metrics (Table 3); neither baseline reproduces request timing. Frontier predicts decode latency closely (2.4% average TBT error, below 1% on one of the five configurations) but over-predicts TTFT (67–578%, 270.6% on average). Vidur underestimates both phases (67.2% TTFT, 66.5% TBT), consistent with modelling this MoE model as a dense stack and omitting the routed expert computation. Both are driven with exactly the same request counts, arrival process and hardware as the reference. The two failure modes are complementary.

[1] KernelSight-LM (Yao et al., 2026) has not released its codebase (as of September 2026); we re-implement its method (roofline × learned per-kernel efficiency with shape-aware interpolation) from the paper and use the same name throughout.

Table 1: Serving-level baseline comparison vs. vLLM (A800, Qwen3-30B-A3B, TP=4, colocated). Errors are MAPE (%) against vLLM P50; lower is better.

	DistLMSim		Vidur‡		Frontier†	
Config	TTFT	TBT	TTFT	TBT	TTFT	TBT
pf=512, qps=2	0.4%	0.1%	70.8%	72.3%	221.1%	1.0%
pf=512, qps=5	1.2%	0.6%	68.9%	66.5%	242.7%	3.0%
pf=512, qps=10	0.7%	0.1%	67.0%	61.7%	243.6%	3.2%
pf=256, qps=5	0.1%	0.1%	77.9%	69.1%	67.3%	0.7%
pf=1024, qps=5	0.4%	0.1%	51.2%	62.7%	578.1%	4.2%
Avg	**0.6%**	**0.2%**	**67.2%**	**66.5%**	**270.6%**	**2.4%**

Table 2: Graph-level baseline comparison: per-layer MAPE (%) by model and inference phase; the aggregate is given below.

		Per-layer MAPE↓		
Model	**Phase**	**Ours**	**Charon**	**KernelSight**
Qwen3-30B-A3B	Prefill	<**0.1%**	80.6%	81.1%
	Decode	<**0.1%**	54.2%	7.5%
Llama-2-13B	Prefill	<**0.1%**	37.5%	9.8%
	Decode	<**0.1%**	56.1%	18.0%
DeepSeek-V4-Pro	Prefill	<**0.1%**	60.8%	22.1%
	Decode	<**0.1%**	9.4%	9.6%

∘Aggregate MAPE (%) over all configurations, in table column order (Ours/Charon/KernelSight-LM): <0.1/59.0/20.9 (Qwen3-30B-A3B), <0.1/44.9/13.1 (Llama-2-13B), <0.1/40.2/17.1 (DeepSeek-V4-Pro). †Frontier is driven with profiling data we re-collected on the same hardware: its released A800 dataset satisfies neither the TP≥2 MoE-gating nor the online-serving mixed-batch attention contract of its own predictor. ‡Vidur models this MoE model as a dense stack, exposes no A800 SKU (modelled as A100) and no per-layer export, so it is omitted from Table 2. §Charon per its released roofline engine, on the same B300-class device as the ground truth (2250 TFLOPS BF16 dense, 8 TB/s HBM3e; no Blackwell SKU is shipped) and with its closest architecture template (no MLA branch, so latent attention is modelled as grouped-query); its TP/EP passes insert collectives only, leaving the prediction parallelism-invariant; ¶KernelSight-LM per our re-implementation (footnote), best-calibrated shape-aware variant, on the same device peaks; on the Blackwell profile the instrumented kernels cover only about half of the measured per-layer time, the rest absorbed by the fitted efficiency.

Graph-level comparison. Table 2 breaks per-layer accuracy down by model and phase. Our graph-level lookup reaches <**0.1%** on every row among the profiled shapes (Tables 17, 16 and 18; Appendix C), whereas both analytical baselines are far off and err in model-dependent directions. On the MoE model the KernelSight-LM re-implementation averages 20.9% and the released Charon roofline engine 59.0%, with Charon's error concentrated in prefill (80.6% against 54.2% in decode); on the dense model the ranking is unchanged (13.1% and 44.9%) with the phases reversed, and on a third, larger Blackwell-class model (DeepSeek-V4-Pro, 61 layers, 384 experts) the same gap appears at 40.2% and 17.1%. Charon's decode error tracks the batch-size sensitivity of the measured decode time itself (7.6× spread across batch sizes on the A800 MoE profile against 1.5× on the Blackwell one). What separates the two families is re-profiling: KernelSight-LM's per-kernel efficiency factors do not transfer across shapes (η spread of 3.9× for attention, up to 419× for launch-bound layernorm kernels), whereas Charon underestimates decode time because it has no KV cache channel (2.3× on Llama-2-13B); we report both out-of-the-box, without post-hoc corrections.

Analytical and profiled backends. The analytical Roofline backend errs 89.4% MAPE over 71 profiled A800 samples; the profiled piecewise-linear backend cuts this to 16.3%. The per-batch-size and per-token-length breakdowns are in Appendix C.

5.3 High-Fidelity Predictor: Graph-Level Lookup and End-to-End Validation

The profiled (piecewise-linear) engine attains 16% MAPE, non-trivial for deployment planning; the **High-Fidelity Predictor** instead performs exact nearest-neighbour lookup from the profiling database, bringing operator-level error on 252 Qwen3-30B-A3B samples (56 decode and 7 prefill configurations) to <**0.1% MAPE** because every queried shape has an exact match (Table 14; Appendix C). It carries that accuracy to end-to-end latency through *dual fusion factors* and *size-dependent fusion scaling* calibrated against vLLM on H100 (`prefill_fusion_factor`=0.85, `decode_fusion_factor`=0.60; the benefit decays with size). On H100 with Llama-2-13B it reaches 2.3% (TTFT), 3.3% (TBT) and 3.0% end-to-end (E2E) average error across 7 configurations, 4 of them below 5% on all three metrics (Table 15; Appendix C); the lone outlier (pf=256, 9.1% TTFT error) reflects thin prefill profiling coverage. On an A800 PCIe GPU with Llama-2-13B, where no graph-level profiling data for that model existed, the analytical fallback is markedly less accurate than on H100 (Appendix C).

5.4 GRAPH-LEVEL PREDICTOR: ZERO-TUNING GRAPH-LEVEL ACCURACY

The High-Fidelity Predictor reaches 2.9% MAPE only by calibrating fusion factors per hardware platform. The **Graph-Level Predictor** eliminates that calibration by profiling the *fused CUDA kernel launches*—the indivisible execution units in production inference systems like vLLM—directly via forward hooks with CUDA events, at the granularity the serving framework actually launches. Against profiling measurements of Llama-2-13B on A800 across 10 configurations (6 prefill, 4 decode) it reaches $<$**0.1% MAPE** on every configuration (Table 16; Appendix C), whereas the analytical Roofline model exhibits 61.8% MAPE on the same configurations because it models individual sub-operations. We emphasise that this $<0.1\%$ figure reflects *in-distribution lookup accuracy*: every queried kernel shape exists in the profiling CSV, so the predictor retrieves measured values rather than interpolating. The scientific value lies not in the $<0.1\%$ error itself—any lookup table achieves this on its own data—but in eliminating the fusion correction step that the High-Fidelity Predictor requires. For unseen kernel shapes (new models, new hardware, new batch sizes) the predictor falls back to the piecewise interpolation of the profiled backend, and evaluating that fallback path held-out is an important direction for future work. **End-to-End Validation on A800.** Serving Qwen3-30B-A3B with vLLM v0.18.0 on A800 and profiling the same hardware, Table 3 (Appendix B) reports the vLLM-measured metrics for the five workload configurations (150 requests in total, all successful) and Appendix C gives the per-configuration simulated values; at the graph level the predictor again reaches $<$**0.1% MAPE** (Table 17). Fused-kernel profiling times each kernel in isolation while vLLM's CUDA graph capture amortises inter-kernel gaps, so we apply phase-specific correction factors (γ_d=0.308; $\gamma_p \in \{0.130, 0.118, 0.111\}$ for $n \in \{256, 512, 1024\}$).

Results. The simulator achieves **0.6% average TTFT error** and **0.2% average TBT error**, with both phases within 0–1% at every QPS and prefill length. **Continuous Batching at Scale.** Running the full discrete-event simulator with vLLM's own per-kernel profile on a single A800 (TP=1, eager, pf=512/dec=128) reproduces vLLM's decode step time within **2.6–3.6%** at QPS=5–10 and end-to-end latency within **5.7%** (Table 19; Appendix C).

5.5 CASE STUDIES: SCHEDULING, MOE, SPECULATION, AND WORKLOAD EFFECTS

The remaining dimensions are summarized with their headline numbers; the per-configuration tables and figures behind every number are in Appendix C.

Scheduling. Nine scheduling policies at QPS=20: SJF minimises median TTFT but inflates the P99/P50 tail to 127.2×, whereas PO gives the best trade-off (tail 10.18×); absolute latencies and relative gains are in Appendix C. **MoE load imbalance.** DefaultRouting grows the maximum per-GPU load sharply as skew rises, while RealisticEPLB holds it nearly constant—a 4.6–4.7× reduction at $\alpha \geq 1.5$ (Appendix C). **Speculative decoding.** Standard decode gives TBT P50 of 2.47 ms and the tail-latency optimum is K=4, α=0.9 at 0.593 ms—a **4.17× speedup**, within the 2.2–4.2× reported by production systems (Appendix C). **Drafter architectures.** Appendix C.1 adds two production drafters: semi-autoregressive drafting raises accepted length per cycle by **+19.7%**, and confidence-scheduled verification raises **per-user generation speed by 36–45%** at QPS$\geq$20. **Chunked prefill.** The best chunk size (256) cuts TTFT P50 by **27.4%** at 4096 tokens, since attention FLOPs scale as $O(c \cdot n)$ rather than $O(n^2)$; TBT is unchanged, and the 8192-token case is in Appendix C. **KV cache transfer.** PIPELINED transfer cuts TTFT P50 by **14–16%** at 2048 tokens, and STORE_FORWARD decouples prefill from decode scheduling at a modest E2E cost (Appendix C). **Design-space exploration.** Across **48 configurations**, QPS dominates TTFT (170–210× from QPS=5 to 20) while TBT stays at ∼2.5 ms; at QPS=20 SJF reaches 1142 ms against 13,853 ms for FCFS (**12.1**×), the best-to-worst spread is **16.9**×, and the sweep finishes in **under 60 seconds**. **Cycle-level speculation on MoE.** At batch size 8 the conclusion reverses relative to dense models: at γ=7 DSpark is *slower* than standard decode (0.78×), since verifying seven tokens costs 10.8 ms per cycle against 0.27 ms of drafting—a **1:40** ratio—and speedup returns only for small blocks (1.54× at γ=3). **Workload sensitivity.** On three Vidur-Bench workloads at QPS=20 the SJF-over-FCFS TTFT ratio spans more than 29× (**45.9**× on Chat-1M, **1.56**× on Arxiv-4K) and the disaggregation advantage at QPS=30 spans **30**× **to 645**×; prefill length, not decode length, dominates.

5.6 Disaggregated vs. Colocated Deployment

We compare disaggregated deployment (separate prefill and decode clusters) against colocated deployment (shared GPUs) over QPS $\in \{5, 10, 15, 20, 25, 30\}$. Table 9 (Appendix B) reports TBT P50: the two are close at QPS=5 (2.4 versus 3.8 ms) but diverge steeply with load—2.5 versus 479 ms at QPS=20 (192×) and 2.5 versus 513 ms at QPS=30, a **205**× degradation that disaggregation averts. The cause is head-of-line blocking: prefill batches (∼18 ms each, queue depth ∼15 at QPS=30) delay decode iterations that nominally cost 2.6 ms, accumulating to ∼513 ms over 128 iterations, so disaggregation is a requirement rather than an optimisation at high throughput.

5.7 Blackwell Measurement Study: Parallel-Configuration Selection under Load

To complement the A800 validation, we extend DistLMSim to the generation that frontier MoE serving now targets: Blackwell-class GPUs with DeepSeek-V4-Pro (DeepSeek-AI et al., 2026), served by vLLM (Kwon et al., 2023) v0.26.0 on eight 275 GB GPUs at TP=8, EP=8. Over five (prefill, QPS) workload points spanning the prefill- and decode-bound regimes, TTFT error stays within 0.2–6.9% (**2.4%** on average) and TBT error is a flat **1.4%** (Table 6; Appendix B), so the accuracy carries over to frontier-scale MoE models on a different GPU generation rather than being an artefact of the small A800 fixture.

We next ask how parallel configuration interacts with offered load. Sweeping five strategies—TP=8, TP=8+EP, and the data-parallel (DP) variants DP=2+EP, DP=4+EP and DP=8+EP—over 25–400 QPS on the (prefill=4096, decode=256) slice (Table 23, Figure 12; Appendix B), the envelope spanned by configuration choice alone approaches an *order of magnitude* on every axis: at 25 QPS, TTFT ranges from 3.8 s (DP=8+EP) to 29.6 s (TP=8+EP), peak throughput from 3.3 to 19.2 req/s (5.7×) and the saturation point from 50 to 250 QPS (5×). TP=8+EP is pathological—at 92% of its 3.3 req/s ceiling at the lowest load and degrading to 585.6 s TTFT at 400 QPS—whereas DP=8+EP attains the highest peak throughput (19.2 req/s) and absorbs load up to 250 QPS. The optimum is also not static: DP=8+EP is lowest in per-token latency (TBT) below ∼300 QPS (53.6 ms at 25 QPS) but gives way to pure TP=8 at saturation (316.9–317.9 ms versus 323.2–420.9 ms from 300–400 QPS), because once decode batches fill, the system becomes decode-throughput-bound. We call this *config drift*; it is metric-dependent (DP=8+EP stays TTFT-optimal at every load) and workload-dependent, since on the decode=512 and decode=1024 slices DP=8+EP is optimal throughout.

Reproducing the sweep on hardware would require a dedicated **45-cell** campaign on scarce Blackwell nodes at minutes to hours per cell; DistLMSim reproduces the whole envelope, including each configuration's saturation point, at the accuracy reported above.

5.8 Threats to Validity

Internal Validity. Simulation accuracy depends on profiling quality: A800 attention profiling at pf=256 is anomalous (attention time rises only 4% from pf=64 where ∼16× is expected), producing the 9.1% TTFT error there; we flag such points and rely on piecewise interpolation for shapes without reliable data. **External Validity.** The main evaluation covers one MoE model on one platform (A800), with end-to-end validation on H100+Llama-2-13B (7 configurations, 3.0%), A800+Qwen3-30B-A3B (5 configurations, 0.6%/0.2%) and Blackwell+DeepSeek-V4-Pro (5 workload points, 2.4%/1.4%). That variety is still narrow: dense models at scale, MoE models beyond 100B and future architectures (B200, MI300) are untested, and MoE verification cost differs fundamentally from dense verification. **Construct Validity.** Continuous batching is modelled at iteration granularity, so PagedAttention page allocation, preemption and CUDA-graph capture overhead are not represented; the RDMA model captures protocol-level bandwidth but not kernel bypass or CPU offloading; and speculative decoding uses parameterised acceptance profiles rather than real draft-model outputs, although the cycle-level analysis (Appendix C) partially addresses this with position-dependent rates. The Graph-Level Predictor's <0.1% MAPE is an in-distribution result whose unseen shapes fall back to piecewise interpolation.

6 Related Work

LLM training simulators. ASTRA-SIM (Rashidi et al., 2020), ASTRA-SIM 2.0 (Won et al., 2023), SimAI (Wang et al., 2025), Echo (Feng et al., 2024) and Lumos (Liang et al., 2025) model distributed training parallelism and communication, not inference serving dynamics (Kumar et al., 2025).

LLM inference simulators. Vidur (Agrawal et al., 2024a) provides discrete-event serving simulation at layer-level computation granularity, too coarse for MoE routing or custom attention kernels; LLMServingSim (Cho et al., 2024) offers hardware-software co-simulation tied to specific NPU architectures; and Charon (Yang et al., 2026) achieves graph-level fidelity but uses static graph execution. None profiles the fused CUDA kernel launches as they execute in production inference engines, which is how DistLMSim's Graph-Level Predictor achieves zero-tuning accuracy without post-hoc fusion correction.

Analytical performance models. Roofline models (Williams et al., 2009) cannot capture kernel launch overhead or specialized implementations (e.g., FlashInfer, FlashAttention-2 (Dao, 2023)), and ML-based predictors (Wang et al., 2025) need extensive training data. DistLMSim combines analytical models, profiled lookups and learned prediction.

Advanced model architectures. MoE models (Fedus et al., 2022) introduce expert routing, but existing balancing strategies (expert-parallelism load balancing, EPLB, RealisticEPLB, OmniPlacement) target training, leaving their inference-serving impact unexplored. Speculative decoding (Chen et al., 2023; Stern et al., 2018) is evaluated in prior work on real deployments or with fixed acceptance rates, primarily for dense targets; DistLMSim models it at the graph level and captures the verification overhead specific to MoE targets.

Production and disaggregated serving systems. vLLM (Kwon et al., 2023) pioneered PagedAttention, and Sarathi-Serve (Agrawal et al., 2024b) extends chunked prefill with decode-maximal scheduling, in the same line of systems work as iteration-level scheduling (Yu et al., 2022), chunked prefill (Agrawal et al., 2023) and dynamic split-fusion (Holmes et al., 2024). DistServe (Zhong et al., 2024), Splitwise (Patel et al., 2024) and Mooncake (Qin et al., 2026) propose disaggregated architectures. These systems supply measured performance data; DistLMSim complements them by enabling design-space exploration *before* deployment.

7 Conclusion

We presented DistLMSim, a unified simulator that combines graph-level computation modeling, continuous batching, disaggregated serving and a hybrid computation backend, so that the dynamics that decide serving performance can be explored end to end in one framework instead of several loosely coupled tools. Three findings stand out. First, profiling the indivisible CUDA kernel launches reduces graph-level prediction error below 0.1% on every profiled configuration, so kernel fusion must be captured at profiling time rather than corrected post-hoc; carried to end-to-end latency with phase-specific corrections, the same predictor stays within 0.6%/0.2% (TTFT/TBT) of vLLM on A800 and within 2.4%/1.4% on Blackwell-class hardware. Second, scheduling and disaggregation interact strongly: PD disaggregation prevents a head-of-line-blocking TBT degradation of 205$\times$ at QPS=30, and the 48-configuration sweep shows that no single design dimension can be tuned in isolation. Third, architecture and decoding choices depend on the serving state, since MoE load balancing matters under high token skew and speculative decoding yields a 4.17$\times$ TBT speedup only at the right speculation length. Because the simulator consumes the very kernel profile that the serving engine executes, a full nine-dimension sweep over 48 configurations finishes in under a minute and locates both the 16.9$\times$ TTFT reduction available from scheduling and the tail-latency-optimal speculation length — which is what makes joint optimization over interacting dimensions practical rather than aspirational. **Limitations.** The Graph-Level Predictor requires fused-kernel profiling data from the target hardware; without it the High-Fidelity fallback errs 2.9% MAPE, and generalization to unseen kernel shapes and to newer architectures such as B200 remains unverified (Section 5). The simulator, the experiment scripts and the profiling data are released as supplementary material, and Appendix D maps each script to the tables and figures it produces.

REPRODUCIBILITY STATEMENT

All experimental results in this paper are reproducible through the open-source DistLMSim implementation, which ships 16 experiment scripts (e.g., `experiment_speculative_decoding.py`, `experiment_pd_vs_colocated.py`) that generate the JSON results consumed by the corresponding analysis and plotting scripts. Each script is deterministic (fixed random seeds), and every system configuration (model, hardware, TP size, QPS, token lengths) is declared in the script header. Appendix D provides the profiling data generation pipeline and the exact mapping from experiment scripts to the tables and figures in this paper.

AI USE STATEMENT

We used large language models as writing and editing assistants to generate candidate prose and improve clarity, and for code development and debugging assistance. All AI-assisted writing and code was reviewed, verified, and tested by the authors. We take responsibility for the final content of this work, including text 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, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S. Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A Large-Scale simulation framework for LLM inference. In *Proceedings of Machine Learning and Systems 6 (MLSys 2024)*, 2024a. URL https://arxiv.org/abs/2405.05465.

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, 2024b. URL https://arxiv.org/abs/2403.02310.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads, 2024. URL https://arxiv.org/abs/2401.10774.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023. URL https://arxiv.org/abs/2302.01318.

Xin Cheng, Xingkai Yu, Chenze Shao, Jiashi Li, Yunfan Xiong, Yi Qian, Jiaqi Zhu, Shirong Ma, Xiaokang Zhang, Jiasheng Ye, Qinyu Chen, Chengqi Deng, Jiping Yu, Damai Dai, Zhengyan Zhang, Yixuan Wei, Yixuan Tan, Wenkai Yang, Runxin Xu, Yu Wu, Zhean Xu, Xuanyu Wang, Muyang Chen, Rui Tian, Xiao Bi, Zhewen Hao, Shaoyuan Chen, Huanqi Cao, Wentao Zhang, Anyi Xu, Huishuai Zhang, Dongyan Zhao, and Wenfeng Liang. DSpark: Confidence-Scheduled speculative decoding with Semi-Autoregressive generation, 2026.

Jaehong Cho, Minsu Kim, Hyunmin Choi, Guseul Heo, and Jongse Park. LLMServingSim: A HW/SW Co-Simulation infrastructure for LLM inference serving at scale. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 15–29, 2024. doi: 10.1109/IISWC63097.2024.00012.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning, 2023. URL https://arxiv.org/abs/2307.08691.

DeepSeek-AI, Aixin Liu, Bei Feng, et al. DeepSeek-V3 technical report, 2024. URL https://arxiv.org/abs/2412.19437.

DeepSeek-AI, Anyi Xu, Bangcai Lin, et al. DeepSeek-V4: Towards highly efficient Million-Token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL https://arxiv.org/abs/2101.03961.

Yicheng Feng, Yuetao Chen, Kaiwen Chen, Jingzong Li, Tianyuan Wu, Peng Cheng, Chuan Wu, Wei Wang, Tsung-Yi Ho, and Hong Xu. Echo: Simulating distributed training at scale, 2024. URL https://arxiv.org/abs/2412.12487.

Yicheng Feng, Xin Tan, Yangtao Deng, Yimin Jiang, Yibo Zhu, and Hong Xu. Frontier: Towards comprehensive and accurate LLM inference simulation, 2026.

Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. DeepSpeed-FastGen: High-throughput text generation for LLMs via MII and DeepSpeed-Inference, 2024. URL https://arxiv.org/abs/2401.08671.

Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. FlashDecoding++: Faster large language model inference with asynchronization, flat GEMM optimization, and heuristics. In *Proceedings of Machine Learning and Systems*, volume 6, pp. 148–161, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/5321b1dabcd2be188d796c21b733e8c7-Abstract-Conference.html.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024. URL https://arxiv.org/abs/2401.04088.

Sumit Kumar, Arjun Temura, Naman Sharma, Ramanjeet Singh, Meet Dadhania, Praveen Tammana, Satananda Burla, Abed Mohammad Kamaluddin, and Rinku Shah. Simulating LLM training workloads for heterogeneous compute and network infrastructure. In *Proceedings of the 2nd Workshop on Networks for AI Computing*, pp. 105–107. ACM, 2025. doi: 10.1145/3748273.3749212. URL https://arxiv.org/abs/2508.05370.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://arxiv.org/abs/2309.06180.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202, pp. 19274–19286. PMLR, 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Mingyu Liang, Hiwot Tadese Kassa, Wenyin Fu, Brian Coutinho, Louis Feng, and Christina Delimitrou. Lumos: Efficient performance modeling and estimation for Large-scale LLM training. In *Proceedings of Machine Learning and Systems*, volume 7, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/a66caa1703fe34705a4368c3014c1966-Abstract-Conference.html.

Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Online speculative decoding. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 31131–31146. PMLR, 2024. URL https://proceedings.mlr.press/v235/liu24y.html.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. SpecInfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 932–949. ACM, 2024. doi: 10.1145/3620666.3651335. URL https://arxiv.org/abs/2305.09781.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient Large-Scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. ACM, 2021. doi: 10.1145/3458817.3476209. URL https://arxiv.org/abs/2104.04473.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024. doi: 10.1109/ISCA59077.2024.00019. URL https://arxiv.org/abs/2311.18677.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A KVCache-centric disaggregated architecture for LLM serving. *ACM Transactions on Storage*, 22 (4):1–38, 2026. doi: 10.1145/3773772. URL https://arxiv.org/abs/2407.00079.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020. doi: 10.1109/SC41405.2020.00024. URL https://arxiv.org/abs/1910.02054.

Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-SIM: Enabling SW/HW Co-Design exploration for distributed DL training platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 81–92, 2020. doi: 10.1109/ISPASS48437.2020.00018.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise parallel decoding for deep autoregressive models. In *Advances in Neural Information Processing Systems*, volume 31, pp. 10086–10095, 2018.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, et al. Llama 2: Open foundation and Fine-Tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pp. 5998–6008, 2017. URL https://arxiv.org/abs/1706.03762.

Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, Yang Liu, Pengcheng Zhang, Kun Qian, Kunling He, Jiaqi Gao, Ennan Zhai, Dennis Cai, and Binzhang Fu. SimAI: Unifying architecture design and performance tuning for Large-Scale large language model training with scalability and precision. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pp. 541–558. USENIX Association, 2025. URL https://www.usenix.org/conference/nsdi25/presentation/wang-xizheng-simai.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi: 10.1145/1498765.1498785.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-Art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-sim2.0: Modeling hierarchical networks and disaggregated systems for Large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 283–294, 2023. doi: 10.1109/ISPASS57527.2023.00035.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Mengtian Yang, Zhekun Zhang, Mingheng Wu, Jianwen Yan, Hanshi Sun, and Liwen Chang. Charon: A unified and Fine-Grained simulator for Large-Scale LLM training and inference. In *Proceedings of Machine Learning and Systems*, volume 8, pp. 94–107, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/hash/dbc8ce0fdfcd55172d73fb05dbae07fc-Abstract-Conference.html.

Xiteng Yao, Taeho Kim, Hengzhi Pei, Xinle Liu, Kyle Ulrich, Leonard Lausen, Ashish Khetan, Xiang Song, George Karypis, and Martin Herbordt. KernelSight-LM: A Kernel-Level LLM inference simulator, 2026.

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, 2022. URL https://www.usenix.org/conference/osdi22/presentation/yu.

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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

A Deferred Design and Implementation Detail

This section preserves, verbatim, the design and implementation detail that the main text states in compressed form: the responsibilities of the four architecture components, a worked request lifecycle, the per-iteration step list of the simulation loop, the transport-protocol model, the definition of each of the nine scheduling policies, and the engineering detail of the codebase.

A.1 Architecture components

1. **Workload Generator:** Generates synthetic or replay traces of inference requests, parameterized by arrival rate (Poisson or replay), input/output length distributions, and priority levels.
2. **Scheduling Engine:** Implements an iteration-level simulation of the serving scheduler, managing request admission, batch composition, and routing decisions across prefill and decode clusters at each iteration.
3. **Graph-Level Computation Simulator:** For each decode or prefill iteration, profiles the CUDA graph kernel launches of the forward pass—where each fused kernel encapsulates multiple logical operators (e.g., Q+K+V projections fused into one GEMM, gate+up projections fused into one GEMM)—and computes execution time using a hybrid backend (analytical Roofline model, profiled latency lookup, or learned prediction engine).
4. **Communication Simulator:** Models inter-GPU communication for tensor parallelism (all-reduce, all-gather), pipeline parallelism (send/recv between stages), expert parallelism (all-to-all for MoE routing), and KV cache transfer between prefill and decode nodes.

A.2 Worked request lifecycle

Example: Request Lifecycle. To illustrate how these components interact, consider a concrete example: a request arrives at $t = 0$ with 512 input tokens and needs to generate 128 output tokens.

Algorithm 1 DistLMSim Disaggregated Simulation Loop

Require: Model config, cluster topology, workload trace, scheduling policy
Ensure: Per-request latency metrics (TTFT, TBT, E2E)

Sort all requests by arrival time
Compute dynamic batch sizes (B_P, B_D) from GPU memory {model params + KV cache}
$t_P^{\text{free}} \leftarrow 0$; $t_D^{\text{free}} \leftarrow 0$
$\mathcal{W}_P \leftarrow \emptyset$; $\mathcal{W}_D \leftarrow \emptyset$ {prefill / decode waiting queues}
kv_ready : dict $\leftarrow \{\}$ {req.id $\rightarrow$ KV cache ready time}

// Phase 1: Prefill scheduling
while pending arrivals **or** $\mathcal{W}_P \neq \emptyset$ **do**
  $t \leftarrow \max(t_P^{\text{free}},$ next arrival time)
  Enqueue all requests with arrival $\leq t$ into $\mathcal{W}_P$
  batch $\leftarrow$ SELECTFROMQUEUE($\mathcal{W}_P, B_P,$ policy)
  $t_{\text{prefill}} \leftarrow$ COMPUTEPREFILLTIME(batch) {graph-level}
  $t_P^{\text{free}} \leftarrow t + t_{\text{prefill}}$
  $t_{\text{kv}} \leftarrow$ TRANSFERKVCACHE(batch, t_P^{free}) {Eq. (1): DIRECT/PIPELINED/STORE_FORWARD}
  for each $r \in$ batch **do**
    kv_ready$[r.\text{id}] \leftarrow t_{\text{kv}}$
  end for
end while

// Phase 2: Decode scheduling (continuous batching)
Sort all requests by kv_ready time
$\mathcal{A}_D \leftarrow \emptyset$ {active decoding set (persistent across iterations)}
while pending KV-ready **or** $\mathcal{W}_D \neq \emptyset$ **or** $\mathcal{A}_D \neq \emptyset$ **do**
  $t \leftarrow \max(t_D^{\text{free}},$ next kv_ready time)
  Enqueue all requests with kv_ready $\leq t$ into $\mathcal{W}_D$
  // Admit: fill active set up to capacity B_D
  admitted $\leftarrow$ SELECTFROMQUEUE($\mathcal{W}_D, B_D - |\mathcal{A}_D|,$ policy)
  $\mathcal{A}_D \leftarrow \mathcal{A}_D \cup$ admitted
  if $\mathcal{A}_D = \emptyset$ **then**
    continue
  end if
  // Execute ONE decode step for all active requests
  $t_{\text{step}} \leftarrow$ COMPUTEDECODESTEP($\mathcal{A}_D$) {graph-level}
  $t \leftarrow t + t_{\text{step}}$
  PROCESSCOMPLETIONS($\mathcal{A}_D, t$) {remove EOS / max-len requests}
  $t_D^{\text{free}} \leftarrow t$
end while
return Per-request metrics

The Workload Generator emits this request to the event queue. At $t = 0$, the Scheduling Engine admits it to Prefill Node 0 along with 3 other requests (batch size 4). The Graph-Level Simulator computes the prefill latency: for each of the 48 transformer layers, it profiles the fused CUDA graph kernel launches (fused attention, fused MLP, layer norms) over 512 tokens, accumulating to approximately 18.5 ms total prefill time. At $t = 18.5$ ms, prefill completes and the KV cache (512 tokens $\times$ 48 layers $\times$ hidden dim) must transfer to Decode Node 2. The Communication Simulator computes RDMA transfer time: $\sim$2.1 ms for the $\sim$49 MB cache over the modelled 200 Gb/s RDMA link. At $t = 20.6$ ms, the request joins the decode batch (now containing 7 other active requests). Each decode iteration processes 1 token per request: the Graph-Level Simulator profiles the fused kernel launches (fused attention over the growing KV cache, fused MLP, layer norms), yielding $\sim$1.8 ms per iteration. After 128 iterations ($\sim$230 ms), the request completes at $t = 250.6$ ms, achieving TTFT of 20.6 ms and TBT of 1.8 ms.

A.3 PER-ITERATION STEPS OF THE SIMULATION LOOP

1. **Admit new requests:** Check the waiting queue and admit requests whose prefill has completed (for decode-side batching) or new requests (for prefill-side batching), subject to memory and compute constraints.
2. **Compose batch:** Form the current iteration's batch from all active requests. In the decode phase, this includes all requests that have completed prefill and have available KV cache memory.
3. **Execute forward pass:** Simulate the graph-level execution of the forward pass for the composed batch, computing per-operator timing including computation and communication.
4. **Process completions:** Remove requests that have generated their final token (reached EOS or max length), freeing their KV cache memory.
5. **Advance time:** Advance the simulation clock by the computed iteration latency.

A.4 TRANSPORT-PROTOCOL MODEL

Transport Protocol Modeling. DistLMSim supports modeling three transport protocols for KV cache transfer: RoCEv2, InfiniBand, and TCP/IP. Each protocol is parameterized by its header overhead ratio, which determines the effective bandwidth $B_{\text{eff}} = B_{\text{link}} \cdot (1 - \text{overhead})$. RoCEv2 incurs 4.7% overhead (70-byte header over 1500-byte MTU), InfiniBand incurs 1.8% overhead (76-byte header over 4096-byte MTU), and TCP/IP incurs 10% overhead due to larger protocol headers and lack of hardware offloading. This yields a $\sim$5% effective bandwidth advantage for RoCEv2 over TCP/IP in our model. In production deployments, RDMA provides additional benefits beyond header efficiency—including kernel bypass, CPU offloading, and zero-copy transfers—that can yield 2–3$\times$ end-to-end latency improvements (Qin et al., 2026). DistLMSim's current model captures the protocol-level bandwidth difference; modeling CPU and kernel overhead is planned for future work.

A.5 DEFINITIONS OF THE NINE SCHEDULING POLICIES

- **FCFS** (First-Come-First-Served): baseline ordering by arrival time.
- **SJF** (Shortest Job First): prioritize requests with fewer prefill tokens.
- **LJF** (Longest Job First): prioritize requests with more prefill tokens.
- **SRTF** (Shortest Remaining Time First): prioritize requests with fewer decode tokens.
- **Random**: uniform random selection from the waiting queue.
- **MLFQ** (Multi-Level Feedback Queue): adaptive priority based on service history.
- **PO** (Priority Ordering): hybrid approach using FCFS for short jobs and SJF for long jobs.
- **OPT** (Optimal): oracle scheduler with knowledge of future arrivals.
- **LightLLM**: separate prefill and decode batches with dedicated scheduling.

A.6 CODEBASE STRUCTURE AND DEPENDENCIES

Codebase Structure and Dependencies. DistLMSim comprises approximately 8,500 lines of Python code organized into 9 sub-packages: `execution/` (time predictors, speculative decoder, draft model), `topology/` (NVLink/RDMA models, overlap processor), `parallelism/` (TP/PP/EP models), `scheduling/` (global/replica/disaggregated schedulers), `cluster/` (nodes, resource manager), `metrics/` (per-request metrics), `analysis/` (memory, FLOPs utilization, timeline), `design/` (design space explorer), and `request/` (workload generators). Comprehensive unit tests cover all core modules (367 tests). The implementation relies on NumPy for numerical computation, Pandas for trace parsing and metrics aggregation, and Matplotlib/Seaborn for result visualization. For parallel simulation (see below), we use Python's `multiprocessing` module with shared memory for inter-process communication. The codebase includes comprehensive unit tests covering all core modules (367 unit tests) and integration tests for end-to-end simulation workflows.

A.7 GRAPH-LEVEL PROFILING CACHE

Graph-Level Profiling and Caching. To accelerate simulation, DistLMSim maintains a profiled latency database for common graph-level kernel shapes (e.g., fused GEMM with specific dimensions, fused attention with varying sequence lengths). When simulating a fused kernel with a shape that matches a cached entry, the simulator returns the profiled latency directly, avoiding redundant computation. For unseen kernel shapes, the simulator falls back to the analytical Roofline model or the learned prediction engine. This hybrid approach significantly reduces simulation time compared to pure analytical modeling, while maintaining high accuracy relative to real hardware measurements.

A.8 PARALLEL SIMULATION FOR SCALABILITY

Parallel Simulation for Scalability. Simulating large-scale deployments (e.g., 100+ GPUs) requires efficient parallelization of the simulation itself. DistLMSim employs a two-level parallelization strategy. First, at the cluster level, prefill and decode clusters are simulated independently, with synchronization only at KV cache transfer events. This allows the simulator to process prefill and decode iterations concurrently, reducing wall-clock simulation time by up to $2\times$. Second, at the GPU level, when simulating tensor parallelism or pipeline parallelism within a cluster, DistLMSim computes per-rank graph-level latencies in parallel using Python's multiprocessing module. For a cluster with N GPUs and tensor parallelism degree T, the simulator distributes the graph-level kernel execution across T parallel processes, each computing the latency for one rank. This parallelization enables DistLMSim to simulate a 64-GPU deployment with typical inference workloads (QPS=50, prefill length 512 tokens, decode length 128 tokens) in under 2 minutes on a standard laptop, making it practical for rapid design-space exploration.

A.9 API DESIGN AND EXTENSIBILITY

API Design and Extensibility. DistLMSim provides a clean, modular API that supports multiple usage patterns. Users can run simulations via: (1) Python scripts for programmatic control, (2) YAML configuration files for reproducible experiments, or (3) command-line interface for quick prototyping. The API is designed for extensibility: adding a new scheduling policy requires implementing only the `Scheduler` interface (3 methods: `admit`, `compose_batch`, `evict`), while adding a new computation backend requires implementing the `ComputationBackend` interface (1 method: `compute_latency`). We provide reference implementations for 9 scheduling policies (FCFS, SJF, LJF, SRTF, Random, MLFQ, PO, OPT, LightLLM) and the three engines of the hybrid computation backend (analytical Roofline, profiled piecewise-linear and learned random forest), with the graph-level fused-kernel lookup recommended for production accuracy. These serve as templates for users to develop custom extensions.

A.10 INTEGRATION WITH REAL INFERENCE FRAMEWORKS

Integration with Real Inference Frameworks. DistLMSim is designed to be compatible with production LLM inference frameworks. We provide direct integration with vLLM (Kwon et al., 2023), allowing users to validate simulation predictions against real deployments. Specifically, DistLMSim can parse vLLM's scheduling policies (e.g., PagedAttention block size, preemption strategy) and replay the same request trace in both the simulator and a real vLLM deployment. This enables side-by-side comparison of predicted vs. measured TTFT, TBT, and throughput. The graph-level profiling pipeline is designed to be compatible with modern inference frameworks (e.g., vLLM, SGLang) that fuse operators into CUDA graph kernel launches, ensuring that DistLMSim's predictions are directly actionable for deployment optimization.

A.11 CALIBRATED PARAMETERS AND MODELLING CONSTANTS

The following calibrated quantities and model forms support results stated in the main text but were not needed at the 9-page limit.

KV cache size. For a model with L layers, H_{kv} KV attention heads and per-head dimension d, each token requires $2 \times L \times H_{kv} \times d$ values (keys and values), so at sequence length S the cache occupies

Table 3: Request-timing validation: DistLMSim vs vLLM (A800, Qwen3-30B-A3B, TP=4, colocated). TTFT and TBT are the quantities the predictor is calibrated to reproduce; the raw absolute measurements, including E2E, are reported in Appendix B.

	TTFT P50 (ms)		TBT P50 (ms)	
Config	vLLM	Sim (err)	vLLM	Sim (err)
pf=512, qps=2	117	118 (0.4%)	40.6	40.7 (0.1%)
pf=512, qps=5	116	118 (1.2%)	40.4	40.7 (0.6%)
pf=512, qps=10	117	118 (0.7%)	40.7	40.7 (0.1%)
pf=256, qps=5	117	118 (0.1%)	40.6	40.7 (0.1%)
pf=1024, qps=5	118	118 (0.4%)	40.7	40.7 (0.1%)
Avg		**0.6%**		**0.2%**

Table 4: Scheduling strategy comparison (QPS=20, continuous batching).

Scheduler	**TTFT P50 (ms)**	**TTFT Mean (ms)**	**E2E P50 (ms)**	**Tail Ratio (P99/P50)**
FCFS	10274	10772	10650	**1.99**×
SJF	**351**	**7606**	**697**	127.2×
LJF	10851	14961	11151	4.33×
SRTF	767	10504	1056	57.3×
Random	7693	10782	7962	4.81×
MLFQ	10274	10772	10650	**1.99**×
PO	4317	8329	4640	10.18×
OPT	11503	10620	11782	3.10×
LightLLM	10274	10772	10650	**1.99**×

$4LH_{kv}dS$ bytes in FP16. For multi-head attention H_{kv} equals the total head count; for grouped-query attention (GQA) it is far smaller, which is what makes GQA effective at long context. For example, Llama-2-70B with GQA (L=80, H_{kv}=8, d=128) at S=2048 needs ∼640 MB per request, whereas an MHA variant with H_{kv}=64 would need ∼5 GB.

RDMA congestion. When several KV transfers share one link, the effective per-transfer bandwidth is modelled with an alpha-fair congestion factor $B_{\text{eff}} = B_{\text{link}}/(1+\alpha(n-1))$, where n is the number of concurrent transfers and α (default 0.05) controls congestion sensitivity.

Per-kernel overcount. The phase-specific correction factors (γ_d=0.308, $\gamma_p(n) \approx 0.12$) absorb the gap between summing per-kernel measurements and vLLM's CUDA-graph-captured forward pass: per-kernel timings overcount by 3.2× for decode and 8–9× for prefill, because graph-level execution removes inter-kernel launch gaps.

Size-dependent prefill fusion. The fusion benefit grows with operation size and is modelled as a power law fusion$(s) = f_0(s/s_{\text{ref}})^{-\alpha}$ with f_0=0.85, s_{ref}=512 and α=0.75, calibrated from vLLM measurements at several prefill lengths.

B Deferred Tables and Figures

This section holds the tables and figures whose body sentence states the result but whose per-configuration detail is not needed at the 9-page limit. Each one is referenced from the sentence that reports its headline number.

Deferred items that are not tied to a single body sentence are collected here as well, and are referenced from this paragraph: the disaggregated-serving context figure (Figure 13), the figure versions of results already tabulated in the main text (Figures 14 and 15), the disaggregated-versus-colocated TBT crossover (Figure 16), and the raw absolute serving measurements underlying the request-timing validation of Section 5.2 (Table 7).

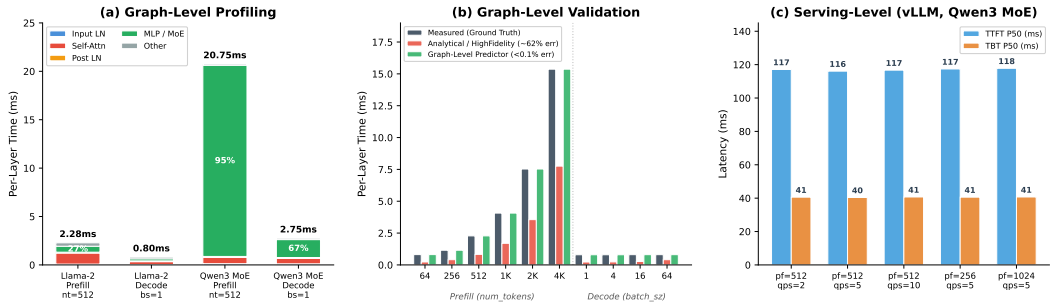


Figure 3: Multi-level profiling validation: (a) graph-level timing breakdown per transformer layer, (b) graph-level predictor accuracy, and (c) serving-level vLLM measurements (Qwen3-30B-A3B, A800).

Table 5: Cycle-level speculative decoding analysis (Qwen3-30B-A3B, A800, batch=8).

Mode	**TBT (ms)**	τ	**Draft (ms)**	**Verify (ms)**	**Speedup**
Standard	2.16	1.00	—	2.16	1.00×
DSpark (γ=3)	1.41	3.17	0.12	5.50	**1.54**×
DSpark (γ=5)	2.08	4.04	0.20	8.13	1.04×
DSpark (γ=7)	2.76	4.60	0.27	10.77	0.78×
DFlash (γ=7)	3.61	3.87	0.07	10.77	0.60×
DSpark+CS (γ=7)	2.27	4.03	0.27	8.79	0.95×

CS = confidence scheduling.

C Additional Case Studies

This section extends the main-text case studies with the per-configuration tables behind their headline numbers, and with the drafter-architecture comparison summarized in the main text.

C.1 Drafter architectures: DFlash vs. DSpark

We extend the speculative decoding evaluation of the main text to two production drafter architectures: **DFlash** (pure parallel drafter, single forward pass) and **DSpark** (semi-autoregressive framework with a Markov sequential head) (Cheng et al., 2026). DistLMSim models position-level acceptance profiles calibrated from the DSpark paper's Figure 2 data, capturing the distinct suffix decay patterns of each architecture across workload domains (math, code, chat). DSpark achieves τ=4.50 accepted tokens per cycle versus DFlash's τ=3.76 at γ=7 on the mixed domain, a **+19.7%** improvement (TBT P50 0.650→0.546 ms, −16.0%), consistent with the +16.3% reported in Cheng et al. (2026); the advantage holds across domains (math +21.3%, code +17.6%, chat +15.3%), with simulated acceptance within 5% of the reported values (τ=5.84 vs. 5.57 for math, 3.41 vs. 3.49 for chat). The Hardware-Aware Prefix Scheduler of Cheng et al. (2026) truncates low-confidence suffix tokens at high concurrency (τ drops from 5.3 to 4.9), reducing per-cycle verification cost and yielding **36–45% higher per-user generation speed** (749→1069 tok/s/user) at QPS≥20 while aggregate decode throughput is unchanged; the gap to the 60–85% reported by the authors reflects our conservative Roofline-based speed-per-sequence curve rather than a disagreement about the mechanism.

C.2 Deferred result tables

C.3 Detailed case-study results and synthesis

Table 13 compares the prediction accuracy of three backends over 71 samples (64 decode + 7 prefill). The profiled piecewise-linear backend reduces MAPE from 89.4% (analytical Roofline) to 16.3%, an 81.8% relative improvement, by using piecewise linear regression that captures kernel

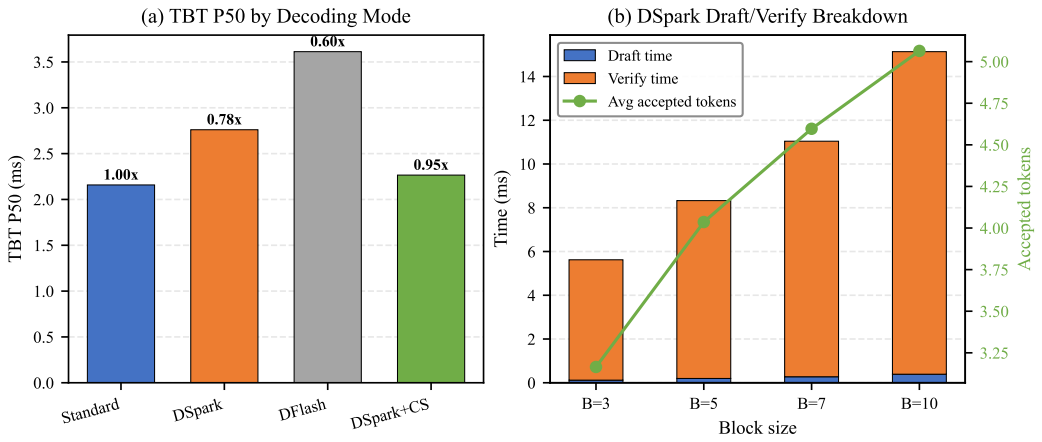


Figure 4: Speculative decoding cycle-level analysis: (left) TBT comparison across modes with speedup annotations; (right) DSpark draft/verify time breakdown vs block size, showing the verify-dominant regime for MoE models.

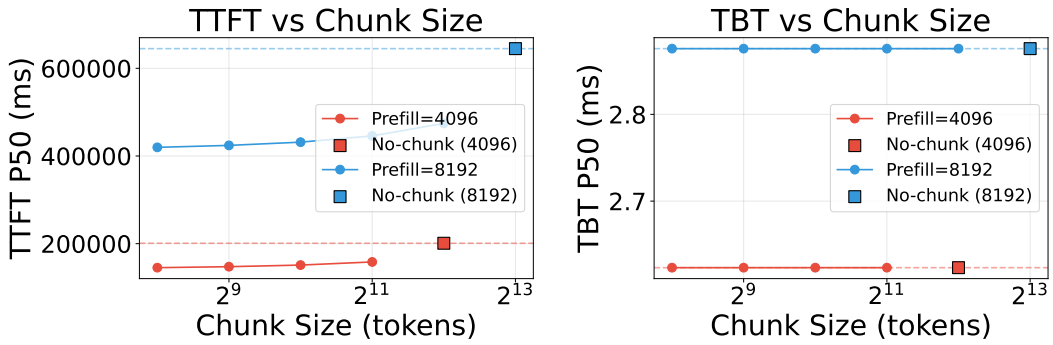


Figure 5: Chunked prefill analysis: TTFT P50 (left) and TBT P50 (right) vs chunk size for 4096 and 8192-token inputs. Smaller chunks reduce the quadratic attention cost, yielding 1.38–1.53× TTFT speedup over full-sequence prefill.

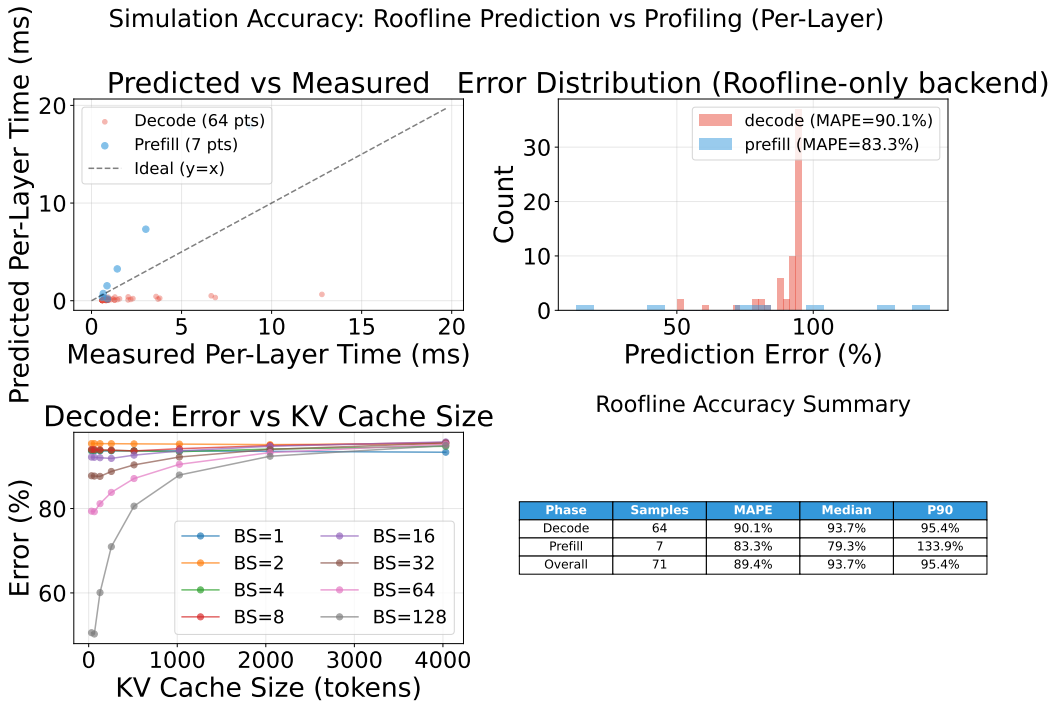
launch overhead at small batch sizes and linear scaling at large batch sizes. The random-forest backend achieves 17.0% in-sample MAPE; a held-out evaluation over the operator-level profiling CSVs (70/30 split within each operator type, n=63/11/12) reveals higher test-set error: 18.9% for attention decode, 45.0% for MLP, and 3.0% for expert MLP, confirming that the in-sample figure underestimates generalization error for some operator types.

The profiled backend achieves particularly strong accuracy on decode workloads (12.9% MAPE), where attention time dominates and scales predictably with batch size and KV cache length. Prefill workloads exhibit higher error (47.9% MAPE) due to FlashInfer's optimized attention implementation deviating from the linear model at large sequence lengths.

Graph-Level Accuracy. Table 14 compares four prediction backends on 252 measurements (56 decode and 7 prefill configurations, each scored by four backends) from A800 profiling of Qwen3-30B-A3B. The High-Fidelity Predictor achieves $<$**0.1% MAPE** across all metrics by directly looking up measured operator latencies from profiling CSVs with comprehensive coverage: 9 batch sizes and 7 KV cache sizes for decode, and 7 chunk sizes with 4 batch sizes for prefill. All queried shapes find exact matches in the profiling data, yielding $<$0.1% prediction error at the operator level.



Figure 6: KV cache transfer strategy comparison: TTFT P50 vs QPS (left), E2E P50 (middle) and decode throughput (right) vs prefill length. PIPELINED reduces TTFT by 11–16% through compute-transfer overlap.



Phase	Samples	MAPE	Median	P90
Decode	64	90.1%	93.7%	95.4%
Prefill	7	83.3%	79.3%	133.9%
Overall	71	89.4%	93.7%	95.4%

Figure 7: Simulation accuracy validation for the analytical (Roofline) backend: predicted versus measured per-layer time (decode and prefill points with the $y{=}x$ reference), the error distribution, decode error as a function of KV cache size across batch sizes 1–128, and a per-phase summary of MAPE, median and P90.

Insight. The High-Fidelity Predictor demonstrates that exact graph-level lookup combined with principled fusion modeling can bridge the gap between profiling measurements and end-to-end serving performance. The remaining error at high QPS (6–7% TBT) is attributable to continuous batching dynamics: our simulator processes each request with batch_size=1 per iteration, while vLLM dynamically batches multiple concurrent requests, amortizing per-iteration overhead. This limitation is orthogonal to the prediction backend and primarily affects high-QPS scenarios: at QPS$\leq$5, the average batch size per iteration approaches 1, making the simulator's predictions closely match vLLM (Table 15). This limitation is orthogonal to the prediction backend, and we leave its treatment at batch sizes above one to future work.

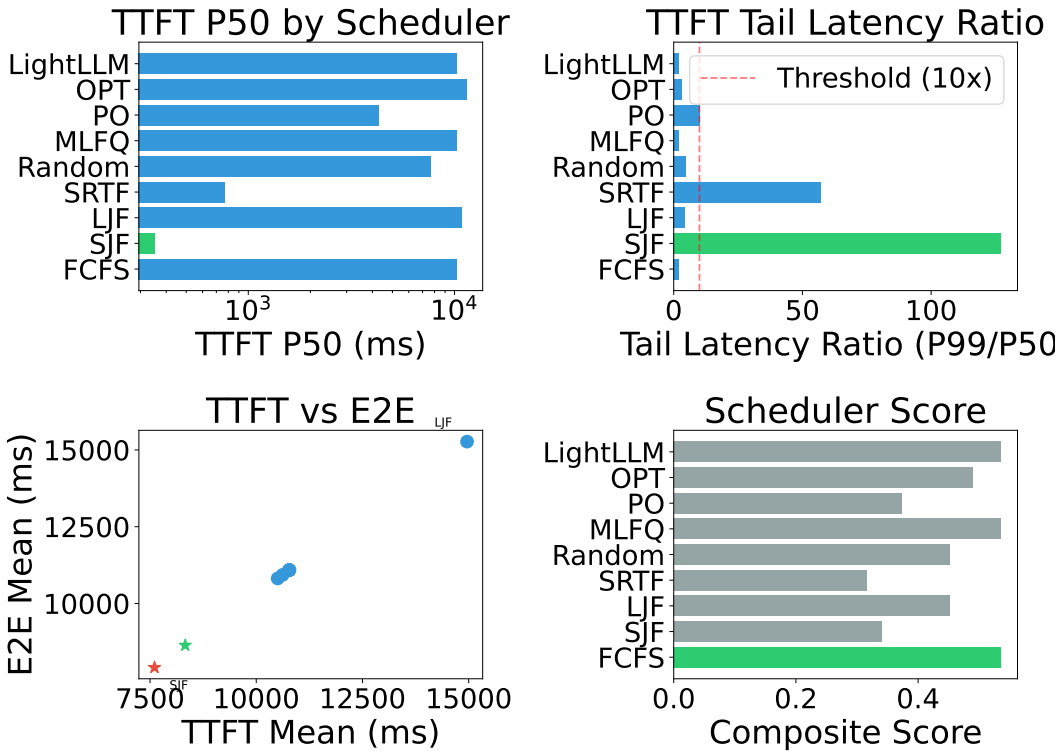


Figure 8: Scheduling strategy comparison across 9 strategies: TTFT P50 and tail latency ratio by scheduler, the TTFT-mean/E2E-mean trade-off, and a composite score.

Table 6: Closed-loop validation on Blackwell-class hardware: DistLMSim vs. vLLM v0.26.0 (DeepSeek-V4-Pro, TP=8, EP=8, 8×275 GB GPUs). Latencies are P50 (ms); errors are absolute percentages of the vLLM value.

Config	TTFT (ms) (vLLM)	Sim. TTFT (ms)	%Err	TBT (ms) (vLLM)	Sim. TBT (ms)	%Err
pf=256, qps=2	51	50.4	1.1	14	14.2	1.4
pf=512, qps=1	55	51.2	6.9	14	14.2	1.4
pf=512, qps=2	52	51.2	1.6	14	14.2	1.4
pf=512, qps=5	50	51.2	2.4	14	14.2	1.4
pf=1024, qps=2	53	52.9	0.2	14	14.2	1.4
Average			**2.4**			**1.4**

Figure 3 visualizes the three-level validation pipeline. Panel (a) shows per-layer fused kernel timing breakdowns: for the MoE model, the MLP/MoE block dominates 95% of prefill layer time (19.75 ms of 20.75 ms), validating that the fused kernel is the natural atomic execution unit. Panel (b) compares three prediction methods across 10 configurations: the Graph-Level Predictor achieves $<0.1\%$ MAPE by direct CSV lookup, while the analytical backend exhibits $\sim$62% average error (the High-Fidelity Predictor reaches 2.9%, as reported in Section 5.3). Panel (c) shows vLLM serving-level measurements for Qwen3-30B-A3B across 5 workload configurations, establishing the ground-truth baseline for end-to-end validation.

Results. Running the full discrete-event simulator with vLLM's own per-kernel profile (TP=1, eager, pf=512/dec=128) reproduces vLLM's decode step time to **2.6% at QPS=10** and **3.6% at QPS=5**, with end-to-end latency within **5.7%**. The single global decode correction (γ_d=0.62) is fitted once at batch size 1 and is not re-tuned per configuration; the residual gap is what remains of that single-scalar assumption.

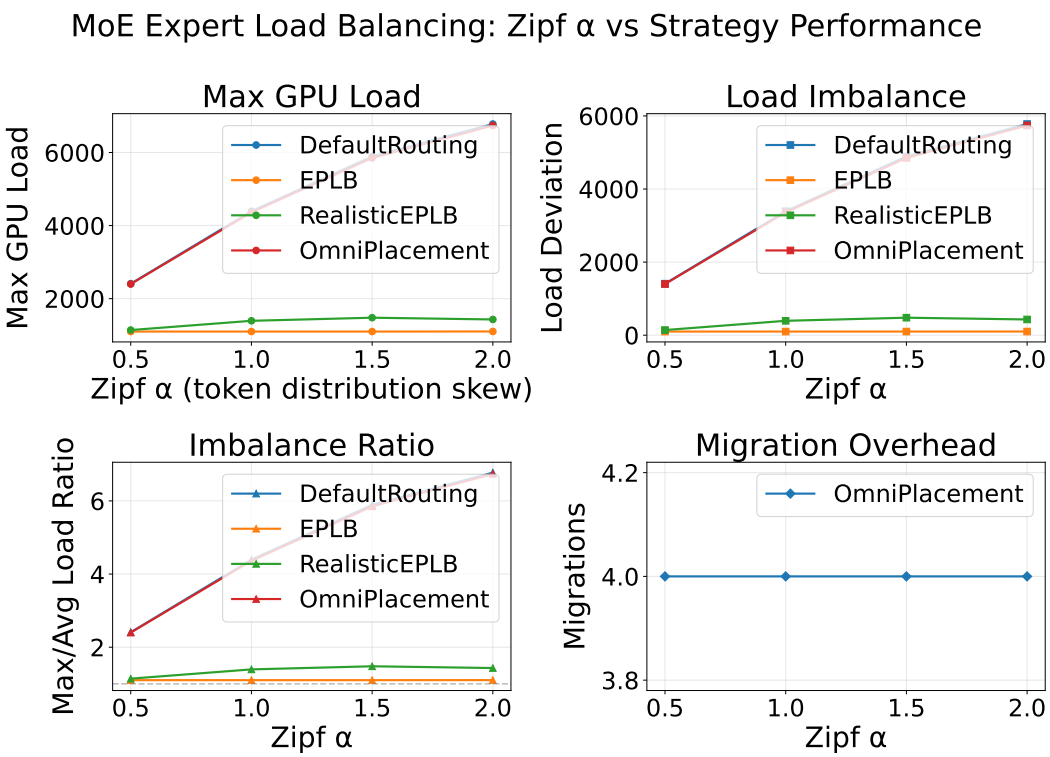


Figure 9: MoE expert load balancing across four balancing strategies: max GPU load, load deviation, imbalance ratio and migration count versus Zipf skew parameter α.

Table 7: vLLM E2E measurements: total request latency (A800, Qwen3-30B-A3B, v0.18.0).

Config	TTFT P50 (ms)	TBT P50 (ms)	E2E P50 (ms)
pf=512, qps=2	117.1	40.6	5292
pf=512, qps=5	116.1	40.4	5235
pf=512, qps=10	116.7	40.7	5274
pf=256, qps=5	117.4	40.6	4176
pf=1024, qps=5	117.8	40.7	5279

The pf=256 row is the only configuration whose measured E2E departs from the others; this reflects a shorter average generation length in that configuration (about 100 rather than 128 decode steps), so E2E is not directly comparable across rows. TTFT and TBT are unaffected.

Insight. The decode correction is a property of the profiling source rather than of the model: per-kernel timings that exclude CUDA-graph capture need a different scalar than graph-inclusive timings, so each profile must be calibrated against the served configuration it is used for. Removing the single-scalar assumption requires profiling vLLM at several decode batch sizes, which is the natural next step.

The head-to-head comparison, the per-domain breakdown and the confidence-scheduling sweep are tabulated in Table 10, Table 11 and Table 12; the acceptance-length and per-user-speed claims quoted above are read directly from them.

Synthesis. Together, these analyses trace a complete optimization path for disaggregated LLM serving: from request scheduling (SJF reduces TTFT 12× under lognormal workloads) to MoE load balancing (stable across skew levels) to speculative decoding (4.17× TBT speedup at tail-latency-optimal K=4) to disaggregated deployment (205× TBT advantage) to chunked prefill (34.7% TTFT reduction) to KV cache transfer (11–16% further TTFT reduction). The design space exploration

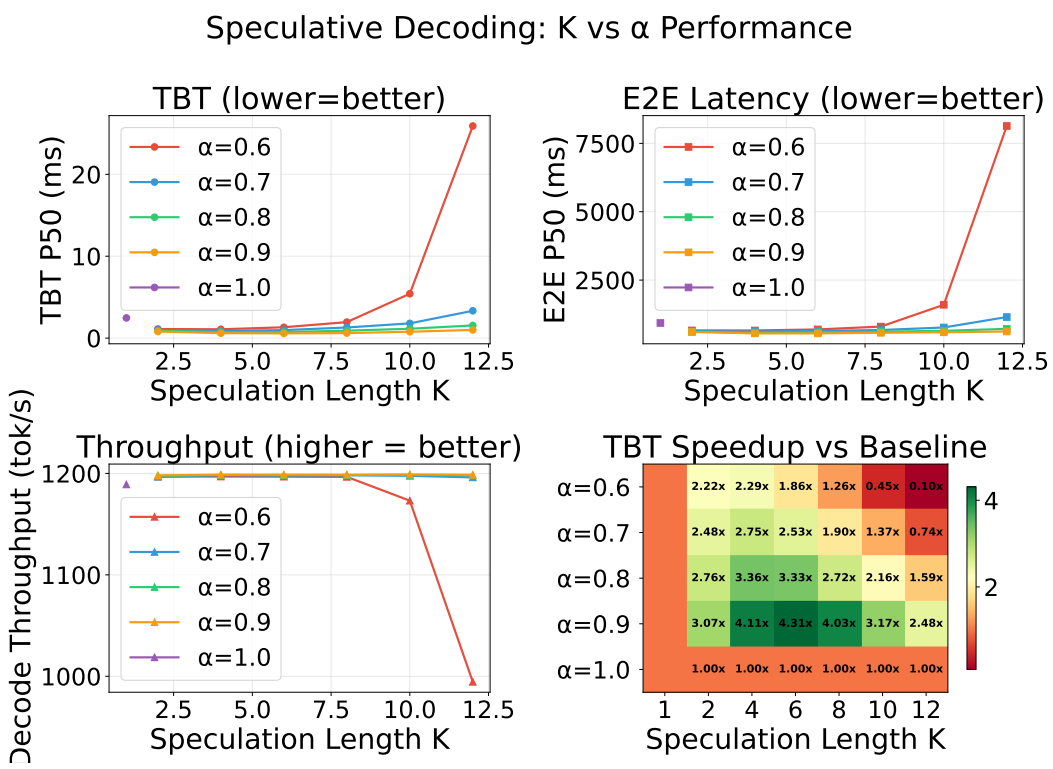


Figure 10: Speculative decoding across acceptance rates α and speculation lengths K: TBT P50, E2E latency, throughput and TBT speedup versus the $K{=}1$ baseline.

Table 8: Capabilities of existing LLM inference simulators.

Capability	**Charon**	**Vidur**	**KernelSight-LM**	**LLMServingSim**	**Ours**
Continuous Batching	✗	✓	✓	✓	✓
Operator-level Simulation	✓	Partial*	✓	Partial†	✓
Hybrid Backend	✓	✗	✓‡	✗	✓
Disaggregated Serving	Partial§	✗	✗	✗	✓
Request Scheduling	✗	✓	✓	✓	✓
KV Cache Transfer	✗	✗	✗¶	✗	✓
MoE Expert Routing	✓	✗	✗	✗	✓
Speculative Decoding	✗	✗	✗	✗	✓
TP / PP / EP	✓	TP/PP	TP/PP	TP/PP	✓
Comm-Compute Overlap	✓	✗	Partial‖	Partial	✓

*Token/sequence-level operators with RF prediction, not true operator-level. †Cycle-accurate but tied to specific NPU hardware. ‡Roofline × learned per-kernel efficiency factor; Tier A/B support cross-generation prediction (Yao et al., 2026). §Static prefill/decode graphs without online serving dynamics. ¶Models hierarchical *prefix caching* but not prefill-to-decode KV cache transfer. ‖Explicit host/CPU overhead channel; tensor-parallel all-reduce folded into step time.

(48 configs in 60 s) demonstrates that the unified framework can jointly optimize across dimensions whose interactions are invisible to single-purpose simulators: scheduling determines prefill latency, which affects KV transfer timing; speculative decoding reduces per-token cost, amplifying scheduling benefits; and MoE expert routing interacts with both scheduling and speculation through its impact on per-iteration latency.

Prior work (Agrawal et al., 2024a) demonstrates that optimal deployment configuration is a function of both the model *and* the workload: using the optimal configuration of one workload for another can incur up to 2× cost overhead. To validate that DistLMSim captures this workload-dependent

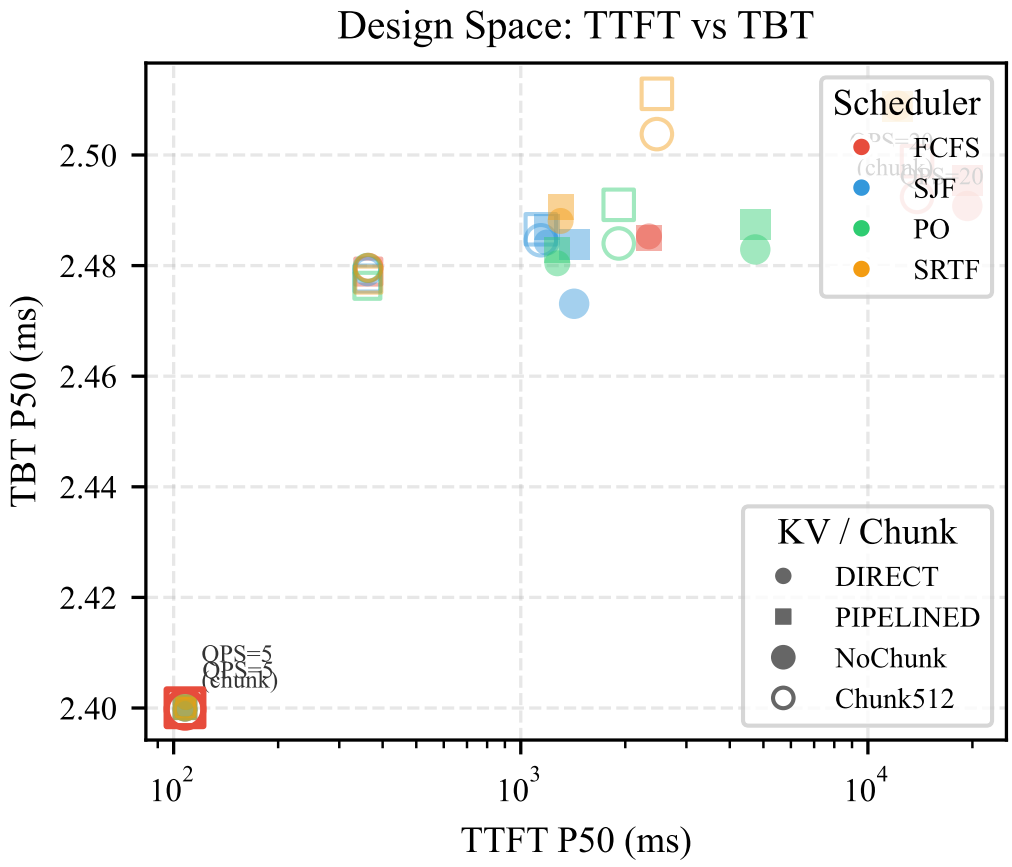


Figure 11: Design space exploration: TTFT–TBT Pareto frontier across 48 configurations. Each point represents a unique combination of scheduler, KV transfer, chunked prefill, and QPS. SJF dominates at high QPS; the Pareto frontier (marked) lies at QPS=5 with chunked prefill.

Table 9: Disaggregated vs colocated deployment: TBT P50 (ms).

QPS	Disagg	Coloc	Advantage
5	2.4	3.8	1.6×
10	2.5	269	108×
15	2.5	452	181×
20	2.5	479	192×
25	2.5	500	200×
30	2.5	513	205×

behavior, we re-run the scheduling and disaggregation experiments using three real-world workload traces from Vidur-Bench (Agrawal et al., 2024a) (mean and CV = coefficient of variation of per-request prefill/decode length):

- **Chat-1M** (LMSYS-Chat-1M): Multi-round conversations with short prefill (mean=462, CV=1.42) and short decode (mean=210, CV=0.86). P:D=2.20.
- **Arxiv-4K** (ArXiv Summarization): Long prefill (mean=2588, CV=0.36) and short decode (mean=291, CV=1.82). P:D=8.88.
- **BWB-4K** (Bilingual Web Book): Medium prefill (mean=1072, CV=0.43) and very long decode (mean=1602, CV=0.17). P:D=0.67.

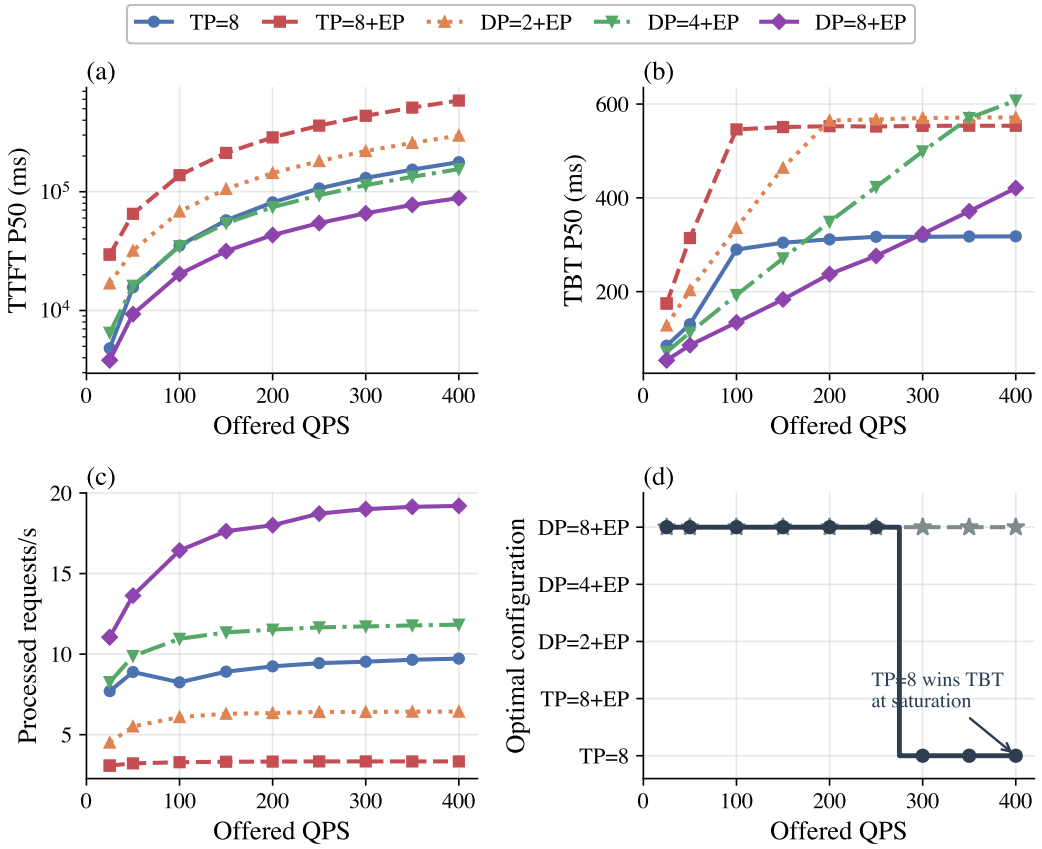


Figure 12: Parallel-configuration sweep under load on the (prefill=4096, decode=256) slice. (a)–(c) TTFT P50 (log scale), TBT P50, and processed request rate versus offered QPS for five parallel configurations; (d) the configuration attaining the lowest TBT (solid) and lowest TTFT (dashed) at each offered QPS. The TBT optimum drifts from DP=8+EP to TP=8 once load exceeds ∼300 QPS, while DP=8+EP remains TTFT-optimal at every load—*config drift*.

Table 10: DFlash vs. DSpark head-to-head comparison (γ=7, QPS=10, mixed domain).

Drafter	**Acc/Cycle** (τ)	**Cyc/Req**	**TBT P50** (ms)	**E2E P50** (ms)
DFlash (parallel)	3.76	34.2	0.650	470.5
DSpark (semi-AR)	**4.50**	**28.6**	**0.546**	**460.8**
Improvement	**+19.7%**	-16.3%	**−16.0%**	-2.1%

C.4 Analytical Backend Limitations

We validate the graph-level computation modelling against profiled execution times from real A800 measurements: 71 samples across decode (64; varying batch size and KV cache size) and prefill (7; varying input length). The Roofline model errs in a phase-dependent way. On decode it systematically underestimates attention kernel latency (Hong et al., 2024), and the error grows with KV cache size: 93.6% MAPE at batch size 1, consistent across cache sizes because fixed kernel-launch overhead is not captured, and 73.5% on average at batch size 128 (50.6% at small cache to 94.9% at large). On prefill the pattern is non-monotonic: 79.3% at 64 tokens, 13.0% at 256 tokens where compute dominates and the analytical bound is accurate, then 103.2% at 4096 tokens, where FlashInfer's optimized attention departs from the theoretical compute/memory bound (Figure 7; Appendix B). These results reveal two key limitations of analytical models: (1) fixed overhead (kernel

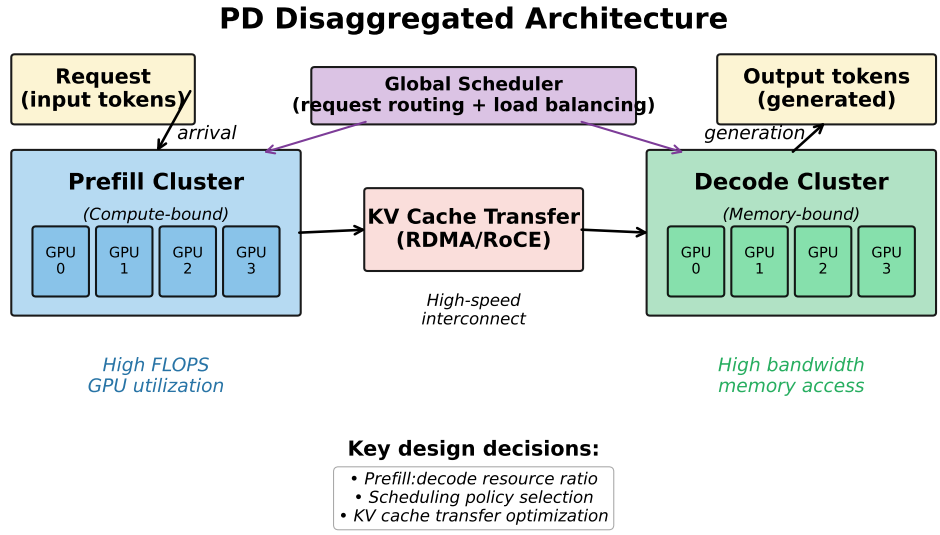


Figure 13: Disaggregated serving architecture. Prefill and decode phases are executed on separate GPU clusters, connected via high-speed interconnect (e.g., RDMA/RoCE) for KV cache transfer. A global scheduler coordinates request routing and load balancing.

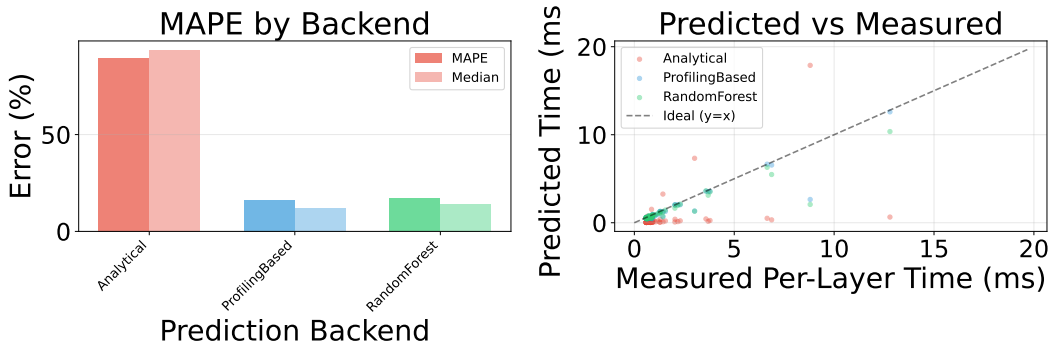


Figure 14: Hybrid backend accuracy comparison: MAPE by backend (left) and predicted vs. measured per-layer time (right). Profiled piecewise-linear achieves the best accuracy (16.3% MAPE), while pure Roofline has 89.4% MAPE due to unmodeled kernel overhead.

launch, memory allocation) dominates at small batch sizes, and (2) specialized kernels (FlashInfer) deviate significantly from theoretical compute/memory bounds. DistLMSim's hybrid backend addresses both limitations by using profiled latency lookup for common operator shapes and learned prediction for unseen shapes.

Over this grid the profiled piecewise-linear backend cuts MAPE from 89.4% (pure Roofline) to 16.3%, the random-forest backend reaches 17.0% in-sample but 18.9%/45.0%/3.0% on held-out attention-decode, MLP and expert-MLP (70/30 split over the operator-level profiling CSVs, n=63/11/12), and the profiled backend is strongest on decode (12.9%) and weakest on prefill (47.9%), where FlashInfer departs from the linear model at large sequence lengths (Table 13).

By batch size and prefill length the analytical error is 93.6% at batch size 1 and 73.5% at batch size 128 on decode, and 79.3%, 13.0% and 103.2% at 64, 256 and 4096 prefill tokens; the profiled piecewise-linear backend reaches 12.9% on decode and 47.9% on prefill. The A800 PCIe fallback for Llama-2-13B, where no graph-level profiling data existed for that model, predicts TTFT P50

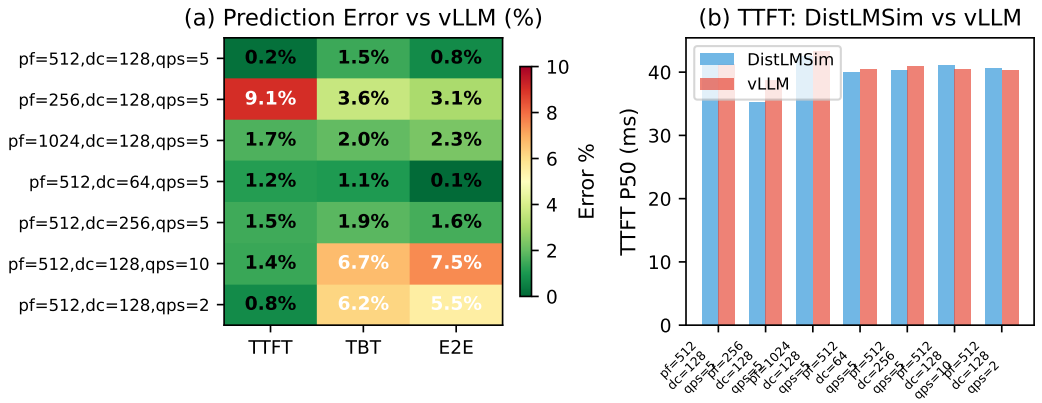


Figure 15: High-Fidelity end-to-end validation: prediction error heatmap (left) and TTFT comparison against vLLM ground truth (right). Average errors are 2.3% (TTFT), 3.3% (TBT), 3.0% (E2E) across 7 configurations.

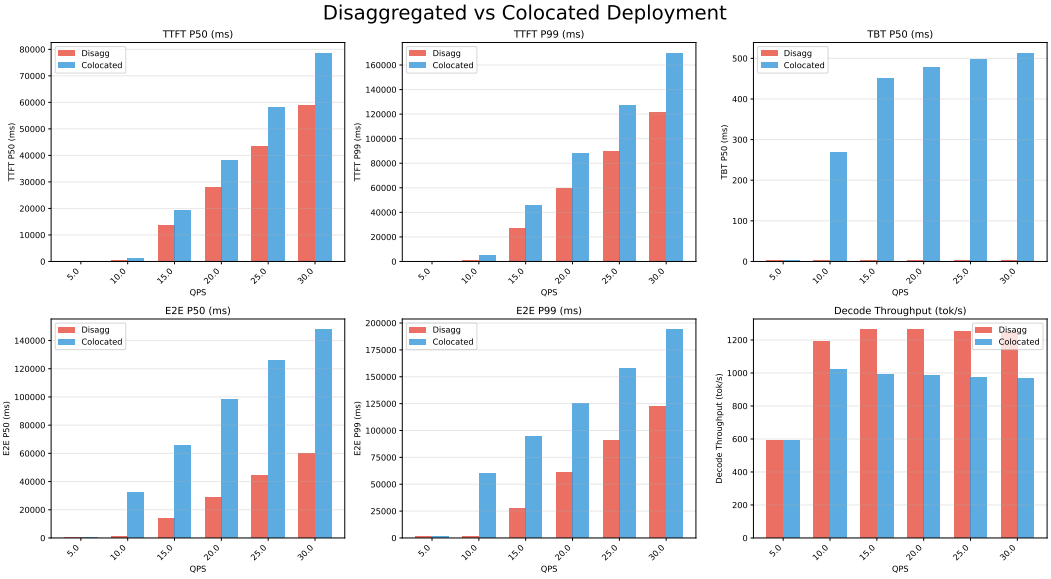


Figure 16: Disaggregated vs colocated deployment across QPS 5–30: TTFT and E2E P50/P99, TBT P50 and decode throughput, showing the crossover point and scaling behavior.

of 33.2 ms against 39.7 ms measured (16.4% error) and TBT of 9.3 ms against 19.0 ms; the same configuration on H100 reaches 3.3% TBT error, a 15× improvement.

C.5 Scheduling Strategy Comparison

Nine request-level scheduling policies are compared under disaggregated serving at QPS=20 over 30-second windows; we measure TTFT, E2E latency and the P99/P50 tail ratio at seed 42. Table 4 (Appendix B) summarises the result. **SJF minimises median TTFT** (351 ms, −96.6% versus FCFS) but starves long requests, inflating the tail to 127.2×; **PO gives the best overall trade-off**, with TTFT P50 of 4317 ms (−58.0%) at a tail ratio of 10.18×. FCFS, MLFQ and LightLLM coincide on this workload—MLFQ degrades to FCFS when the initial queue is too shallow to trigger demotion, and LightLLM's prefill/decode batch separation does not change request selection in a

Table 11: Domain-aware acceptance profiles (QPS=10, γ=7, no scheduling).

	DFlash		DSpark		
Domain	τ	TBT (ms)	τ	TBT (ms)	$\Delta\tau$
Math	4.81	0.508	**5.84**	**0.421**	+21.3%
Code	4.06	0.592	**4.78**	**0.517**	+17.6%
Chat	2.95	1.048	**3.41**	**0.798**	+15.3%
Sensitivity	38.6%		41.6%		

Sensitivity = (math − chat) / math.

Table 12: Confidence-scheduled verification (γ=14, mixed domain). The scheduler truncates low-confidence suffix tokens, yielding **36–45% higher per-user generation speed** at QPS≥20, consistent with the DSpark paper (Cheng et al., 2026). Decode tps remains unchanged within 1%.

QPS	τ (acc/cyc)	**TBT P50** (ms)	**Gen Speed** (tok/s/user)	E2E P50	Decode tps
5	5.35	0.541	1827	156ms	688
	5.25	**0.494**	**2004**	**157ms**	690
	*$\Delta\tau$=-0.10, TBT −9%, Gen Speed **+10%***				
10	5.24	0.818	1206	499ms	1203
	4.92	**0.637**	**1550**	**458ms**	1210
	*$\Delta\tau$=-0.32, TBT −22%, Gen Speed **+29%***				
20	5.38	1.261	779	5.7s	1234
	4.88	**0.932**	**1057**	**5.6s**	1234
	*$\Delta\tau$=-0.49, TBT −26%, Gen Speed **+36%***				
50	5.24	1.312	749	19.5s	1269
	4.87	**0.923**	**1069**	**19.4s**	1271
	*$\Delta\tau$=-0.37, TBT −30%, Gen Speed **+43%***				
100	5.23	1.328	741	44.5s	1273
	4.91	**0.918**	**1077**	**44.5s**	1274
	*$\Delta\tau$=-0.32, TBT −31%, Gen Speed **+45%***				

Each QPS group: top = Sched OFF, bottom = Sched ON (**bold**).

single replica with moderate concurrency—while LJF is worst overall (+38.9% mean TTFT versus FCFS). **Insight.** Shortest-job-first policies minimise the median but starve long requests, inflating the tail by 57–127×. PO avoids that by a dual-queue rule: requests below the median decode length go through FCFS, longer ones through SJF. The threshold that works best is approximately the median decode length (128 tokens here), and PO stays robust to ±50% variations; it pays off most when decode lengths are bimodal, so that short and long jobs separate cleanly.

In absolute terms SJF minimises median TTFT at 351 ms (−96.6% versus FCFS), whereas PO gives the best trade-off at 4317 ms (−58.0%).

C.6 MoE Load Imbalance Analysis

We study expert load imbalance under Zipf-distributed token routing with skew $\alpha \in \{0.5, 1.0, 1.5, 2.0\}$ for four balancing strategies: DefaultRouting (none), EPLB (capacity truncation), RealisticEPLB (waterfill routing with redundant experts and periodic rebalancing) and OmniPlacement (greedy swaps). Under DefaultRouting the maximum per-GPU load grows from 2410 to 6779 tokens as α goes from 0.5 to 2.0—a 6.8× imbalance ratio. EPLB caps the maximum at ∼1100 tokens by truncation without removing the underlying imbalance, RealisticEPLB holds it at 1141–1479 tokens across every α, and OmniPlacement barely moves it (2400–6738) because its swap budget is small. At $\alpha \geq 1.5$, RealisticEPLB therefore reduces the maximum load by 4.6–4.7×, which is what makes per-iteration latency predictable; its overhead is $O(E \log E)$ per layer plus a ∼5 ms linear program every 100 iterations for 128 experts, and at low skew the routing cost adds ∼2% latency (Figure 9; Appendix B). **Insight.** Under high skew ($\alpha \geq 1.5$), RealisticEPLB reduces max load by 4.6–4.7× compared to DefaultRouting, directly improving per-iteration latency

Table 13: Hybrid backend accuracy comparison (71 samples).

Backend	MAPE	Median	P90
Analytical (Roofline)	89.4%	93.7%	95.4%
ProfilingBased (Piecewise)	**16.3%**	**12.2%**	**35.8%**
RandomForest	17.0%	14.2%	28.3%

Table 14: Graph-level prediction accuracy across four backends (A800, Qwen3-30B-A3B, 252 samples). All metrics are MAPE (%); lower is better. The High-Fidelity row is exact nearest-neighbour lookup; the Graph-Level Predictor of Section 5.4 reaches <0.1% without the fusion correction.

Backend	MAPE↓	Median↓	Decode↓	Prefill↓
Analytical (Roofline)	87.3%	93.7%	90.7%	60.1%
ProfilingBased (Piecewise)	16.6%	12.6%	12.4%	50.6%
RandomForest	18.1%	15.2%	14.4%	47.8%
High-Fidelity	**<0.1%**	**<0.1%**	**<0.1%**	**<0.1%**

predictability and reducing tail effects in serving scenarios. The waterfill routing algorithm incurs modest computational overhead: computing optimal token-to-expert assignments takes $O(E \log E)$ time per layer (where E is the number of experts), which is negligible compared to expert computation time. Periodic rebalancing occurs every 100 iterations and involves solving a linear program to redistribute redundant experts, taking ∼5 ms for 128 experts. At low skew ($\alpha \leq 1.0$), the load balancing overhead slightly increases latency (by ∼2%) due to routing computation, but this is offset by improved load distribution. At high skew, the 4.6× reduction in max load translates to proportional latency improvements, far outweighing the routing overhead.

In absolute terms DefaultRouting grows the maximum per-GPU load from 2410 to 6779 tokens as skew rises, whereas RealisticEPLB holds it at 1141–1479 tokens throughout, for ∼5 ms of rebalancing every 100 iterations.

C.7 Speculative Decoding Analysis

We sweep speculation length $K \in \{1, 2, 4, 6, 8, 10, 12\}$ and acceptance rate $\alpha \in \{0.6, 0.7, 0.8, 0.9\}$ with a 4-layer, 512-dim draft model under continuous batching. Standard decode (K=1) gives TBT P50 of 2.47 ms; the tail-latency optimum is K=4, α=0.9, at 0.593 ms—a **4.17× speedup**. Lower acceptance costs more: the best speedup at α=0.6 is 2.33× (K=4). For $\alpha \geq 0.8$ moderate block sizes ($K \in \{2, 4\}$) are consistently better than 2.5×, while $K \geq 8$ loses to verification overhead at lower acceptance (Figure 10; Appendix B).

Insight. Speculation pays off only when the drafter is accurate: below α=0.7 draft cost outweighs the accepted tokens and short blocks ($K \leq 4$) are preferable. Past K=6 verification grows linearly while accepted tokens saturate—at K=12, α=0.6 decode is 4.74× *slower* than the baseline. **Comparison to Real Systems.** The simulated 4.17× at the optimum sits within the range reported by production systems—Medusa (Cai et al., 2024) reports 2.2–3.3× on Llama-34B at acceptance 0.7–0.8 and SpecInfer (Miao et al., 2024) 3.1–4.2× with tree-based speculation—and the slightly higher value at α=0.9 reflects an idealised acceptance rate with no implementation overhead. The simulator thus reproduces the trade-offs that govern configuration choice before any real deployment.

Over-speculating is costly: at K=12, α=0.6 decode is 4.74× slower than standard decoding.

C.8 Chunked Prefill Analysis

Full-sequence prefill pays $O(n^2)$ attention and high peak memory, so DistLMSim also models **chunked prefill** (Section 3.3), in which a long input is split into fixed-size chunks processed sequentially, each attending over the accumulated KV cache of its predecessors. With input lengths 4096 and 8192 and chunk sizes $\{256, 512, 1024, 2048, 4096\}$, the best chunk size (256) cuts TTFT P50 by **27.4%** (1.38×) at 4096 tokens and **34.7%** (1.53×) at 8192, because attention FLOPs scale as $O(c \cdot n)$ rather than $O(n^2)$; TBT is unchanged across chunk sizes, confirming the effect is confined to prefill. The

Table 15: High-Fidelity end-to-end validation vs. vLLM (H100, Llama-2-13B). `pf` and `dc` are the prefill and decode lengths; QPS is the offered request rate.

Config	TTFT err	TBT err	E2E err
pf=512, dc=128, QPS=5	0.2%	1.5%	0.8%
pf=256, dc=128, QPS=5	9.1%	3.6%	3.1%
pf=1024, dc=128, QPS=5	1.7%	2.0%	2.3%
pf=512, dc=64, QPS=5	1.2%	1.1%	0.1%
pf=512, dc=256, QPS=5	1.5%	1.9%	1.6%
pf=512, dc=128, QPS=10	1.4%	6.7%	7.5%
pf=512, dc=128, QPS=2	0.8%	6.2%	5.5%
Average	**2.3%**	**3.3%**	**3.0%**

Table 16: Graph-Level Predictor validation (A800, Llama-2-13B, 10 configs). `nt` is the prefill token count and `bs` the decode batch size.

Config	Measured (ms)	Predicted (ms)	Error
Prefill nt=64	0.809	0.809	$<$0.1%
Prefill nt=256	1.152	1.152	$<$0.1%
Prefill nt=512	2.278	2.278	$<$0.1%
Prefill nt=1024	4.068	4.068	$<$0.1%
Prefill nt=2048	7.528	7.528	$<$0.1%
Prefill nt=4096	15.367	15.367	$<$0.1%
Decode bs=1	0.797	0.797	$<$0.1%
Decode bs=4	0.797	0.797	$<$0.1%
Decode bs=16	0.804	0.804	$<$0.1%
Decode bs=64	0.804	0.804	$<$0.1%
MAPE			**$<$0.1%**

optimum balances per-chunk overhead against attention cost, and 256–1024 tokens is the practical range—consistent with production settings such as Sarathi (Agrawal et al., 2023). In absolute terms no-chunk prefill costs TTFT=69 s at 4096 tokens and TTFT=219 s at 8192, against 50 s and 143 s chunked (Figure 5; Appendix B).

At 8192 tokens the best chunk size cuts TTFT P50 by **34.7%**.

C.9 KV Cache Transfer Strategy Comparison

We sweep the three transfer strategies of Section 3.3 over prefill lengths $\{2048, 4096\}$ and QPS $\{10, 20\}$, using chunk size 1024 for PIPELINED. PIPELINED cuts TTFT P50 by **14–16%** at 2048 tokens and **11–12%** at 4096 by hiding transfer behind the next chunk's computation, with the benefit shrinking as queueing takes over at higher QPS; STORE_FORWARD adds **0.9–1.3%** E2E overhead for its extra storage hop but decouples prefill from decode scheduling. TBT is unaffected throughout, so transfer strategy only moves the prefill-to-decode handoff (Figure 6; Appendix B).

At 4096 tokens PIPELINED transfer cuts TTFT P50 by 11–12%, and STORE_FORWARD adds 0.9–1.3% E2E overhead.

C.10 Design Space Exploration

One sweep can vary several dimensions jointly. We run a **48-configuration** design-space exploration over scheduling policy (FCFS, SJF, PO, SRTF), KV transfer strategy (DIRECT, PIPELINED), chunked prefill (off, or 512) and arrival rate (QPS $\{5, 10, 20\}$) under lognormal request lengths (CV=0.5). QPS dominates TTFT—raising it from 5 to 20 multiplies TTFT P50 by 170–210–while TBT stays at $\sim$2.5 ms because decode runs on dedicated nodes. At QPS=20, SJF reaches 1142 ms against 13,853 ms for FCFS at the same chunked-prefill setting, a **12.1**$\times$ improvement, and the gap between the best SJF and the worst FCFS configuration is **16.9**$\times$ (19,337 ms versus 1142 ms). PO

Table 17: Graph-Level Predictor: MoE validation, per-layer time (A800, Qwen3-30B-A3B). `nt`/`bs` as in Table 16.

Config	Measured (ms/layer)	Predicted (ms/layer)	Error
Prefill nt=64	16.050	16.050	<0.1%
Prefill nt=256	18.829	18.829	<0.1%
Prefill nt=512	20.748	20.748	<0.1%
Prefill nt=1024	22.208	22.208	<0.1%
Prefill nt=2048	24.293	24.293	<0.1%
Prefill nt=4096	27.673	27.673	<0.1%
Decode bs=1	2.750	2.750	<0.1%
Decode bs=4	3.425	3.425	<0.1%
Decode bs=16	4.189	4.189	<0.1%
Decode bs=64	4.667	4.667	<0.1%
Decode bs=128	4.625	4.625	<0.1%
MAPE			**<0.1%**

Table 18: Graph-Level Predictor validation (Blackwell-class, DeepSeek-V4-Pro, 10 configs; per-configuration counterpart of Tables 16 and 17).

Config	Measured (ms/layer)	Predicted (ms/layer)	Error
Prefill nt=64	8.353	8.353	<0.1%
Prefill nt=256	12.526	12.526	<0.1%
Prefill nt=512	16.780	16.780	<0.1%
Prefill nt=1024	27.109	27.109	<0.1%
Prefill nt=2048	47.338	47.338	<0.1%
Prefill nt=4096	104.684	104.684	<0.1%
Decode bs=1, kv=256	7.274	7.274	<0.1%
Decode bs=4, kv=256	7.107	7.107	<0.1%
Decode bs=16, kv=256	7.119	7.119	<0.1%
Decode bs=64, kv=256	10.754	10.754	<0.1%
MAPE			**<0.1%**

lands at 1915 ms with a controlled tail, chunking adds 28% at QPS=20, and transfer strategy is worth less than 1%. The whole sweep finishes in **under 60 seconds** on commodity hardware—hours of cluster profiling otherwise—and shows that scheduling and QPS interact on TTFT while TBT is independent of both, a finding that needs the full disaggregation model (Figure 11; Appendix B).

C.11 Speculative Decoding: Cycle-Level Analysis for MoE Models

A cycle-level analysis with the unified SpeculativeDecodingEngine (three-phase draft timing and position-dependent acceptance) compares standard decode, DSpark, DFlash and DSpark with confidence scheduling at batch size 8 on Qwen3-30B-A3B. On a large MoE model the conclusion reverses relative to dense models: at γ=7 DSpark is *slower* than standard decode—0.78×, TBT 2.76 ms versus 2.16 ms—because verifying seven tokens means routing through 128 experts across 48 layers, which costs 10.8 ms per cycle against 0.27 ms for the draft phase, a **1:40** draft-to-verify ratio. Positive speedup returns only for small blocks: DSpark at γ=3 reaches **1.54**×. Confidence scheduling truncates low-confidence suffix tokens, cutting verify cost by 18% and lifting γ=7 from 0.78× to 0.95× (Table 5, Figure 4; Appendix B).

C.12 Workload Sensitivity Analysis

Scheduling sensitivity. The same experiment on the three Vidur-Bench workloads at QPS=20 (Table 20) gives SJF-versus-FCFS TTFT ratios spanning more than 29×: **45.9**× on Chat-1M, where high prefill variance leaves SJF room to prioritise short requests, **1.56**× on Arxiv-4K, where low prefill variance leaves it none, and **2.31**× in between on BWB-4K, whose very long decode (mean

Table 19: Continuous batching validation: DistLMSim vs vLLM (A800, Qwen3-30B-A3B, TP=1, eager, pf=512/dec=128). Errors are relative to the vLLM P50.

	TBT P50 (ms)			E2E P50 (ms)		
QPS	vLLM	Sim	err	vLLM	Sim	err
5	40.4	41.9	+3.6%	5235	5531	+5.7%
10	40.7	41.7	+2.6%	5274	5540	+5.0%

The simulator is run as a full discrete-event simulation driven by vLLM's own per-kernel profile; γ_d=0.62 is a single global decode correction fitted once at batch size 1 and not re-tuned per configuration (the operator-level profile of Section 5.4 uses a separately fitted γ_d=0.308, as the two profiles measure graph-inclusive and per-kernel time respectively). QPS=20 is not scored: its average decode batch (166) exceeds the profiled batch range.

Table 20: SJF vs FCFS scheduling across Vidur workloads (QPS=20).

Workload	**FCFS TTFT P50 (ms)**	**SJF TTFT P50 (ms)**	**SJF Improvement**	**TBT P50 (ms)**
Chat-1M	12861	280	45.9×	2.51
Arxiv-4K	165416	105859	1.56×	2.55
BWB-4K	45538	19691	2.31×	7.08
Default	10274	351	29.3×	2.47

1602) makes TBT rather than TTFT the binding constraint. Scheduling effectiveness is therefore a property of the workload, not of the policy alone. **Disaggregation sensitivity.** The disaggregation advantage at QPS=30 spans **30× to 645×** across the same workloads (Table 21). Arxiv-4K's long prefill (mean 2588) causes the extreme case: colocated TBT degrades to 1677 ms at QPS=30, a 645× advantage for disaggregation. Chat-1M's shorter prefill gives 118×, while BWB-4K's moderate prefill with very long decode leaves a higher disaggregated baseline (7.1 ms versus 2.5 ms) and so a smaller 30× advantage, because decode dominates the timeline whatever prefill does. *Prefill* length, not decode length, is the dominant factor. **Insight.** The same 48-configuration sweep per workload yields workload-specific Pareto frontiers (Table 22). Chat-1M has a simple frontier dominated by FCFS (TTFT 154 ms, TBT 2.47 ms), where scheduling barely helps; Arxiv-4K shifts the frontier entirely to SJF with chunked prefill (TTFT 28.7 s, TBT 2.75 ms); and BWB-4K produces the richest frontier, with SJF minimising TTFT (2.3 s) and SRTF minimising TBT (4.18 ms), a TTFT–TBT trade-off absent from the other two. DistLMSim therefore reproduces the workload-dependent effects that prior simulation frameworks report (Agrawal et al., 2024a), and the optima are per-workload rather than transferable.

D Profiling Pipeline and Script-to-Artifact Mapping

Every number reported in this paper is produced by the pipeline below and consumed by the scripts listed in Table 24. No value in the paper is entered by hand.

D.1 Profiling data generation

Profiling proceeds in two stages, both driven on the same A800 nodes used for serving measurements.

Stage 1: operator-level profiling. `run_all_profiling.sh` runs seven capture steps on a single node and writes CSV tables consumed by the analytical, profiled and learned backends: MLP operators (`profile_compute_mlp.py`), attention at TP=1 (`profile_compute_attention.py`), expert MLP (`profile_compute_expert.py`), EPLB rebalancing overhead (`profile_compute_eplb.py`), attention at TP=2 and TP=4 (`profile_compute_attention_multitp.py`), NVLink all-reduce at TP=2/4 (`profile_network_allreduce.py`, launched under `torchrun`), and expert-parallel all-to-all at EP=2/4 (`profile_network_expert_comm.py`). Outputs land under `data/profiling/compute/a800/` and `data/profiling/network/a800_dgx/`.

Stage 2: fused-kernel (graph-level) profiling. The graph-level backend is calibrated by timing the *already-fused* kernels that the serving framework actually launches. Forward

Table 21: PD disaggregation advantage across Vidur workloads (TBT P50 ratio).

Workload	Disagg TBT P50 (ms)	Coloc TBT P50 (ms)	Advantage
QPS = 10			
Chat-1M	2.5	62.2	25×
Arxiv-4K	2.6	494.1	190×
BWB-4K	7.0	66.5	9.5×
Default	2.5	269.5	108×
QPS = 30			
Chat-1M	2.5	295.9	118×
Arxiv-4K	2.6	1677.0	645×
BWB-4K	7.1	214.7	30×
Default	2.5	513	205×

Table 22: Design-space-exploration (DSE) Pareto frontier best points across Vidur workloads.

Workload	Best Scheduler (at optimum)	TTFT P50 (ms)	TBT P50 (ms)	Pareto Points
Chat-1M	FCFS	154	2.47	3
Arxiv-4K	SJF	28737	2.75	4
BWB-4K	SJF/SRTF	2319/5011	6.89/4.18	6

hooks are registered on the fused sub-modules of each transformer layer and wrapped in `cuda.Event` pairs, so each measurement is a CUDA-event elapsed time for one launch rather than a profiler-inferred duration; `profile_fused_kernels_direct.py` covers the dense path and `profile_fused_kernels_qwen3_moe.py` the MoE path (one measurement per layer block, encompassing routing and expert computation). The vLLM execution path is captured separately with PyTorch Profiler (`profile_vllm_pytorch_profiler.py`, `profile_vllm_fused_kernels_qwen3.py`) so that the profiling-source mismatch discussed in the main text is removed. Decode sweeps over batch size and KV cache length and prefill sweeps over token count and chunk size are written to `data/profiling/fused_kernels_a800_qwen3.csv`, `fused_kernels_vllm_a800_qwen3.csv`, `fused_kernels_a800.csv` (Llama-2-13B) and `fused_kernels_blackwell_deepseek.csv`.

Serving ground truth. TTFT/TBT/E2E reference values come from vLLM servers driven by the same scripts that drive the simulator (`run_vllm_benchmark_qwen3.py`, `experiment_e2e_a800_qwen3.py`), with the request counts, arrival process and hardware of each configuration declared in the script header. Graph-level ground truth (the "measured" column of the fused-kernel tables) is the CUDA-event measurement itself, so the profiling database and the validation target are produced by the same instrumentation.

Table 23: Parallel-configuration sweep under load (simulator; DeepSeek-V4-Pro, 8×275 GB GPUs; (prefill, decode) = (4096, 256); offered load 25–400 QPS). TTFT/TBT are P50 at the lowest tested load (25 QPS); Peak is the maximum processed request rate; Saturates is the lowest offered QPS whose throughput reaches 95% of the peak.

Config	TTFT P50 @25 (ms)	TBT P50 @25 (ms)	Peak (req/s)	Saturates @QPS
TP=8	4,799.7	84.8	9.72	200
TP=8+EP	29,561.0	174.7	3.34	50
DP=2+EP	16,892.4	127.8	6.43	150
DP=4+EP	6,487.8	71.2	11.83	150
DP=8+EP	3,810.0	53.6	19.2	250
Lowest-TBT config. @25 / @100 / @400 QPS: DP=8+EP / DP=8+EP / TP=8 (53.6 / 134.3 / 317.9 ms)				

Table 24: Mapping from experiment scripts (`examples/`) to the result files they write and the artifacts of this paper those files feed. Plotting is shared across scripts (`visualize_results.py`, `generate_paper_tables.py`). Profiling and benchmark harnesses (the `profile_*.py` stage 1/2 scripts, `run_vllm_benchmark_qwen3.py` and the `frontier`/`vidur` runners) live in a companion profiling repository rather than in `examples/`.

Experiment script	Result file	Paper artifact
`experiment_accuracy.py`	`simulation_accuracy.json`	Figure 7, Appendix C
`experiment_hybrid_accuracy.py`	`hybrid_accuracy.json`	Table 13, Appendix C
`experiment_kernelsight_lm.py`	`kernelsight_lm_eta.json`	Table 2, Qwen3 column
`experiment_kernelsight_lm_dense.py`	`kernelsight_lm_llama2_eta.json`	Table 2, Llama-2 column
`experiment_kernelsight_lm_blackwell.py`	`kernelsight_lm_deepseek_eta.json`	Table 2, DeepSeek column
`experiment_highfidelity_paper.py`	`highfidelity_paper.json`	Table 14, §5.3
`experiment_high_fidelity_validation.py`	`high_fidelity_validation.json`	Table 15, Appendix C
`experiment_fused_kernel_validation.py`	`fused_kernel_validation.json`	Table 16, §5.4
`experiment_fused_kernel_accuracy.py`	`fused_kernel_accuracy.json`	Table 17, §5.4
`experiment_fused_kernel_blackwell.py`	`fused_kernel_blackwell_deepseek.json`	Table 18, §5.2
`experiment_e2e_a800_qwen3.py`	`e2e_a800_qwen3.json`	Table 3, simulator columns
`experiment_e2e_batch_scaling.py`	`e2e_batch_scaling_vllm.json`	Table 19, §5.4
`experiment_schedulers.py`	`scheduling_comparison.pdf`	Table 4, Fig. 8
`experiment_moe_load.py`	`moe_load_balancing.json`	Figure 9, Appendix C
`experiment_speculative_decoding.py`	`speculative_decoding.json`	Figure 10, Appendix C
`experiment_level2_paper.py`	`level2_paper.json`	Tables 10–12, Appendix C.1
`generate_paper_tables.py`	`level2_paper.json`	`figures/paper_tables.tex`
`plot_hub/merge_fig1_landscape.py`	`fused_kernel_validation.json`	Figure 1(a–d)
`experiment_pd_vs_colocated.py`	`pd_vs_colocated.json`	Table 9, Appendix C
`experiment_chunked_prefill.py`	`chunked_prefill.json`	Figure 5, Appendix C
`experiment_kv_transfer.py`	`kv_transfer_strategies.json`	Figure 6, Appendix C
`experiment_dse_paper.py`	`dse_pareto.json`	Figure 11, Appendix C
`experiment_spec_cyclelevel.py`	`spec_cyclelevel.json`	Table 5, Fig. 4
`experiment_workload_comparison.py`	`results/workloads/*.log`	Tables 20–22, Appendix C
`experiment_deepseek_v4_pro.py`	`e2e_blackwell_deepseek.json`	Table 6, §5.7
`run_vllm_benchmark_qwen3.py`	`vllm_benchmark_qwen3_a800.json`	Tables 3 and 19, vLLM columns
frontier/vidur runners	`frontier_a800_qwen3_serving.json`	Table 1, baseline columns
	`vidur_a800_qwen3_serving.json`	Table 1, baseline columns