Dooly: Configuration-Agnostic, Redundancy-Aware Profiling for LLM Inference Simulation

Joon Ha Kim **Geon-Woo Kim** **Anoop Rachakonda** **Daehyeok Kim**
{jhk.james1110, gwkim, anooprac, daehyeok}@utexas.edu
The University of Texas at Austin

Abstract

Selecting the optimal LLM inference configuration requires evaluation across hardware, serving engines, attention backends, and model architectures, since no single choice performs best across all workloads. Profile-based simulators are the standard tool, yet they hardcode their operation set to a specific configuration and re-profile every operation from scratch, making exploration prohibitively expensive. This cost stems from a missing structural understanding: every input dimension of each operation is fixed by the model configuration or determined by the incoming request. Many model-configuration values (*e.g.,* head size, layer count) recur across models, so the same operation runs in many configurations; a single sweep over the request-dependent dimensions can serve them all. We present Dooly, which exploits this structure to achieve *configuration-agnostic, redundancy-aware* profiling. Dooly performs a single inference pass, labels each input dimension with its origin via taint propagation, and selectively profiles only operations absent from its latency database; stateful operations such as attention are isolated by reusing the serving engine's own initialization code, eliminating manual instrumentation. From this database, it builds latency regression models that serve as a drop-in backend for existing simulators. Across two GPU platforms, three attention backends, and diverse model architectures, Dooly achieves simulation accuracy within 5% MAPE for TTFT and 8% for TPOT while reducing profiling GPU-hours across 12 models by 56.4% compared to the existing profiling approach. We have open-sourced Dooly at `https://dooly-project.github.io`.

1 Introduction

Deploying an LLM inference system requires navigating a configuration space spanning hardware, serving engines [25, 43], attention backends [42, 14], model architectures, and workload characteristics. No single configuration performs optimally across this space: the best (model, backend) pair can invert across workload types and sequence lengths, making pre-deployment evaluation indispensable (§2.1). Profile-based simulators [3, 5, 11, 12, 26, 18] have become the standard tool for this evaluation, collecting GPU execution latencies to predict end-to-end serving performance without running a live system.

Today's simulators, however, suffer from two limitations that hinder this exploration. First, they hardcode the operation set and the inputs to each operation, locking each simulator to a specific configuration; any stack update or new architecture requires manual rework to restore coverage. Second, because simulators cannot recognize when two configurations invoke the same underlying computation, they re-profile each from scratch. Profiling four similar-sized models in Vidur [3] takes 27.7 wall-clock hours on an A100 GPU, and this cost grows linearly with each new model or backend.

Neither limitation is inherent to profiling; both stem from a missing structural understanding of LLM inference, which we develop through two observations. First, every input dimension of each

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

operation derives from one of two sources: model configuration parameters (hidden dimension, number of heads) or incoming request characteristics (token count, batch size). Second, many model-configuration values recur across models, so a large fraction of operations are shared across configurations: models with the same attention geometry invoke identical operations regardless of tokenizer differences. Thus, labeling each dimension with its origin lets a profiler detect equivalent operations across configurations and eliminate redundant profiling runs.

Building on these observations, we present Dooly, a framework for *configuration-agnostic, redundancy-aware* LLM inference profiling. Dooly dynamically extracts all operations invoked by a target model through a single inference pass with a dummy prompt, labeling each input dimension with its origin. It then profiles only operations absent from its latency database: from these labels, it computes *signatures* that canonically identify each operation by its model-derived dimensions, allowing it to skip equivalent operations across configurations. From the latency database, Dooly trains per-signature regression models that existing simulators can adopt as a drop-in profiling backend; for end-to-end simulation, Dooly reuses the serving engine's own scheduler to reproduce exact scheduling decisions and queries these regression models as it walks the model's call graph.

Two key techniques make this practical across diverse LLMs and serving stacks. First, *taint propagation* labels each input dimension by tainting values at the serving engine's model-configuration and request-processing entry points and propagating them through the inference pass; the resulting labels let Dooly compute operation signatures from model-derived dimensions only, enabling correct deduplication. Second, *hierarchical context resolution* handles stateful operations, such as attention kernels that require KV cache memory and request metadata, by reusing the serving engine's own initialization code to reconstruct the necessary runtime state.

We implement Dooly for vLLM and SGLang, and evaluate it across two hardware platforms (NVIDIA A100 and H100), three attention backends (FlashInfer [42], FlashAttention [14], TritonAttention [34]), and 12 model architectures spanning dense [6, 39], MoE [37, 22], and heterogeneous model families [9]. Dooly predicts vLLM's serving latency within 5% MAPE for TTFT and 8% for TPOT, and reproduces batch scheduling decisions within 0.5%. Across 12 models profiled with three attention backends, it reduces total profiling GPU-hours by 56.4% compared to the existing profiling approach.

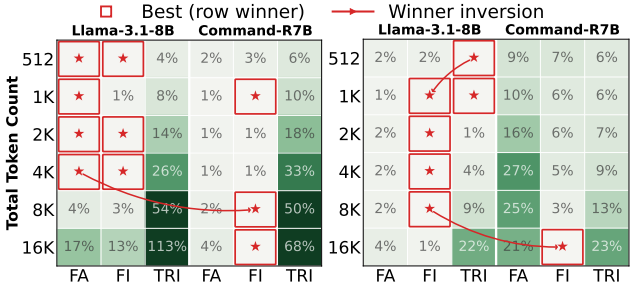
2 Background and Motivation

2.1 Configuration Space in LLM Inference Deployment

Deploying an LLM inference system requires navigating a large configuration space whose choices substantially affect performance, typically characterized by Time To First Token (TTFT), Time Per Output Token (TPOT), and throughput [10, 44, 32, 4]. We organize this space along four key axes:

(1) Hardware (H): GPU selection (*e.g.,* NVIDIA A100 or H100), each with distinct memory capacity and compute throughput; **(2) System (S):** The software runtime stack, primarily defined by the *serving engine* [25, 43], which manages request batching and KV-cache memory, and the *attention backend* [14, 42], which provides GPU-optimized attention kernels; **(3) Model (M):** The model architecture, including hidden dimensions, layer counts, and structural variants such as Mixture-of-Experts (MoE); and **(4) Workload (W):** Runtime load characteristics, including request arrival rates, prompt lengths, and batch sizes.

No single configuration performs optimally across the full $H \times S \times M \times W$ space. We demonstrate this empirically using two similar-sized models, Llama-3.1-8B [27] and Command-R7B [13], on an A100 GPU; Figure 1 reports TTFT for prefill-heavy and TPOT for decode-heavy workloads, where total token count denotes prompt size and cached context size, respectively. Even at $\leq$1K tokens, where all (model, backend) combinations lie within 10% of each other, the winner shifts by workload: Llama-3.1-



(a) Prefill-Heavy Workload **(b)** Decode-Heavy Workload

Figure 1: Performance variation across configurations. TTFT and TPOT are reported for prefill and decode workloads. Red lines mark winner inversions; ties are within 0.5% of the best (FA: FlashAttention, FI: FlashInfer, TRI: TritonAttention).

8B with FlashAttention leads prefill-heavy workloads while Llama-3.1-8B with FlashInfer and TritonAttention leads decode-heavy ones. Above 4K tokens, the prefill-heavy winner changes to Command-R7B, whose interleaved sliding-window attention confines most layers to a 4K context window and caps per-layer attention cost as sequences grow.

The decode-heavy winner is more stable: Llama-3.1-8B with FlashInfer leads until it, too, inverts to Command-R7B at 16K tokens. At 16K tokens, choosing the wrong model alone, even with its best backend, costs 13% in prefill latency; the worst (model, backend) pairing falls 113% behind the optimum. Backend choice within each (model, workload) pair also matters: each backend implements a distinct mix of compile-time kernel specialization and runtime kernel selection, yielding different latency profiles under identical model configurations. The optimal deployment must thus be discovered empirically.

2.2 Limitations of Profile-Based LLM Simulators

Since exhaustively running each configuration on a live system to measure performance is impractical, profile-based simulators [3, 5, 11, 12, 26, 18] have become the standard tool for estimating LLM performance before deployment. These simulators treat the GPU as a black box, collecting the execution latency of operations and modules via PyTorch's Kineto profiler [33], which records GPU activity through NVIDIA's CUPTI [1]. We define an *operation* as a low-level CPU call that dispatches a GPU kernel (*e.g.,* `aten::linear`), and a *module* as a coarser-grained PyTorch block composed of multiple operations (*e.g.,* `RowParallelLinear`).

Two limitations undermine the cross-configuration exploration these simulators are meant to enable.

Static configuration dependencies. Today's simulators *fix* the system and model configuration axes by hardcoding both the operation set to profile and the arguments used to invoke each operation, tying themselves to a specific software stack. Vidur [3] and Revati [5], for example, are tied to Sarathi-Serve [4] and FlashInfer [42], while LLMServingSim [11, 12] is tied to HuggingFace or vLLM. Each relies on a per-family model template that maps parameters such as *head_dim* and *num_heads* onto its predefined operator set. Consequently, the simulators cover only a single column of the configuration matrix (Figure 1), and any software stack update (*e.g.,* a changed module signature or a new fused operation) invalidates existing profiles. Expanding coverage requires manual effort: *e.g.,* patching Vidur to support vLLM required understanding its parallelism initialization logic, integrating its custom fused kernels, and rewriting module callables to match Vidur's expected structure. The unsustainability of this approach is reflected in GitHub issues for Vidur [15, 36, 35], LLMServingSim [19], and Apex [21].

High profiling overhead. Despite their restricted configuration scope, today's simulators incur substantial profiling overhead. To construct an accurate latency model, each model requires an exhaustive sweep over a grid of batch sizes, sequence lengths, and context lengths up to the model's maximum context window, repeated for every new model or backend.

Consider a practitioner deploying a chat application and evaluating FlashInfer on vLLM across four sub-8B models: Llama-3.1-8B-Instruct, BioMistral-7B-Instruct, Qwen2-7B-Instruct, and DeepSeek-R1-Distill-Qwen-7B. Vidur profiles 11 modules per model: attention modules are swept over sequence lengths, batch sizes, and context lengths, while non-attention modules are swept over sequence lengths alone. Across the four models, this totals 44 module profiling runs and 27.7 wall-clock hours on a single A100 GPU. While additional hardware reduces wall-clock time, the linear scaling with model count directly undermines the rapid iteration that simulation is meant to enable.

3 Overview of Dooly

We propose Dooly, a framework that explores the full configuration space while reducing profiling cost in LLM inference simulators by reusing profiles across configurations without manual setup. We build on two structural observations: (1) every input dimension derives from either model configuration or the incoming request, and (2) model-configuration values recur across models, so the same operation runs in many configurations. Rather than relying on manually defined templates or hardcoded operation sets, Dooly *dynamically extracts* the operations invoked for a given target model and configuration through a single inference pass with a dummy prompt, and *selectively profiles* only operations not already in its latency database.

3.1 Challenges

Realizing dynamic extraction and selective profiling in practice presents two non-trivial challenges.

C1: Determining the semantic meaning of input parameters. Sweep-range selection and redundancy detection both require knowing what each tensor dimension represents, yet raw tensor shapes do not encode this. Consider an `aten::matmul` with traced input shapes [60, 2048] and [2048, 4096]: 60 is a request-dependent batch size, while 2048 and 4096 are fixed by the model configuration. Without this distinction, sweeping all dimensions explodes the profiling space, and dimension-level comparison across configurations fails: Qwen-3-8B and Llama-3-8B share identical attention geometry (32 query heads, 8 KV heads, 128 head dim), making their attention operations equivalent, yet tokenizer differences yield different token counts that flag them as distinct.

C2: Isolating stateful operations. To profile per-operation latency, each operation must be runnable in isolation. Existing tools such as the PyTorch Profiler extract the call graph of operations and their tensor inputs, but do not capture the runtime state that certain operations depend on. Attention operations, for example, require attention metadata, KV cache memory addresses, and request-to-slot mappings, none of which are recorded in the trace. Without this context, such operations cannot be replayed standalone even when all tensor inputs are available, and thus cannot be profiled without running the entire model. Manually reconstructing this context demands deep knowledge of the serving engine's internals and is time-consuming and error-prone.

3.2 Key Ideas

To address these challenges, Dooly introduces two key ideas.

I1: Assigning semantic meaning to tensor dimensions via taint propagation. Since every input dimension derives from either model configuration or the incoming request, Dooly taints values at the serving engine's model-configuration and request-processing entry points and propagates them through a single inference pass, labeling each dimension as model-derived (fixed across workloads) or request-derived (variable). These labels enable principled sweep-range selection and correct identification of redundant operations across configurations.

I2: Isolating context-dependent operations via hierarchical context resolution. Serving engines already perform the necessary context initialization during their own startup sequence; Dooly reuses this code directly, reconstructing stateful operations in their proper execution environment without manual effort. When an operation cannot execute in isolation, Dooly falls back to its enclosing parent module in the PyTorch hierarchy, which already holds the initialized state. This ensures every operation is profiled at the finest granularity possible: standalone when self-contained, or within its enclosing module when context is required.

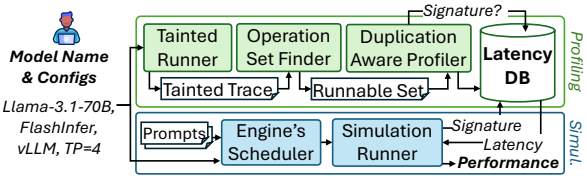


Figure 2: Dooly architecture.

Workflow. Figure 2 illustrates how Dooly incorporates the two ideas. The user provides a target model name and a set of configuration tuples to explore. In the profiling phase, the Tainted Runner (§4) performs a single inference pass on a dummy prompt and annotates the resulting PyTorch execution trace with taint labels for each tensor dimension, producing a *tainted trace*. The Operation Set Finder (§5) parses this trace into a hierarchical call graph and resolves each operation to the finest profiling granularity. The Duplication-Aware Profiler (§6) then uses the semantic labels to determine per-operation sweep ranges and eliminate redundant operations across configurations, populating the latency database with only the necessary measurements. The simulation phase (§7.1) reuses the serving engine's own scheduler to drive end-to-end simulation; for each scheduled request, the simulation runner walks the operation call graph and queries per-signature regression models trained on the latency database.

4 Tainted Runner

The Tainted Runner executes a single inference pass and produces a semantically annotated trace in which every tensor dimension is labeled with its origin. It operates in two phases: **source tainting** intercepts the serving engine's entry points to initialize taint labels for model configuration

and request parameters, and **taint propagation** tracks how these labels flow through arithmetic and tensor operations across the pass, an approach inspired by information flow tracking in prior systems [16, 8, 29]. Dooly also emulates the parallelism setup and replaces collective operations with no-op hooks so that multi-GPU configurations can be traced on a single GPU.

4.1 Source Tainting

Serving engines expose model- and request-specific parameters through well-defined dataclasses (*e.g.,* `SchedulerOutput` and `ModelConfig` in vLLM), whose member values Dooly recursively *taints*. Model configuration values receive a MODEL_CONFIG taint; request-dependent values receive NUM_TOKS for token counts and NUM_REQS for the number of requests in the batch. These source taints initialize the *global taint registry*, which maps concrete values back to their taint labels.

4.2 Taint Propagation

When arithmetic combines tainted scalars from different sources, the result's taint records all contributing origins (*e.g.,* `batch_size` $\times$ `head_dim` records both NUM_TOKS and MODEL_CONFIG). Table 1 formalizes the propagation rules, which extend naturally to tensor dimensions since shape computations are arithmetic over dimension sizes. Dooly categorizes tensor operations as *dimension-mapping* or *dimension-preserving*.

Table 1: Scalar taint propagation rules. Taints (τ) are either untainted ($\perp$), a base label ($l \in L$ where $L = \{\text{MODEL_CONFIG}, \text{NUM_TOKS}, \text{NUM_REQS}\}$), or composite ($\text{MIX}(H)$) where H maps values to base taints.

Operation	Result	Rule
$\perp \otimes \tau$	τ	Absorption
$\tau \otimes \tau$	τ	Preservation
$l_1 \otimes l_2$ $(l_1 \neq l_2)$	$\text{MIX}(\{l_1, l_2\})$	Conflict
$\text{MIX}(H) \otimes l$	$\text{MIX}(H \cup \{l\})$	Extend
$\text{MIX}(H_1) \otimes \text{MIX}(H_2)$	$\text{MIX}(H_1 \cup H_2)$	Merge

Dimension-mapping operations create new tensors or rearrange existing shapes (*e.g.,* `torch.empty`, `torch.reshape`). These require explicit hooks because they directly establish the taint-to-dimension relationship. For tensor creation (*e.g.,* `torch.empty`), Dooly tags each output dimension with the taint of its corresponding scalar argument. For tensor manipulation operations (*e.g.,* `torch.reshape`), the input parameters explicitly define the permutation and merge/split behavior of dimensions. When dimensions merge, Dooly applies the Conflict, Extend, or Merge rules to assign a MIX taint with H recording the value-to-taint mapping. When dimensions split, it recovers the original taints by consulting H. Newly tainted dimensions are registered in the global taint registry, and Dooly consults this registry to resolve inferred dimensions (*e.g.,* `reshape(-1)`).

Dimension-preserving operations maintain input shapes and thus preserve dimension taints. Rather than instrumenting each operation individually, Dooly intercepts these via PyTorch's Dispatcher. This lets Dooly observe operations *after* dispatch but *before* kernel execution, providing full access to input parameters and output tensor shapes. This dispatch-level interception covers all operations without per-operation hooks, including custom kernels in attention backends and fused operations in serving engines. For each intercepted operation, Dooly applies shape-matching heuristics: output dimensions that match an input dimension inherit its taint, and any new dimensions are resolved via the global taint registry.[1] Dooly assigns a $\perp$ taint to unresolved dimensions, which can later be recovered via reshape operations if they merge with known dimensions.

Trace annotation. Together, these rules tag every dimension with its source taint as the inference pass executes. Dooly records these taints via the PyTorch Profiler [2, 20], capturing them at multiple levels of the PyTorch [31] stack: module hierarchy, high-level operations, and low-level ATen operators (see Appendix C for hook details). This ensures all operations are semantically labeled within the call graph, including custom kernels that bypass the dispatcher. We evaluate the effect of each rule in §7.3.

5 Operation Set Finder

The tainted trace from §4 contains every operation invoked during the model's forward pass, but not all of them can be profiled in isolation. The Operation Set Finder resolves this by parsing the trace

[1] Taint ambiguity arises when multiple dimensions share the same concrete value but carry different taints. This is rare in practice and is resolved by retracing with a different prompt; see Appendix B for details.

into a hierarchical call graph, testing each operation for standalone execution, and constructing the minimal runnable set covering all kernel calls in the pass.

5.1 Hierarchy Constructor

Dooly parses the tainted trace into a tree capturing the module-to-operation nesting of the model; the hierarchy is derived from the recorded durations of modules, operations, and kernels in the trace. Each node corresponds to an `nn.Module` invocation (*e.g.,* `model.layers.0.self_attn`), an ATen operator, or a CUDA kernel recorded within that module. Because transformer architectures repeat the same layer structure, Dooly prunes the tree across layers: structurally identical subtrees (*e.g.,* `layers.0.self_attn` and `layers.1.self_attn`) are collapsed into a single canonical subtree, reducing the resolution workload to one representative per repeated module.

5.2 Testing for Runnability

The hierarchy provides a fallback structure for resolution: if a leaf operation cannot run in isolation, Dooly falls back to its parent module. For any module to execute standalone, Dooly must both reconstruct the necessary execution context and generate valid input tensors; it does so through execution context emulation and taint-driven input generation, respectively.

Execution context emulation. Serving engines establish context in two stages. At initialization, the engine binds module-level state such as KV cache memory and metadata builders for attention modules. At each forward pass, it generates per-request context based on the batch's characteristics (sequence lengths, batch size). This per-pass context is backend-dependent: different attention backends (*e.g.,* FlashInfer or FlashAttention) impose distinct memory layouts and computation strategies, requiring different metadata and execution contexts. Dooly faithfully emulates both stages by reusing the serving engine's internal APIs, injecting the constructed context (*e.g.,* via `set_forward_context()` in vLLM) to enable isolated module execution.

Input generation. Dooly uses the taint annotations to determine the shapes and values of input tensors. For stateless modules (*e.g.,* linear layers, embeddings), tensors are generated directly from the tainted argument values in the trace. For stateful modules (*e.g.,* Attention, Mamba, MoE layers), the same strategy applies, but context setup is delegated to the initialized engine. In both cases, the following rules govern taint-driven input generation: MODEL_CONFIG-tainted dimensions remain fixed, as they define the model's architecture. NUM_TOKS- or NUM_REQS-tainted dimensions are set to small values to verify runnability with minimal overhead. MIX-tainted dimensions are recalculated by substituting the workload-dependent component while preserving the model-derived component recorded in H; for example, a flattened dimension $\text{NUM_TOKS}(269) \times \text{MODEL_CONFIG}(40) = 10760$ becomes $1 \times 40 = 40$. Untainted dimensions (typically size-1 vector dimensions) are left unchanged to preserve operator semantics. The Duplication-Aware Profiler (§6) reuses this input generation strategy when sweeping across workload configurations.

Bottom-up resolution. With the hierarchy, context emulation, and input generation in place, Dooly performs a bottom-up traversal to find the runnable set. Resolution starts at tainted leaf nodes, skipping untainted leaves introduced by dispatch mechanics (§4.2). Starting at the finest granularity maximizes deduplication in the profiling stage (§6), since fine-grained operations are more likely to be shared across models and configurations.

Each leaf is tested for standalone execution by (1) importing its operator(s) from the relevant libraries and (2) running them with the generated input tensors. Leaves that pass both steps are added to the runnable set; those that fail trigger a fallback to the parent node. Common failure cases are context-dependent operations (*e.g.,* `unified_attention`), whose required runtime state cannot be recovered from the trace alone. Stateless parents undergo the same import-and-run test as leaves, while stateful parents additionally require context emulation. Dooly identifies stateful modules by consulting the serving engine's registries (*e.g.,* vLLM's `AttentionGroup` registry). If the parent cannot be resolved, the fallback continues upward until either a node succeeds or the tree root is reached.

When a parent resolves successfully, its children are removed from the runnable set to prevent double-counting. This coarsening reduces deduplication opportunities, but it occurs only when isolated execution fails, since leaf-level resolution is always attempted first. The resulting *runnable set* is forwarded to the Duplication-Aware Profiler (§6).

5.3 Runnable Set Characteristics

The runnable set produced by this resolution process is heterogeneous in granularity. Standalone-runnable operations (*e.g.,* `aten::linear`) appear as operator-level entries. Operations whose isolated execution fails (typically context-dependent kernels or Triton implementations that cannot be imported standalone) are absorbed into their nearest enclosing module and appear as module-level entries carrying the necessary execution context. This mixed-granularity construction lets Dooly produce a complete runnable set for any architecture PyTorch can execute, including those whose dominant operators are not exposed as importable kernels.

Sufficiency of a single trace. A single dummy-prompt trace suffices for full operation coverage across both prefill and decode phases. Although the two phases invoke different code paths in context-dependent modules, the divergence is driven by attention metadata that Dooly's input generation injects per sweep point, exercising both paths at profile time; see Appendix D for details.

6 Duplication-Aware Profiler

The runnable set from §5 defines the operations to profile, but profiling every operation for every configuration is wasteful: many operations are semantically equivalent across models and configurations. The Duplication-Aware Profiler detects these equivalences before profiling.

Signature generation. Dooly formalizes equivalence through *signatures*, where each signature identifies an operation by what is invariant across workloads. A signature comprises three components: (1) the operation name and the values of its MODEL_CONFIG-tainted dimensions and scalars, which are fixed by the model architecture; (2) the set of GPU kernel symbols dispatched during execution, which captures the backend's compile-time kernel selection; and (3) for module-level entries, a digest of the module instance's primitive Python attributes (*e.g.,* `sliding_window`), which captures runtime branching invisible at the kernel level. Workload-dependent dimensions (NUM_TOKS, NUM_REQS) are excluded because they parameterize the sweep, not the operation's identity. Dooly computes the *signature hash* as SHA-256 over a canonical serialization of these three components; the resulting hash serves as the operation's primary key in the latency database. Two operations collapse to the same signature only when all three components match; Dooly prunes the runnable set by skipping any entry whose signature hash is already present in the database from a prior session.

Taint-driven sweep. For each remaining operation, Dooly sweeps the NUM_TOKS- and NUM_REQS-tainted dimensions over user-defined ranges of sequence lengths and batch sizes, keeping MODEL_CONFIG-tainted dimensions fixed. Input tensors are constructed using the taint-driven generation strategy from §5.2, with MIX-tainted dimensions recalculated at each sweep point. The resulting measurements are stored in the latency database, keyed by signature hash and workload configuration, providing the training data from which downstream simulators build per-signature regression models for latency prediction.

Latency database. Profiling results are persisted in a SQLite database keyed by signature hash and workload configuration, making deduplication a primary-key lookup. Communication operations live in a separate sub-schema indexed by hardware topology and tensor-parallel degree, since their latency does not depend on model architecture. See Appendix E for the full schema.

Together, the Tainted Runner, Operation Set Finder, and Duplication-Aware Profiler form *DoolyProf*.

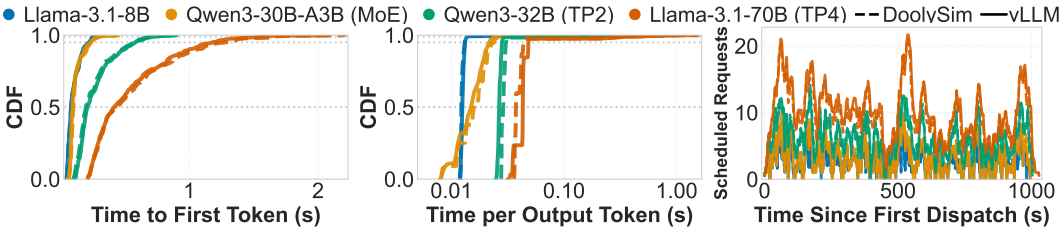
7 Evaluation

We evaluate Dooly on simulation accuracy (§7.1), profiling overhead reduction (§7.2), and taint coverage correctness (§7.3). We span the configuration space introduced in §2.1: model architectures across dense, GQA, MoE, and interleaved sliding-window attention families; three attention backends (FlashInfer, FlashAttention, TritonAttention) and two serving engines (vLLM and SGLang); real-trace (ShareGPT4 [30]) and synthetic prefill-heavy and decode-heavy workloads; and two hardware platforms (NVIDIA A100 and H100).

DoolySim implementation. To evaluate Dooly, we build a simulator called *DoolySim* that predicts per-iteration inference latency from per-signature regression models trained on Dooly's latency database, using the same features as Vidur [3]. The simulator comprises a scheduler that reuses the serving engine's logic to reproduce request batches, and a latency estimator that walks the model's call

graph and queries the per-signature regression models. We implement DoolySim for vLLM; existing simulators [3, 5, 11, 12, 23] can train per-signature regression models from Dooly's standalone latency database as a drop-in profiling backend.

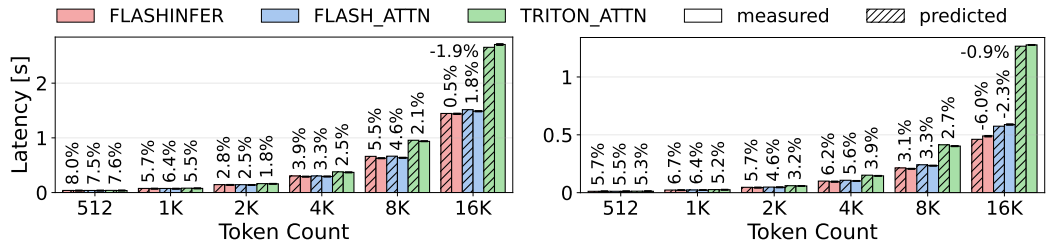
7.1 DoolySim: Simulation Accuracy



(a) TTFT Predicted vs Measured **(b)** TPOT Predicted vs Measured **(c)** Scheduled requests over time

Figure 3: DoolySim accuracy on four model architectures with vLLM and FlashInfer on an A100 GPU, using ShareGPT4 at 0.5 requests per second.

End-to-end performance and scheduling. Dooly predicts vLLM's measured latency with MAPE below 5% for TTFT (Figure 3a) and below 8% for TPOT (Figure 3b) across all reported percentiles. The higher TPOT error reflects the small absolute magnitude of per-token decode latency: sub-millisecond computation and measurement jitter translate into disproportionately large relative error. Figure 3c shows that Dooly faithfully emulates vLLM's batch scheduling decisions, yielding throughput and latency predictions with MAPE below 0.5%. Dooly's operation-level accuracy thus aggregates into reliable whole-system predictions.



(a) A100 **(b)** H100

Figure 4: Per-batch latency predictions for three attention backends on A100 and H100, using Llama-3.1-8B.

Configuration coverage. To validate that DoolySim expresses the configuration space of §2.1, we simulate the prefill-heavy workload on Llama-3.1-8B across three attention backends on both A100 (Figure 4a) and H100 (Figure 4b) GPUs. Dooly predicts TTFT within 0.5–8.0% error across all configurations, with the largest errors at small batch sizes, which are most sensitive to millisecond jitter. This per-operator accuracy lets DoolySim correctly reproduce the inversion points across model and backend selections on both hardware platforms and on the decode-heavy workload. See Appendix F for results on different models, hardware, and workloads, Appendix G for inversion point analysis, and Appendix H for experiments with pipeline parallelism, expert parallelism, and others.

7.2 DoolyProf: Profiling Overhead Analysis

We measure profiling savings across a realistic corpus of 12 models in the 7B–8B parameter range, each profiled with three attention backends (FlashInfer, FlashAttention, TritonAttention), yielding 36 configurations. The corpus spans three attention-architecture families: multi-head attention (Llama-2-7B, DeepSeek-LLM-7B-base), grouped-query attention (Llama-3/3.1-8B, Qwen2.5/3-8B, Mistral-7B-v0.3), and interleaved sliding-window attention (Ministral-8B-Instruct, Command-R7B). All experiments required no modification to DoolyProf or DoolySim to support new backends, models, or hardware, demonstrating out-of-the-box support across the configuration space.

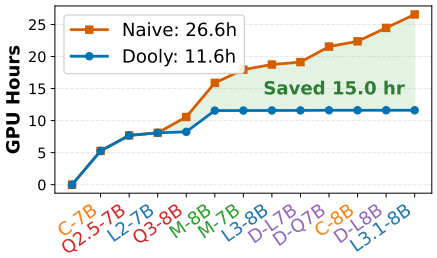


Figure 5: Deduplication in Dooly reduces profiling cost by 56% by sharing latency measurements across configurations. Model labels are defined in Appendix I.

Group	Variant	N	R	Profile (h)	Saved (h)	Red. (%)
Attention	*(aggregate)*	42	27	10.86	14.24	56.7
	32/8/128	24	21	2.83	11.62	80.4
	28/4/128	6	3	2.24	2.24	50.0
	32/32/128	6	3	0.35	0.38	52.2
	window=4K	3	0	2.24	0.00	0.0
	window=32K	3	0	3.20	0.00	0.0
Linear (aten::linear)		180	153	0.54	0.56	50.8
Other operators		390	332	0.18	0.19	50.8
Total		**612**	**512**	**11.58**	**14.98**	**56.4**

Table 2: Per-operation profiling savings. **N**: number of occurrences; **R**: number of reuses; **Profile**: total profiling latency; **Saved**: total hours saved; **Red.**: % reduction.

As shown in Figure 5, a naive per-configuration profiler would profile 612 operator signatures across the 36 configurations, taking 26.6 GPU-hours in total. Dooly identifies 100 unique signatures and deduplicates the rest automatically via call-graph-aware signature matching. Profiling just 4 of the 12 models covers over 98% of the total profiling time, with each of the remaining 8 models adding under 0.2 GPU-hours of new work. This saturation is driven by attention, which dominates per-model profiling cost: Table 2 shows that 95% of the 15 GPU-hours saved come from attention-signature reuse across architecturally similar models, while the remaining 5% is distributed across other operations. Once the first few models exercise the distinct attention configurations in the corpus, later models inherit their attention kernels and add negligible overhead. Appendix J shows that this observation holds for larger models and MoE architectures as well.

Comparison with baselines. We compare Dooly with three profiling strategies. Naive profiling measures every configuration from scratch. HF-Diff (HD) reuses a profile only when the full HuggingFace [17] model configuration files match. HF-Model-Diff (HMD) compares architectural parameters (*e.g.,* head count, layer count, hidden dimension) to identify reusable profiles across models with different configurations. Across the original 36 configurations, Dooly reduces profiling hours by 56.4%, while HD and HMD reduce them by 7.9% and 39.8%, respectively.

Table 3: Profiling cost (GPU-hours) of different methods across the same 12-model corpus. Savings are relative to naive profiling.

Method	Profile (h)	Saved (%)
Naive	26.54	0.0
HF-Diff	24.44	7.9
HF-Model-Diff	15.97	39.8
Dooly	11.58	56.4

HD and HMD fail to capture the models' operation-level structure that Dooly exploits. Consider two models with similar architectures but different attention types: Command-R7B (interleaved, sliding-window attention) and Aya-Expanse-8B (full attention). HMD treats Command-R7B and Aya-Expanse-8B as equivalent because their compared architecture fields match. Reusing the latter's profile for the former causes a 37.3% error in TTFT prediction, while Dooly's operation-level tainting correctly identifies the two models' attention operations as distinct, yielding a 4.1% error (see Appendix K for more details).

7.3 Taint Propagation: Coverage Analysis

To validate that taint propagation correctly classifies each tensor dimension by source, we trace four models spanning diverse architectures: Llama-3.1-8B (dense), Command-R7B (heterogeneous), Qwen3-30B-A3B (MoE), and Llama-3.1-70B (tensor parallelism). We hold the dummy prompt length constant and vary the batch size to inspect all MODEL_CONFIG-, NUM_TOKS-, and NUM_REQS-tagged dimensions in the taint registry, covering vLLM and SGLang, each paired with FlashInfer, FlashAttention, and TritonAttention.

Across all models, MODEL_CONFIG values are constant across batch sizes while NUM_TOKS and NUM_REQS scale exactly with the workload. To stress-test the system, we deliberately use prompt lengths and batch sizes whose values collide with MODEL_CONFIG values (*e.g.,* a batch size of 8 collides with the KV head count dimension); Dooly detects these via conflicting taint assignments and resolves them by retracing with collision-free prompts. Taint classification thus achieves 100% accuracy in distinguishing model-derived from workload-derived dimensions.

Effect of taint propagation rules. The tainted runner relies on two propagation rules, one for each type of tensor operation: (1) dimension-mapping (DM) and (2) dimension-preserving (DP). To measure their effect, we ablate each rule and inspect the resulting trace. As shown in Table 4, removing the dimension-mapping rule leaves 448 untainted operations, while removing the dimension-preserving rule causes a failure due to ambiguity in taint assignment. When the ambiguity failure is suppressed, 241 operations remain untainted, demonstrating that both rules are essential for complete taint coverage. These missing labels affect downstream profiling: an untainted request dimension may remain fixed instead of being swept, while an untainted model dimension is omitted from the signature used for deduplication.

Table 4: Ablation study of Dooly's taint-propagation rules for the two types of operations discussed in §4.2. "Forced" suppresses the ambiguity failure.

Trace variant	Untainted operations
Full Dooly	0
w/o DP	(ambiguity failure) ✘
w/o DP, Forced	241
w/o DM	448

7.4 Operation Set Finder: Effect of Profiling Granularity

Dooly produces a runnable set with mixed granularity: operations that run standalone are profiled at the *operation level*, while context-dependent operations are profiled at the *module level* (§5.3). We compare three module–operation pairs. The first pair is vLLM's `RowParallelLinear` module and the `aten::linear` operation it dispatches through the PyTorch Dispatcher; both execute the same cuBLAS GEMM. The other two pairs are Gemma's RMSNorm and Cohere's LayerNorm modules and their `forward_impl` operations, which `torch.compile` fuses into Triton kernels that bypass the Dispatcher. Across all swept token counts, module- and operation-level latencies differ by less than 1% for the linear pair and less than 0.1% for both normalization pairs, within measurement jitter (Appendix M). Mixed-granularity profiling is therefore accurate for end-to-end simulation; the cost of falling back to a module is a lost deduplication opportunity, not a loss of accuracy.

8 Applications of Dooly

Beyond the profile-based simulation use case, Dooly's configuration-agnostic profiling can be applied to other scenarios where operation-level profiling is needed across multiple configurations. llm-d [28, 24], a Kubernetes-native distributed inference framework, deploys model servers as pods and routes each request to the pod with the lowest predicted latency, a task that grows harder as the pool comes to span different models, GPUs, and serving configurations. Dooly complements such *latency-prediction-based schedulers*: because its operation signatures are shared across configurations, measurements profiled once can be reused across pods, giving accurate TTFT and TPOT predictions for each one without profiling every pod separately. Model engineers can also use Dooly to *benchmark new configurations* that share operations with previously profiled ones, speeding up model development by evaluating them without profiling from scratch.

9 Limitations

Operations that bypass PyTorch's dispatcher (*e.g.,* ones implemented in Triton [38]) can lack kernel-level taints; their enclosing modules, however, receive tainted inputs directly from the trace source, so the call graph remains fully covered. Dooly's Tainted Runner currently traces text prompts, but taint propagation is modality-agnostic and extends to vision or audio by tainting their input sources; adapting it requires only identifying the new modality's request entry points. Finally, Dooly needs updates as serving engines evolve, but only source tainting is engine-specific (*e.g.,* vLLM v0.25.1 needed a 2-line fix), which is easier to maintain than a full engine-specific profiler.

10 Conclusion

We presented Dooly, a configuration-agnostic, redundancy-aware profiling framework for LLM inference simulators. Dooly exploits the fact that operations share common model-specific parameters across configurations: taint propagation labels each tensor dimension at its source to enable cross-configuration deduplication, while hierarchical context resolution reuses the serving engine's logic to isolate stateful operations without manual instrumentation. Dooly lets practitioners evaluate serving strategies across a wide range of configurations at reduced profiling cost. **Broader impacts:** We foresee no significant ethical or societal impact from this work.

Acknowledgments and Disclosure of Funding

We thank the anonymous reviewers for their constructive feedback, which helped improve this paper. This work was supported by the U.S. National Science Foundation (NSF) under Award 2403026.

References

[1] Nvidia cuda profiling tools interface, 2025. URL https://developer.nvidia.com/cupti.

[2] Pytorch profiler documentation, Jul 2026. URL https://docs.pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[3] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for llm inference, 2024. URL https://arxiv.org/abs/2405.05465.

[4] 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, 2024. URL https://arxiv.org/abs/2403.02310.

[5] Amey Agrawal, Mayank Yadav, Sukrit Kumar, Anirudha Agrawal, Garv Ghai, Souradeep Bera, Elton Pinto, Sirish Gambhira, Mohammad Adain, Kasra Sohrab, Chus Antonanzas, and Alexey Tumanov. Revati: Transparent gpu-free time-warp emulation for llm serving, 2026. URL https://arxiv.org/abs/2601.00397.

[6] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023. URL https://arxiv.org/abs/2305.13245.

[7] AMD. vllm v1 performance optimization, 2026. URL https://rocm.docs.amd.com/en/latest/how-to/rocm-for-ai/inference-optimization/vllm-optimization.html.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594299. URL https://doi.org/10.1145/2594291.2594299.

[9] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020. URL https://arxiv.org/abs/2004.05150.

[10] Siyuan Chen, Zhipeng Jia, Samira Khan, Arvind Krishnamurthy, and Phillip B. Gibbons. Slos-serve: Optimized serving of multi-slo llms, 2025. URL https://arxiv.org/abs/2504.08784.

[11] 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)*, page 15–29. IEEE, September 2024. doi: 10.1109/iiswc63097.2024.00012. URL http://dx.doi.org/10.1109/IISWC63097.2024.00012.

[12] Jaehong Cho, Hyunmin Choi, and Jongse Park. Llmservingsim2.0: A unified simulator for heterogeneous hardware and serving techniques in llm infrastructure. *IEEE Computer Architecture Letters*, 24(2):361–364, July 2025. ISSN 2473-2575. doi: 10.1109/lca.2025.3628325. URL http://dx.doi.org/10.1109/LCA.2025.3628325.

[13] Cohere. Coherelabs/c4ai-command-r7b-12-2024. https://huggingface.co/CohereLabs/c4ai-command-r7b-12-2024/.

[14] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

[15] duanzhaol. Adapting vidur to vllm and profiling cpu overhead. https://github.com/microsoft/vidur/issues/51, 2025.

[16] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32 (2), June 2014. ISSN 0734-2071. doi: 10.1145/2619091. URL https://doi.org/10.1145/2619091.

[17] Hugging Face. https://huggingface.co/.

[18] Yicheng Feng, Xin Tan, Kin Hang Sew, Yimin Jiang, Yibo Zhu, and Hong Xu. Frontier: Simulating the next generation of llm inference systems, 2025. URL https://arxiv.org/abs/2508.03148.

[19] fwyc0573. How to get the profile csv of vllm instead of tensorrt-llm? https://github.com/casys-kaist/LLMServingSim/issues/10, 2025.

[20] Google. Perfetto trace viewer. URL https://perfetto.dev.

[21] hariag. Questions about simulation fidelity under vllm version differences, profiling utilization, and throughput estimation. https://github.com/microsoft/apex_plus/issues/8, 2025.

[22] 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.

[23] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. Accelwattch: A power modeling framework for modern gpus. MICRO '21, page 738–753, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480063. URL https://doi.org/10.1145/3466752.3480063.

[24] Pravein Govindan Kannan, Praveen Jayachandran, Jaikrishnan Hari, Varun Raste, Prasad Mukhedkar, Vinod Pathangay, Jayanth Babu Reddy, and Abhisyant Anasapurapu. Heterogeneous inference serving across three GPU vendors with llm-d. https://llm-d.ai/blog/heterogeneous-inference-3-vendor-sovereign-cluster, 2026.

[25] 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 ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[26] Yi-Chien Lin, Woosuk Kwon, Ronald Pineda, and Fanny Nina Paravecino. Apex: An extensible and dynamism-aware simulator for automated parallel execution in llm serving, 2025. URL https://arxiv.org/abs/2411.17651.

[27] Meta. meta-llama/llama-3.1-8b. https://huggingface.co/meta-llama/Llama-3.1-8B.

[28] Kaushik Mitra, Benjamin Braun, Abdullah Gharaibeh, and Clayton Coleman. Predicted latency-based scheduling for LLMs. https://llm-d.ai/blog/predicted-latency-based-scheduling-for-llms, 2026.

[29] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, 2005.

[30] OpenChat. openchat_sharegpt4_dataset. https://huggingface.co/datasets/openchat/openchat_sharegpt4_dataset.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[32] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2024. URL https://arxiv.org/abs/2311.18677.

[33] PyTorch. GitHub - pytorch/kineto: A CPU+GPU Profiling library that provides access to timeline traces and hardware performance counters. — github.com. https://github.com/pytorch/kineto.

[34] Burkhard Ringlein, Jan van Lunteren, Radu Stoica, and Thomas Parnell. The anatomy of a triton attention kernel, 2025. URL https://arxiv.org/abs/2511.11581.

[35] rxz 0420. Running problems with replica scheduler orca/sarathi. https://github.com/microsoft/vidur/issues/64, 2025.

[36] samarth1612. Adding new model. https://github.com/microsoft/vidur/issues/32, 2024.

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

[38] Philippe Tillet. Trtion. https://triton-lang.org/main/index.html, 2020.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

[40] vLLM. Optimization and tuning, 2026. URL https://docs.vllm.ai/en/stable/configuration/optimization/#attention-backend-selection.

[41] An Yang, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoyan Huang, Jiandong Jiang, Jianhong Tu, Jianwei Zhang, Jingren Zhou, Junyang Lin, Kai Dang, Kexin Yang, Le Yu, Mei Li, Minmin Sun, Qin Zhu, Rui Men, Tao He, Weijia Xu, Wenbiao Yin, Wenyuan Yu, Xiafei Qiu, Xingzhang Ren, Xinlong Yang, Yong Li, Zhiying Xu, and Zipeng Zhang. Qwen2.5-1m technical report, 2025. URL https://arxiv.org/abs/2501.15383.

[42] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving, 2025. URL https://arxiv.org/abs/2501.01005.

[43] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

[44] 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, 2024. URL https://arxiv.org/abs/2401.09670.

A Feature Comparison with Existing LLM Simulators

	Dooly	RT [5]	VD [3]	FT [18]	LS[11]	AP [26]	LS2 [12]
Profiler							
Automatic Model Discovery	✓	✗	✗	✗	✗	✗	✗
Operator Set Extraction	✓	✗	✗	✗	✗	✗	✗
Automatic Profiling	✓	✗	✗	✗	✗	✗	✗
Simulation							
Continuous Batching	✓	✓	✓	✓	✓	✓	✓
Chunked Prefill	✓	✓	✓	✓	✓	✓	✓
Prefix Caching	✓	✓	✗	✗	✗	✗	✓
Heterogeneous Models	✓	✗	✗	✗	✗	✗	✗
Mixture of Experts	✓	✓	✗	✓	✗	✓	✓

Table 5: Out-of-the-box capability comparison across profile-based LLM simulators. Dooly (highlighted) automates model architecture discovery and profiling, while existing simulators require manual configuration to support new models or configurations (RT: Revati, VD: Vidur, FT: Frontier, LS: LLMServingSim, AP: Apex, LS2: LLMServingSim 2.0).

DoolyProf automatically discovers the model architecture and extracts the operation set from a single trace, eliminating manual configuration when profiling new models or hardware. By reusing the serving engine's scheduling logic, DoolySim supports continuous batching, chunked prefill, and prefix caching out of the box. Existing simulators, in contrast, require extensive manual configuration to profile new models or support new scheduling features, posing a significant barrier to adoption and maintenance as the ecosystem evolves.

B Resolving Taint Ambiguity

Taint ambiguity arises when multiple dimensions share the same concrete value but carry different taints; such collisions are rare in practice. MODEL_CONFIG values come from the model architecture and are typically powers of two (*e.g.,* 32, 8192); collisions *within* MODEL_CONFIG are benign because the colliding dimensions share the same taint. NUM_TOKS and NUM_REQS come from the dummy prompt, which Dooly controls and sets to values unlikely to match any model-configuration dimension. When a collision does occur, Dooly detects it via conflicting taint assignments and retraces with a different prompt to resolve the ambiguity.

C Trace Annotation Hooks

Dooly augments the trace with taint information at three levels of the PyTorch stack: `Module.__call__` records the enclosing module name (*e.g.,* `layers.0.self_attn`), establishing the module hierarchy context for each operation; `__torch_function__` captures high-level operations with their original shape arguments before operator decomposition; and `__torch_dispatch__` intercepts the resulting low-level ATen operators. Some operations, such as custom attention kernels, may execute outside the `__torch_dispatch__` path and thus lack taint context at the kernel level; Dooly preserves their semantic meaning at higher levels in the call graph, ensuring comprehensive labeling for downstream profiling.

D Bottom-Up Resolution and Single-Trace Sufficiency

After extracting the operation set with the Tainted Runner, DoolyProf iterates through the call graph bottom-up, from leaf kernels to root modules (Figure 6). At each level, DoolyProf performs a dummy run with the tainted parameters to determine whether the operation can run in isolation or is context-dependent and requires module-level resolution. When a parent becomes an anchor because its children are context-dependent, DoolyProf absorbs the children to avoid double-counting. The dummy runs reuse the serving engine's initialization and forward contexts to set up the necessary metadata.

Sufficiency of a single trace. A natural concern is whether a single dummy-prompt trace can cover both prefill and decode call paths. Phase-dependent branching is driven entirely by token-count fields in the attention metadata that the serving engine passes through the forward context, and only context-dependent modules consume this metadata; all other operations in the runnable set are phase-invariant by construction. Dooly prepares context-dependent operations in their module form (§5.2), capturing both branches regardless of which phase the trace exercised, and its input generation mechanism injects the appropriate metadata for each sweep point at profile time. A single trace therefore suffices for full operation coverage.

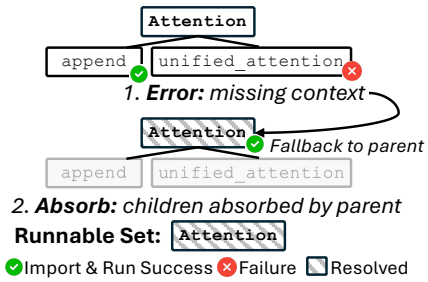


Figure 6: Bottom-up resolution process.

E Latency Database Schema

Dooly persists profiling results in a SQLite database organized around three orthogonal axes: profiled configurations (hardware × model × backend × tensor-parallel degree), unique operation signatures, and workload-dependent measurements. The `model_operations` join table maps each profiled configuration to the signatures it invokes, making deduplication a primary-key lookup against the `signatures` table. Communication operations reside in a separate sub-schema because their latency depends on hardware topology and tensor-parallel degree rather than model architecture; they can therefore be reused across all configurations sharing the same `(topology, tp_degree)`. Figure 7 shows the full schema.



Figure 7: Latency database schema.

F DoolySim Simulation Accuracy

F.1 End-to-End Setup and H100 Results

System configuration	
Hardware	4× NVIDIA A100 GPU
Serving engine	vLLM v0.17.1 (asynchronous scheduling)
Attention backend	FlashInfer
Prefill chunk size	8,192 tokens
Workload trace (>8B parameters)	
Request rate	0.5 req/s
Prompt length	median 950 / mean 1,232 tokens
Output length	median 388 / mean 397 tokens
Models evaluated	
Llama-3.1-8B	Dense, 0.5 rps trace
Qwen3-32B (TP=2)	Dense, 0.5 rps trace
Llama-3.1-70B (TP=4)	Dense, 0.5 rps trace
Qwen3-30B-A3B	MoE, 0.5 rps trace

Table 6: Experiment setup for evaluating the accuracy of DoolySim on an A100 GPU.

System configuration	
Hardware	1× NVIDIA H100 GPU
Serving engine	vLLM v0.17.1 (asynchronous scheduling)
Attention backend	FlashInfer
Prefill chunk size	8,192 tokens
Workload trace (≤14B parameters)	
Request rate	1.0 req/s
Prompt length	median 1,717 / mean 1,911 tokens
Output length	median 385 / mean 410 tokens
Workload trace (>14B parameters)	
Request rate	0.5 req/s
Prompt length	median 950 / mean 1,232 tokens
Output length	median 388 / mean 397 tokens
Models evaluated	
Llama-3.1-8B	Dense, 1.0 rps trace
Ministral-8B	Heterogeneous, 1.0 rps trace
Qwen3-14B	Dense, 1.0 rps trace
Mixtral-8×7B	MoE, 1.0 rps trace, FP8 Quantized
Qwen3-32B (TP=1)	Dense, 0.5 rps trace

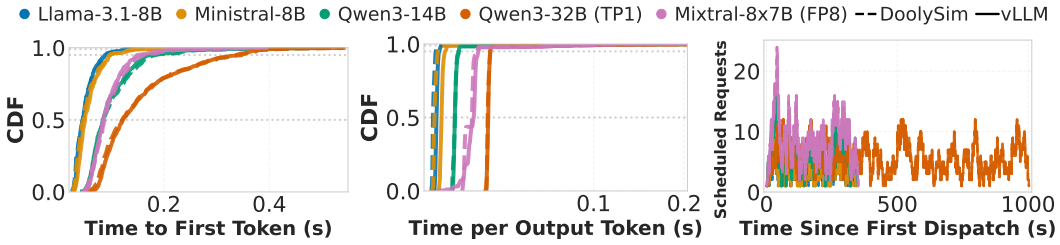
Table 7: Experiment setup for evaluating the accuracy of DoolySim on an H100 GPU.

Experiment setup. Table 6 and Table 7 summarize the experimental setups for the A100 and H100 evaluations, which share the same serving engine and scheduler configuration. We drive the simulation with traces from the ShareGPT4 dataset. Following Vidur [3] and Revati [5], we set the workload intensity just below each platform's maximum serving capacity: too low a rate under-utilizes the system and yields negligible scheduling delay, making latency predictions less meaningful; too high a rate overloads the system and rapidly inflates scheduling delay.

For latency estimation, DoolySim trains one regression model per signature in the latency database. The input features follow Vidur [3] and Revati [5]: token count for non-Attention operations, and prefill token count, batch size, prefill chunk size, and pre-computed KV-cache size for Attention operations. The pipeline is modular, so users can add new features for both profiling and estimation.

Results on an H100 GPU. We repeat the evaluation of §7.1 on an H100 GPU (Figure 8). The MAPEs of TTFT and TPOT predictions are 2% and 5%, respectively. The simulated schedule closely matches

the measured schedule, demonstrating DoolySim's high fidelity in modeling latency and scheduling behavior on H100 GPUs as well.

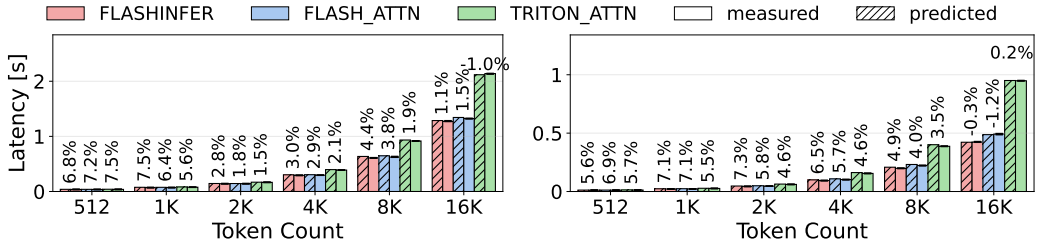


(a) TTFT Predicted vs Measured **(b)** TPOT Predicted vs Measured **(c)** Scheduled requests over time

Figure 8: DoolySim accuracy on five model architectures with vLLM and FlashInfer on an H100 GPU, using ShareGPT4 at 1.0 requests per second.

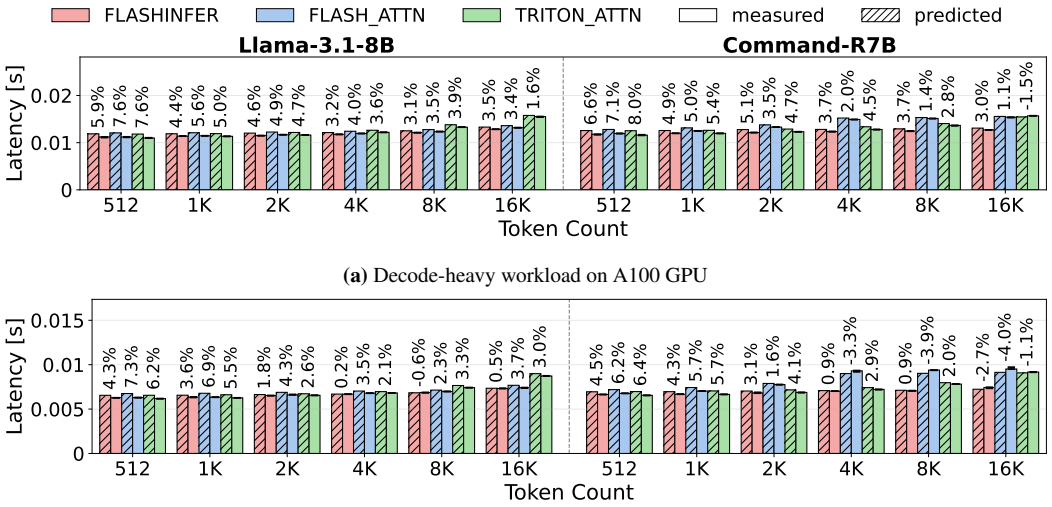
F.2 Per-Batch Accuracy Across Models, Hardware, and Workloads

Experiment setup. Instead of streaming requests, we run a single batch with a fixed size and measure its latency. This isolates per-batch latency prediction accuracy from scheduling effects; per-batch accuracy is critical for accurately modeling the serving engine's scheduling behavior. To prevent vLLM's scheduler from splitting the batch, we set the prefill chunk size larger than the batch's total token count.



(a) A100 **(b)** H100

Figure 9: DoolySim accurately predicts the per-batch latency on Command-R7B with three attention backends on both A100 and H100 GPUs for a prefill-heavy workload.



(a) Decode-heavy workload on A100 GPU

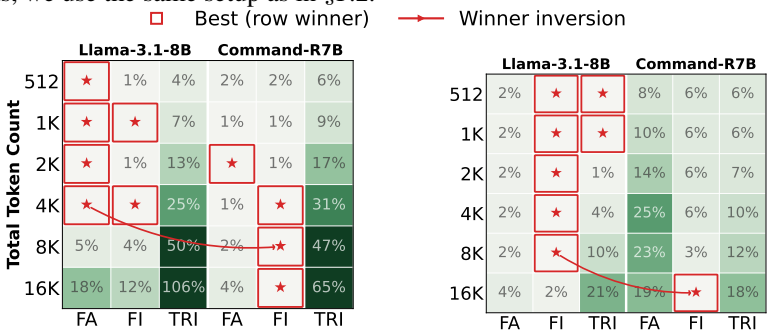
(b) Decode-heavy workload on H100 GPU

Figure 10: DoolySim accurately predicts the per-batch latency across different model and attention backend selections on A100 and H100 GPUs for a decode-heavy workload. See Figure 4 for the prefill-heavy workload's per-batch accuracy.

Figure 9 shows the per-batch accuracy on Command-R7B across three attention backends and two GPUs. Mirroring the Llama-3.1-8B results in Figure 4, DoolySim predicts per-batch latency within 4.5% MAPE across all configurations, with accuracy improving at higher token counts as latency stabilizes. Figure 10 shows the corresponding decode-heavy results on Llama-3.1-8B and Command-R7B; despite higher jitter from smaller token counts, MAPE stays below 3.5% across all configurations.

G Inversion Point Validation

Experiment setup. To validate DoolySim's ability to predict the inversion point across different configurations, we use the same setup as in §F.2.



(a) Predicted Latency for Prefill-Heavy Workload

(b) Predicted Latency for Decode-Heavy Workload

Figure 11: DoolySim correctly predicts the inversion points across different model and attention backend selections on an A100 GPU. See Figure 1 for the measured inversion points.



(a) Measured Latency for Prefill-Heavy Workload

(b) Predicted Latency for Prefill-Heavy Workload

(c) Measured Latency for Decode-Heavy Workload

(d) Predicted Latency for Decode-Heavy Workload

Figure 12: DoolySim correctly predicts the inversion points across different model and attention backend selections on an H100 GPU.

DoolySim's high per-batch accuracy enables it to correctly predict the inversion points across model and attention backend selections on both A100 (Figure 11) and H100 (Figure 12) GPUs.

H Simulation Accuracy Under Pipeline, Expert, and Data Parallelism

Every parallelism strategy composes computation and communication latencies, both of which Dooly already profiles independently; only their composition differs across strategies, which the simulator handles with per-strategy logic. Supporting a new strategy therefore reduces to simulation engineering rather than profiling changes.

Pipeline parallelism (PP). PP decomposes into per-shard compute and point-to-point send/receive between N stages: $t_{\text{total}} = t_{\text{compute}} + (N-1)\, t_{\text{p2p}}$. We extend DoolySim accordingly and evaluate Llama-3.1-8B with PP=2 on two A100 GPUs (Table 8). TPOT error exceeds TTFT error because decode is more susceptible to timing jitter (§7.1).

	TTFT			TPOT		
Context	**Pred. (ms)**	**Meas. (ms)**	**Err. (%)**	**Pred. (ms)**	**Meas. (ms)**	**Err. (%)**
2048	145.76	140.72	3.7	11.71	12.14	3.5
4096	299.85	295.10	1.6	11.81	12.42	4.9
8192	645.86	639.41	1.0	12.19	12.98	6.1
16384	1475.67	1464.12	0.8	13.00	13.94	6.7

Table 8: DoolySim prediction accuracy for Llama-3.1-8B with pipeline parallelism (PP=2) on A100 GPUs.

Expert parallelism (EP). With the per-block (tokens, activated experts) latency tables of §J.2, determining which experts activate, and thus the per-rank load under EP, is the simulator's responsibility. We extend DoolySim to compose the profiled latencies under a balanced routing algorithm. For Qwen1.5-MoE-A2.7B at EP=2, TP=2 on NVLink-connected A100 GPUs, the composition matches measured vLLM TTFT within 7.4% and TPOT within 6.2% (Table 9).

	TTFT			TPOT		
Context	**Pred. (ms)**	**Meas. (ms)**	**Err. (%)**	**Pred. (ms)**	**Meas. (ms)**	**Err. (%)**
2048	44.8	48.3	7.4	5.76	5.42	6.2
4096	77.0	82.0	6.2	6.96	6.69	4.1
8000	146.8	157.9	7.0	9.18	8.93	2.8

Table 9: DoolySim prediction accuracy for Qwen1.5-MoE-A2.7B with expert parallelism (EP=2, TP=2) on A100 GPUs.

Data parallelism (DP). Model replicas run on different GPUs, each serving separate requests. Since each replica behaves identically to the single-GPU (or TP shard) case, the per-configuration accuracy results in §7.1 already validate per-replica latency; simulating DP requires only spawning N independent simulation instances, with no changes to the profiler or simulator logic.

Context parallelism. Sequence parallelism is not a standard inference axis; instead, vLLM exposes Decode Context Parallelism (DCP) and an experimental prefill counterpart, which shard the KV cache and attention along the context dimension and reconcile outputs with an all-gather and a reduce-scatter. Dooly extends to these in principle by profiling attention at its context-sharded shape (as for TP) and the two collectives per topology (as for TP, PP, and EP); we leave an end-to-end evaluation to future work.

I Profiling Overhead: Setup and SGLang Results

Experiment setup. To verify the profiling overhead reduction from deduplication, we profile our 12-model corpus across three attention backends and measure the total profiling time with and without deduplication. For Figure 5, we use a 32K prefill chunk size (commonly used in production) and a maximum batch size of 256 (vLLM's default) [7, 40, 41, 25]. For the SGLang [43] experiment in Figure 13, we keep the same prefill chunk size but use a smaller maximum batch size of 64.

The labels used in Figure 5 and Figure 13 are defined as follows:

- **L3-8B**: `meta-llama/Meta-Llama-3-8B`
- **L3.1-8B**: `meta-llama/Llama-3.1-8B`

- **L2-7B**: `meta-llama/Llama-2-7b-hf`
- **Q2.5-7B**: `Qwen/Qwen2.5-7B`
- **Q3-8B**: `Qwen/Qwen3-8B`
- **C-8B**: `CohereLabs/aya-expanse-8b`
- **C-7B**: `CohereLabs/c4ai-command-r7b-12-2024`
- **M-7B**: `mistralai/Mistral-7B-v0.3`
- **M-8B**: `mistralai/Ministral-8B-Instruct-2410`
- **D-L7B**: `deepseek-ai/deepseek-llm-7b-base`
- **D-L8B**: `deepseek-ai/DeepSeek-R1-Distill-Llama-8B`
- **D-Q7B**: `deepseek-ai/DeepSeek-R1-Distill-Qwen-7B`

Deduplication savings for SGLang. SGLang shows a 65.6% reduction in profiling time through deduplication, with Attention again dominating the total cost (Table 10), consistent with the vLLM results in Table 2. Unlike vLLM, SGLang does not distinguish the interleaved attention kernels, so the Ministral model's profiling cost cannot be amortized.

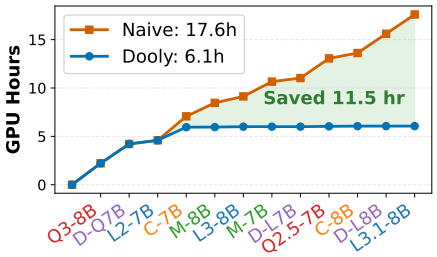


Figure 13: SGLang counterpart to Figure 5: deduplication reduces profiling cost by 65.6% by sharing latency measurements across configurations.

Group	Variant	N	R	Profile (h)	Saved (h)	Red. (%)
Attention	*(aggregate)*	37	26	5.28	10.35	66.2
	GQA 32/8/128	23	20	2.01	8.25	80.4
	GQA 28/4/128	6	3	1.76	1.76	50.0
	MHA 32/32/128	6	3	0.34	0.34	50.0
	SWA window=4K	2	0	1.17	0.00	0.0
Linear (aten::linear)		140	121	0.59	1.01	63.0
Other operators		213	175	0.19	0.18	48.4
Total		**390**	**322**	**6.06**	**11.54**	**65.6**

Table 10: Per-operation profiling savings for SGLang. **N**: number of occurrences; **R**: number of reuses; **Profile**: total profiling latency; **Saved**: total hours saved; **Red.**: % reduction.

J Profiling Savings for Larger and MoE Models

J.1 Larger Models (13B–72B)

To test whether deduplication savings hold beyond the 7B–8B corpus, we extend the analysis of Table 2 to nine models from 13B to 72B parameters, with no changes to Dooly. As shown in Table 11, deduplication saves 8.21 GPU-hours, a 62.4% reduction. Attention again dominates, accounting for 91% of the savings: four models across two Qwen generations and a DeepSeek release (14B–32B) collapse to one attention signature, and the three largest models (TP=4) collapse to another. This works because Dooly profiles tensor-parallel models at per-rank geometry by emulating a single shard (§4). At TP=4, the 70B–72B models' 64/8/128 attention becomes 16/2/128 per rank, so compute operations deduplicate against any model sharing that per-rank geometry. Communication introduced by tensor parallelism is profiled separately and reused across all configurations sharing the same topology and TP degree (Appendix E).

Group	Variant	N	R	Profile (h)	Saved (h)	Red. (%)
Attention	*(aggregate)*	30	15	3.82	7.47	66.2
	DeepSeek-Qwen-32B, Qwen2.5-14B/32B, Qwen3-14B	12	9	0.99	5.54	84.8
	Llama-2-70B, Llama-3.3-70B, Qwen2.5-72B	9	6	1.61	1.93	54.5
	Llama-2-13B	3	0	0.26	0.00	0.0
	Gemma-2-27B	6	0	0.96	0.00	0.0
Linear (aten::linear)		135	107	0.98	0.63	39.1
Other operators		252	207	0.14	0.11	43.7
Total		**417**	**329**	**4.94**	**8.21**	**62.4**

Table 11: Per-operation profiling savings for nine models from 13B to 72B parameters. Columns follow Table 2.

J.2 Mixture-of-Experts Models

Dooly supports MoE layers by adding the number of activated experts as a sweep dimension. A hook on vLLM's `FusedMoE` module, the common entry point for every MoE architecture in vLLM, forces exactly the target number of distinct experts to fire, producing a (tokens, activated experts) latency table per MoE block that is independent of the model's learned routing.

Routing independence. Routing affects MoE latency only through the token count and the number of activated experts: since experts share identical shapes, latency is independent of which experts fire or of token values. Gate-router latency varies by at most 1.4% across random inputs at a fixed shape. Consequently, `FusedMoE` signatures generalize across architectures: Table 12 shows that matching signatures yield nearly identical latency for DeepSeek-V2-Lite (DVL) and DeepSeek-MoE-16B (DM), despite different model weights.

# Tokens	# Activated Experts	DVL (ms)	DM (ms)	Diff. (%)
512	16	0.57	0.57	0.5
512	64	0.95	0.94	0.4
1024	16	1.01	1.00	0.2
2048	64	2.38	2.39	0.2

Table 12: Profiled `FusedMoE` latency for DeepSeek-V2-Lite (DVL) and DeepSeek-MoE-16B (DM).

Deduplication savings. Table 13 extends the deduplication analysis of Table 2 to a corpus of dense and MoE models. Dooly saves 61.4% of GPU-hours overall and 72.8% for MoE operations, with no model-specific implementation.

Group	Variant	N	R	Profile (h)	Saved (h)	Red. (%)
Attention	*(aggregate)*	16	2.3	1.59	2.47	60.8
	Llama-3.1-8B, Mistral-7B, Mixtral-8×7B	9	3.0	0.93	1.99	68.2
	Qwen1.5-MoE, DM	6	2.0	0.27	0.48	63.5
	DVL (MLA)	1	1.0	0.39	0.00	0.0
FusedMoE	*(aggregate)*	10	3.3	0.17	0.47	72.8
	Mixtral	3	3.0	0.14	0.28	66.7
	Qwen	3	3.0	0.02	0.11	85.7
	DVL, DM	4	4.0	0.02	0.08	81.6
Linear (aten::linear)		72	4.5	0.10	0.09	47.3
Other operators		138	4.5	0.05	0.01	21.0
Total		**236**	**4.1**	**1.91**	**3.03**	**61.4**

Table 13: Per-operation profiling savings across dense and MoE models. DVL: DeepSeek-V2-Lite; DM: DeepSeek-MoE-16B. Columns follow Table 2.

K Accuracy Cost of Architecture-Level Deduplication

Context	Measured (ms)	Dooly Pred. (ms)	HMD Pred. (ms)	Dooly Err. (%)	HMD Err. (%)
2K	153.24	159.5	159.4	4.1	4.0
4K	308.36	323.0	323.2	4.8	4.8
8K	628.62	666.1	688.8	6.0	9.6
16K	1310.73	1388.0	1556.8	5.9	18.8
32K	2788.10	2921.9	3827.5	4.8	37.3

Table 14: TTFT prediction for Command-R7B under Dooly and HF-Model-Diff (HMD) profiles.

HF-Model-Diff (HMD) deduplicates profiles using only the model architecture parameters in the HuggingFace configuration file. Command-R7B and Aya-Expanse-8B agree on all of these fields, so HMD reuses Aya-Expanse-8B's profile for Command-R7B. However, Command-R7B interleaves sliding-window attention, whereas Aya-Expanse-8B uses full attention. The two attention types call

different kernels, as determined by the module-instance state. This state, however, is invisible from the configuration parameters alone, so Command-R7B's sliding-window attention is never profiled under HMD. Table 14 shows that HMD's TTFT error grows with context length, from 4.0% at 2K to 37.3% at 32K, while Dooly's error stays between 4.1% and 6.0%.

L Ablation: Module-Instance State in Signatures

A naive alternative to Dooly's signatures hashes operations by module name, treating every instance of a class as identical. This breaks for models with heterogeneous layers. Command-R7B interleaves sliding-window attention (24 layers) and full attention (8 layers), yet all 32 layers are instances of the same attention class with identical tensor shapes and configuration dimensions, so a name-based key merges them. Adding the dispatched kernel symbol to the key does not help: on the FlashAttention and Triton backends, both attention types dispatch the same kernel, differing only in a runtime argument (the window size). Dooly resolves this by folding the module instance's runtime state (here, the sliding-window setting) into the signature.

Name-based hashing introduces no error for Llama-3.1-8B, whose 32 attention layers are homogeneous, but overestimates Command-R7B's attention latency by up to 3.66× at its 132K-token maximum context: the 24 sliding-window layers are never profiled independently and each inherits the higher latency of the full-attention layers it is merged with.

M Profiling Granularity: Accuracy vs. Deduplication

Measurement procedure. Given a callable (an operation or a module), Dooly profiles it with CUDA Graphs: after GPU warm-up, a graph captures 10 runs of the callable, repeated for 20 captures. Each capture is timed with CUDA events and averaged over its 10 runs, and Dooly reports the median of the 20 averages. Because CUDA Graphs execute kernels back to back regardless of Python-level call structure and CUDA events time only kernel execution, module- and operation-level callables that invoke the same kernels yield the same latency.

Tokens	Module (μs)	Op (μs)	Abs. gap (μs)	Rel. gap (%)	CoV (%)
512	347.49	348.44	−0.96	−0.27	0.90
2048	1118.21	1120.69	−2.48	−0.22	0.49
8192	4309.24	4306.06	3.18	0.07	0.27
16384	8519.24	8528.62	−9.37	−0.11	0.12

Table 15: `RowParallelLinear` (module) vs. `aten::linear` (operation) latency (in: 4096, out: 14336).

Dispatcher-level operation vs. parent module. We compare `aten::linear`, captured via the PyTorch dispatcher, against its parent module `RowParallelLinear` using Llama-3.1-8B's dimensions ($[X, 4096] \times [4096, 14336]$, X swept from 512 to 16384 tokens). Both paths invoke the same cuBLAS kernel; Table 15 shows relative differences under 1% with comparable jitter.

Implementation	Tokens	Module (μs)	Op (μs)	Abs. gap (μs)	Rel. gap (%)	CoV (%)
Gemma RMSNorm	512	5.38	5.38	0.0007	0.013	0.42
	2048	21.79	21.80	−0.01	−0.027	0.33
	8192	83.59	83.62	−0.03	−0.033	0.13
	16384	161.64	161.66	−0.03	−0.017	0.07
Cohere LayerNorm	512	6.36	6.35	0.003	0.06	0.15
	2048	22.56	22.55	0.013	0.06	0.29
	8192	85.05	85.03	0.0157	0.02	0.14
	16384	165.76	165.81	−0.05	−0.03	0.27

Table 16: Triton-fused custom modules vs. their `forward_impl` operations.

Triton-fused custom kernels. We repeat the comparison for `torch.compile`-fused Triton normalization kernels that bypass the dispatcher entirely (Gemma RMSNorm and Cohere LayerNorm), comparing each module against its `forward_impl` operation. Table 16 shows relative gaps under 0.1%.

Cost of coarser granularity. The cost of falling back to module-level profiling is a lost deduplication opportunity, not a loss of accuracy. For instance, Mamba-Codestral-7B-v0.1's `MambaMixer2` module calls two linear operations, an RMS normalization, and the `mamba_mixer2` operation. Because `mamba_mixer2` is context-dependent, the Operation Set Finder profiles the enclosing `MambaMixer2` module and absorbs its sibling operations (§5.2); the linear operations, which could otherwise be shared across configurations, are no longer deduplicated. More generally, a standalone operation is absorbed only when a context-dependent operation forces a fallback to an enclosing module that also contains it. This does not occur for attention or MoE, whose context-dependent operations resolve at a tight submodule separate from the standalone operations.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We explain the problems mentioned in the abstract and introduction in §2 and explain our solutions in §3. Further detail is provided in §4,§5,§6.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss the limitations of our work in §9.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: [N/A]

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We provide detailed information about the experiments including the models used, the public dataset used, and the configurations used for the experiments. We also provide the RPS value used during the streaming experiments. The code will be publicly released upon publication.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We used open datasets for experiments. We will release the source code used for our experiments upon publication.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: We provide details about the experiment set-up in Appendix F and §F.2.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We report the MAE across all percentiles of the CDF in the streaming experiments. We also report the error bars in the per-batch experiments for all figures. Note that the error bars are not visible in some configurations due to the small variance of latency across runs that have been measured after multiple warm up runs.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We report the specific GPU used for all experiments.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The application of this work to LLM inference simulations does not have negative societal impacts or ethical concerns. Our method does not have negative societal impacts, as we use publicly released data and model checkpoints.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: Dooly is a profiling tool for LLM inference systems with no direct path to negative societal applications; we note this in §10.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: [N/A]

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We cite the original paper/website/license used in our experiments.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.

- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: This paper does not release new assets.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: [N/A]

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: [N/A]

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: [N/A]

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.