

MIST: A Co-Design Framework for Heterogeneous, Multi-Stage LLM Inference

Abhimanyu Rajeshkumar Bambhaniya^{1,7}, Hanjiang Wu¹, Suvinay Subramanian², Sudarshan Srinivasan³, Souvik Kundu⁴, Amir Yazdanbakhsh⁵, Midhilesh Elavazhagan³, Madhu Kumar³, Minlan Yu⁶, Arijit Raychowdhury^{1,7}, Tushar Krishna^{1,7}

¹Georgia Institute of Technology, ²Google, ³Intel, ⁴Intel Labs, ⁵Google DeepMind, ⁶Harvard University, ⁷Infravana

Corresponding email: abambhaniya3@gatech.edu

Abstract—Modern LLM serving spans multi-stage pipelines, including RAG retrieval and KV cache reuse, with distinct compute, memory, and latency demands. Inference engines expose vast configuration spaces with no systematic navigation methodology, and exhaustive benchmarking can exceed \$40K in cloud costs. Meanwhile, hardware is diversifying across AMD GPUs, TPUs, and custom ASICs, yet cross-vendor prefill-decode (PD) disaggregation lacks a unified software stack for end-to-end evaluation. We present MIST¹, an event-driven simulation framework that models the full serving stack across AI Workload, System & Software, and Hardware layers, with pluggable accelerator models enabling cross-vendor evaluation. MIST achieves end-to-end fidelity within 6% of real deployed systems. In design-space exploration case studies, under stated pricing and hardware assumptions, multi-vendor PD-disaggregated deployments identified by MIST yield up to 49.3% higher tokens/\$ than single-vendor PD disaggregation. MIST also enables KV storage co-design, revealing how memory hierarchy choices shape tail latency. MIST is available at <https://github.com/MIST-Simulator/MIST>.

Index Terms—Large language models, discrete event simulation, computational modeling, performance evaluation, graphics processing units, cloud computing, memory management

I. INTRODUCTION

Large Language Model (LLM) inference has rapidly become a principal workload in modern computing infrastructure. Inference must satisfy strict latency, throughput, and cost constraints under highly variable request patterns, spanning trillion-parameter frontier models for reasoning [27], [58] and agentic workflows [41], [60] to compact models powering chatbots and retrieval-based assistants [43], [48], [53]. As adoption accelerates, the efficiency of LLM serving platforms increasingly determines the usable performance of AI systems at scale, and navigating their design space has become a challenge in its own right.

Today’s LLM inference is mediated by *inference serving engines* such as vLLM [37], SGLang [65], llama.cpp [25], TensorRT-LLM [47], and Nvidia Triton [3], which sit between AI applications and the underlying hardware. As shown in Fig. 1(a), an inference server exposes a large number of configurable *knobs*: AI model selection and routing policy; request scheduling (prefix cache placement); token generation

batching (continuous, chunked, mixed, disaggregated); inter-node communication (KV cache transfer, context retrieval); kernel optimizations (parallelism, flash attention, KV memory organization); and model-level optimizations (sparsity, quantization, KV compression). Unfortunately, there is a lack of a systematic methodology for navigating these knobs today. Practitioners rely on community wisdom and expensive trial-and-error to configure these knobs for deployments. As an example, exhaustively benchmarking Llama-3-70B configurations for a single H100 8-GPU deployment would exceed \$40,000 in cloud cost. **Challenge 1: The configuration space of LLM inference engines is vast and interdependent, with no systematic methodology to identify optimal deployments, making exhaustive search prohibitively expensive.**

Beyond configuration complexity, LLM inference is no longer just prefill and decode: it also includes RAG lookup, KV cache retrieval, and tool calls. This is shown in Fig. 2(a). These additional stages may not always dominate end-to-end runtime, but they introduce non-trivial secondary effects that cannot be ignored. RAG retrieval injects a large external context that must be transferred from the retrieval device to the prefill accelerator, increasing effective prompt length and Time-To-First-Token (TTFT). KV cache retrieval transfers previously stored key-value tensors from tiered storage (DRAM, SSD, or remote memory) to the prefill and decode devices, adding memory bandwidth pressure and transfer latency on top of compute savings. In disaggregated deployments, these cross-device data movements compound with stage-to-stage handoffs, making the network and memory fabric a first-order performance concern. **Challenge 2: Modern AI requests are multi-stage; even when individual stage runtimes are modest, the associated data transfers and their downstream effects on context size and KV state make end-to-end modeling essential.**

This complexity compounds with a rapidly diversifying hardware ecosystem beyond Nvidia GPUs: AMD GPUs offer competitive compute density, Google TPUs high memory bandwidth per dollar, and custom ASICs (e.g., Trainium, Inferentia, Etched) target patterns such as attention-heavy decode or memory-bound prefill. Prefill-decode (PD) disaggregation expands the space further by letting different hardware serve different phases. Even a simple news search

¹Multi-stage AI Inference Simulation Toolkit

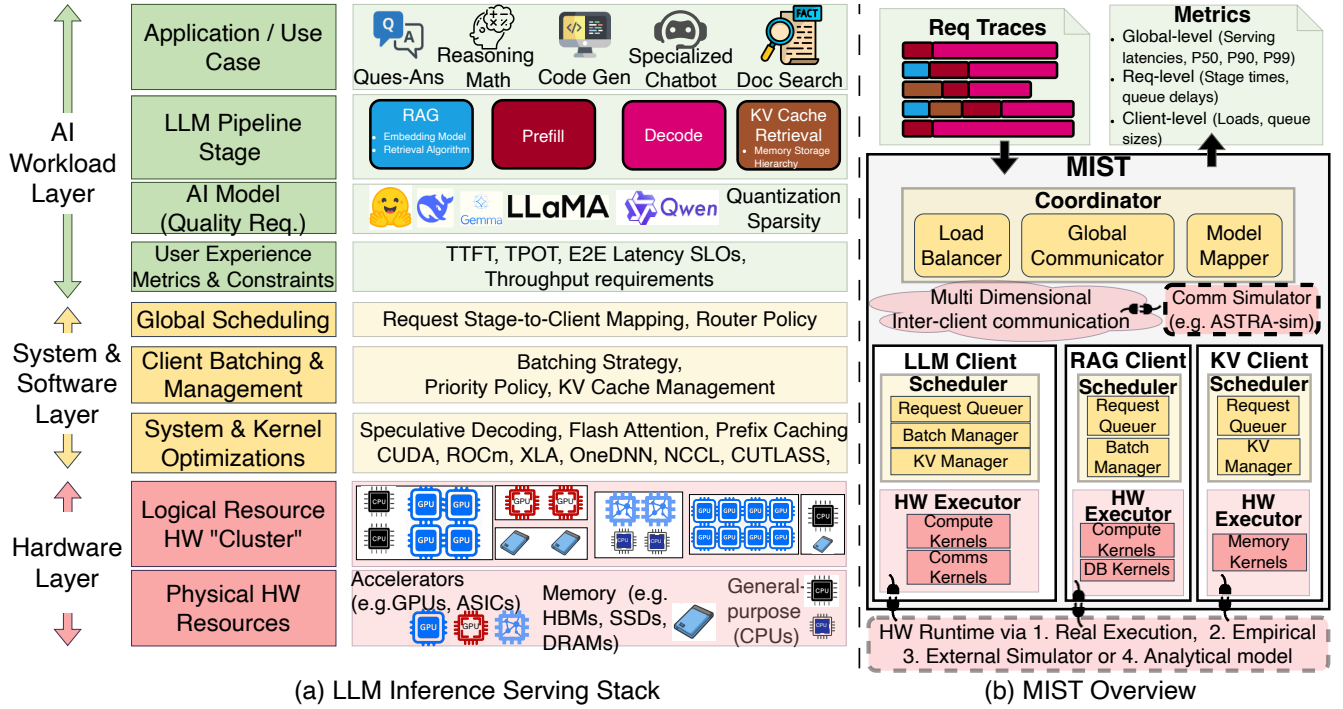


Fig. 1: Understanding the LLM inference serving stack and how MIST models the stack. MIST simulates a collection of clients. Each client consists of a scheduler that issues steps (e.g., tokens for decode, chunks for prefill, Embedding/Retrieval for RAG) to an HW cluster (e.g., Nvidia HGX, AMD MI300X, CPU host with an offloading memory instance). The HW cluster is a collection of multiple HW Nodes (e.g., GPUs, ASICs, memory, and CPUs).

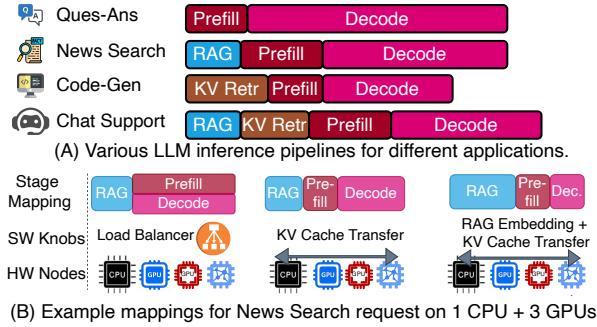


Fig. 2: (a) LLM inference request types: *Question-answering* (Standard); *News search* (RAG pipeline) [24]; *Code generation* (KV cache reuse) [31]; *Chat support* (RAG + KV cache) [22]. (b) Different ways of mapping news search requests across 1 CPU and 3 heterogeneous GPU nodes. For each mapping, different SW knobs, such as the load balancer and global communicator, must be activated.

pipeline (Pre → RAG → Prefill → Decode → Post) admits many mappings: all stages on one node, RAG on a CPU with prefill and decode sharing a GPU, or each stage on a dedicated node (Fig. 2(b)). Industry moves like Nvidia+Groq [12], AWS+Cerebras [57], and Gimlet Labs’ heterogeneous inference cloud [23] signal growing multi-vendor adoption, yet most such combinations lack a mature software stack and cannot be benchmarked end-to-end today, forcing architects into multi-year investment decisions without evaluating cross-vendor pipelines at scale. **Challenge 3: Mapping diverse inference stages to heterogeneous multi-**

vendor hardware and reasoning about interactions across stages and chip boundaries create a non-trivial challenge for efficient LLM inference deployment.

We present MIST, an event-driven, end-to-end simulation framework for heterogeneous, multi-stage AI inference pipelines that addresses all three challenges. MIST models the full serving stack across an **AI Workload Layer** (diverse pipeline stages), a **System & Software Layer** (scheduling, batching, and kernel optimizations), and a **Hardware Layer** (logical abstraction of compute, memory, and network resources), with pluggable accelerator models enabling evaluation of cross-vendor deployments without a unified software stack. MIST provides:

- Diverse inference stages (RAG, KV cache retrieval across DRAM/SSD/remote memory, reasoning, prefill, and decode) with multi-model pipelines and heterogeneous PD-disaggregated configurations.
- Global routing and load-balancing; client-level scheduling; batching strategies; and KV cache management (placement, eviction, reuse) across heterogeneous clients and memory tiers.
- Modular hardware abstractions supporting real-HW plugins, runtime predictors, or cycle-accurate simulators for accelerators not yet in production.

Across validated hardware platforms, MIST achieves end-to-end fidelity within 6% of real-system performance.

Using MIST, we explore two previously inaccessible classes of question:

- **Heterogeneous deployment exploration:** under our stated

pricing and hardware assumptions, multi-vendor PD-disaggregated configurations can yield up to 49.3% more tokens/\$ than homogeneous baselines (Sec. V-A).

- **KV storage architecture co-design:** how memory-hierarchy design and cache granularity interact with end-to-end latency across workloads with varying KV length and reuse patterns (Sec. V-B).

MIST provides the first end-to-end framework spanning heterogeneous hardware, multi-stage pipelines, and KV storage co-design, offering a principled foundation for optimizing current deployments and architecting next-generation AI infrastructure.

II. BACKGROUND ON LLM INFERENCE PIPELINES

Fig. 2 showcases the components within the workflow of modern inference. Diverse application use cases demand different pipelines. For example, a news search involves a RAG lookup followed by prefill and auto-regressive decode, while accurately solving complex mathematical problems typically interleaves generation with reward-model evaluation [9], [40].

A. Stages of LLM Inference

A modern LLM inference request passes through several distinct computational stages, each with different resource demands and latency characteristics.

Prefill and Decode. Prefill performs a single forward pass over the input prompt to generate the first token, making it compute-intensive. Decode then proceeds auto-regressively, generating tokens sequentially. This stage is memory-bound and benefits from batching multiple tokens for higher throughput. Chunked (Aggregated) batching and Disaggregated batching are the most popular ways of batching prefill and decode for concurrent requests to improve the efficiency and throughput of the system.

Chunked Batching improves system throughput by splitting long input sequences into smaller, fixed-size chunks.

Disaggregated Batching decouples prefill and decode stages by assigning them to independently scaled hardware instances, enabling flexible resource allocation for heterogeneous workloads. Splitwise [49] introduced the idea of heterogeneous GPU use, but the software stack is still evolving, and multi-vendor disaggregated LLM inference lacks open-source support. Recent trends in the industry [12], [23], [57] highlight the growing adoption of **multi-vendor hardware** for disaggregated inference.

Retrieval Augmented Generation (RAG). As shown in Fig. 3(a), RAG retrieves relevant documents via embedding models and ANN search (e.g., FAISS [33]) before augmenting the prompt for LLM generation, improving factuality and reducing hallucination. We adopt DPR (Dense Passage Retrieval) [34] for embedding and IVF-PQ [19] as our ANN method for its balance of memory efficiency and recall at billion scale [5].

KV Cache Retrieval. As shown in Fig. 3(b), Prefix Caching [1] reduces TTFT by reusing KV cache entries from

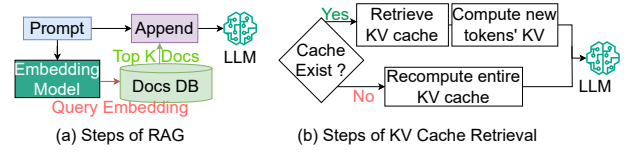


Fig. 3: Steps in (a) RAG & (b) KV Retrieval.

prior queries that share a prefix, bypassing redundant prefill computation [18], [42], [61]. KV caches can also be persisted across sessions to support multi-turn conversations [61].

B. Architecture of an Inference Server

Table I shows the module hierarchy of a modern LLM inference server. The top-level **Inference Server** owns a *Coordinator* which orchestrates the *Load Balancer*, *Global Communicator* (e.g., for inter-client KV cache transfer), and *Model Mapper*. Within each **Inference Client**, a **Scheduler** handles request ordering, batching strategy, and KV cache management, while the **HW Executor** has accelerator-specific optimized kernels for compute (e.g., CUDA/ROCm/XLA) and communication (e.g., NCCL/RCCCL).

TABLE I: Modules of LLM Inference Server.

Module	Description
Inference Server	Receives req & sends response.
Coordinator	Manages coordination between clients.
Load Balancer	Decodes request→client mapping.
Global Communicator	Inter-client comm. (e.g., KV cache ⇔).
Model Mapper	Routes request to appropriate LLM.
Inference Client	Hosts a single LLM replica
Scheduler	Selects requests for each fwd pass.
Request Queuer	Orders requests by packing policy.
Batch Manager	Batches requests by batching policy.
KV Manager	Manages KV Cache across GPU, CPU, and tiered KV caches.
HW Executor	HW-specific model executor
Compute Kernels	Optimized kernels for the target HW.
Comms Kernels	Collective kernels for parallelism.

III. MULTI-STAGE AI INFERENCE SIMULATION TOOLKIT

We introduce MIST, a simulation framework designed to capture the complexity of real-world LLM inference pipelines. To resemble real deployments, MIST simulates the end-to-end execution of state-of-the-art LLM inference pipelines across the three layers of the LLM Inference serving stack shown in Fig. 1(a).

A. Overview

Fig. 1(b) shows an overview of MIST. The **AI Workload Layer** knobs are the input to the MIST framework. Use-case-specific *requests* are injected into the framework at different arrival times. A **Request** (Sec. III-B1) is a pipeline of *stages*, where each stage could have one or more *steps*; for instance, RAG has an embedding step and a document retrieval step (Fig. 3).

The **System and Software Layer** forms the core of MIST. The flow starts with a **Coordinator** (Sec. III-C) that performs global scheduling of the inference pipeline. Specifically, it maps and routes different stages of requests to appropriate *clients* based on routing and load-balancing policies. Each **Client** (Sec. III-D) employs a runtime **Scheduler** (Sec. III-E) for executing the different steps of the mapped stage over its underlying HW *cluster* using its **HW Executor**.

The **HW Layer** is modeled as a **Cluster** (Sec. III-F), a logical collection of the hardware nodes within a physical AI platform². The hardware nodes in the cluster can be any viable combination of GPUs, ASICs, CPUs, and memories, along with their connectivity. For each client's HW executor runtime simulation, users can choose among (i) real execution runtime, (ii) empirically observed runtime values from real HW, (iii) analytical modeling equation, or (iv) external simulators.

Thus, each client incorporating a scheduler and an HW executor is similar to real-world LLM frameworks such as SGLang [65] and vLLM [37].

The final outputs (Sec. III-G) of MIST are various LLM performance metrics at the request level, client level, and coordinator level, such as latency and throughput.

B. Workload Layer: Inputs and Parameters

MIST accepts request traces as input from the AI workload layer. In the system and software layer, the router policy, client configuration (Scheduler config + HW clusters), and global client connections are user-defined hyperparameters.

1) *Request Modeling*: MIST currently supports requests that are any combination of four stages of the LLM pipeline (**RAG**, **Prefill**, **Decode**, and **KV Cache Retrieval**). Users can construct an arbitrary combination of stages, as shown in Fig. 2, to simulate a modern AI use case³. Because each stage maps independently onto a hardware client, disaggregation beyond the prefill/decode split is expressed purely as a stage-to-client mapping: splitting decode yields 3-way disaggregation (Prefill / Decode-Attn / Decode-FFN), and splitting prefill as well yields a 4-way configuration [28]. MIST also models speculative decoding [38] with an EAGLE-style [39] draft model, exposing the speculative token count and acceptance rate as configurable parameters; the draft stage maps to its own client, so dedicated drafter hardware is expressible in the same way. **Model specification**. MIST describes each model by its architectural parameters (number of layers, embedding dimension, number of attention heads, attention type, number of experts, and parallelism strategy) rather than by a fixed catalog of released checkpoints. Hypothetical models can therefore be evaluated by scaling these parameters, which matters at design time, when the target model for a platform shipping in 18–24 months may be an order of magnitude larger than anything available today.

²For example, a physical HGX box with 8×H100 GPUs and 2 CPUs can be mapped as two logical clusters, each with 4×H100 and 1 CPU, where the first cluster serves as a prefill client and the second is a decode client in a disaggregated setup.

³Stages can be extended to model tool calls in agentic workflows.

Algorithm 1: Coordinator Simulation Algorithm

```

1: Initialize: Client interconnect topology
2: Enqueue arrival of all requests (STAGE-PUSH)
3: while request_serviced < request_accepted do
4:   Execute next discrete event in queue
5:   if Event is STAGE-PUSH then
6:     Dispatch stage to client
7:     if Client not allotted then
8:       Clientnext ← Router(Request)
9:     end if
10:    Clientnext.add(Request)
11:    Enqueue Client to activate next step if idle (CLIENT-STEP)
12:  else if Event is CLIENT-STEP then
13:    Process client step and completed requests
14:    Finished_Requests ← Client.next_step()
15:    if Client has requests to process then
16:      Enqueue Client for next step (CLIENT-STEP)
17:    end if
18:    for each request finished current stage do
19:      if request is complete then
20:        Mark request as serviced
21:      else
22:        Clientnext ← Router(Request)
23:        Start client-transfer event
24:        Enqueue request for next stage (STAGE-PUSH)
25:      end if
26:    end for
27:  end if
28: end while

```

When the corresponding hardware is also unavailable, the forward-pass runtimes for such models are supplied by the external analytical backends described in Sec. III-F.

2) *Input Datasets and Workloads*: Inference begins with feeding a trace of requests into the system.

Request size: To model diverse prefill and decode token workloads, we use a combination of real and synthetic traces. *Real traces* from production services, such as the Azure trace [13] (Conv and Code), capture realistic input-output token distributions. *Synthetic traces* are generated based on observed characteristics in common workloads. They are modeled as a normal distribution with a user-configurable mean and variance for input and output tokens.

Request injection is modeled using a range of models including uniform, normal, Poisson, and bursty distributions. This approach better reflects real-world traffic patterns and enables more robust evaluation of system behavior under diverse operational scenarios.

Additionally, each request includes additional parameters depending on the associated stages (e.g., RAG requests require RAG algorithm parameters).

C. System SW Layer: Coordinator

The coordinator manages end-to-end inference execution across clients, ensuring ordered stage scheduling and inter-stage communication. algorithm 1 shows the core simulation loop, which integrates event scheduling (*Client & Request* events), routing, and inter-client communication in a unified discrete-event framework. *Request event*: A new request entering the system, or a client returning a request after

servicing a stage. *Client event*: Scheduling and execution of the steps for requests assigned to a client. E.g., for a prefill/decode client running on an 8xH100 node with chunked batching, the client event schedules a single chunk batch with assigned prefill and decode batches, and simulates the HW runtime of the scheduled batch. *Client Transfer event*: Communication from one client to another based on how the stages are assigned to different clients. E.g., for disaggregated batching, this would be KV cache transfer from prefill to decode client.

1) *Load Balancer*: To determine the next client for a given request stage, the coordinator uses a routing module. When multiple clients can execute the same stage, the router uses the user-defined load-balancing policy to distribute work efficiently. We support four routing policies: Round Robin, Least Work Outstanding, Load-Based, and Heavy-Light Load Split [30].

Load in the latter two policies can be defined using various request attributes, such as i) input context length, ii) current KV cache size (relevant when clusters differ in memory capacity), or iii) tokens remaining to be generated. Together, these yield up to nine routing strategies, and MIST’s modular router API accepts new policies with minimal effort.

2) *Model Mapper*: For different AI model instantiations, each request contains metadata specifying the target model.⁴ The router can also exploit client placement information to reduce communication costs, especially in disaggregated serving, where large KV caches must be transferred between clients.

3) *Global Communicator*: Once a routing decision is made, the global communication simulator handles data transfers between clients. It estimates communication overhead based on data size and transfer granularity (e.g., full KV cache vs. layer-wise transfer [49]), accounting for transitions between the multi-dimensional network hierarchy of HW nodes. For simulating multi-dimensional inter-client communication, MIST integrates with ASTRA-sim [11], [51], a network simulator that models multi-dimensional networks in a *contention-aware* fashion, enabling accurate modeling of communication latency and bandwidth constraints. After the data is transferred, the target client can start the stage processing.

Resource sharing across stages. Because MIST assigns every hardware component (NPU, host CPU, memory tier, storage device, and network link) a fixed capacity (memory size, bandwidth), concurrently scheduled stages contend for those resources rather than executing in isolation. When multiple transfers traverse a shared link, ASTRA-sim resolves per-message and per-link contention, so that, for example, a PD KV-cache transfer and a KV-cache retrieval issued between the same pair of racks divide the available bandwidth in proportion to their message sizes. Capacity limits are likewise enforced on the memory hierarchy: for long-context requests whose KV cache exceeds device memory,

⁴Future extensions may support adaptive model routing based on request characteristics such as complexity, quality, or priority.

MIST models offload to host memory and eviction under a configurable policy, so memory pressure created by one stage is visible to the others sharing that tier.

D. System and Software Layer: Clients

MIST has three different client types: *LLM Client* (for Prefill and Decode), *RAG Client*, and *KV Retrieval Client*, covering most modern LLM use cases. Each client in MIST is composed of a runtime Scheduler and an HW Executor. Following vLLM [37], each client operates at the granularity of a step. In the case of prefill and decode, a step corresponds to a single forward pass. During the execution of a step, new requests may arrive asynchronously but cannot preempt the ongoing computation. With the scheduled batch at each step, the runtime and cost are modeled separately for each client type. For example, KV cache retrieval depends on both the size of the KV cache and the details of the hierarchical memory architecture to estimate fetch latencies. Further details on how different hardware clusters are simulated according to different client types are provided in Sec. III-F.

E. System SW Layer: Schedulers

Each client has a scheduler that assigns requests to execute at each step.

1) *LLM Scheduler*: Since LLM inference requires multiple steps to complete a request, it requires a special scheduler. The LLM Scheduler comprises three modular components:

Batch Manager: Enforces the batching policy for each forward pass. MIST currently supports five batching strategies: *Static Batching* (FasterTransformer [46]), *Continuous Batching* (Orca/vLLM [36], [62]), *Chunked Batching* (Sarathi-Serve, FastGen [8], [29]), *Mixed Batching* (Splitwise Prefill [49]), and *Disaggregated Batching* (Splitwise/DistServe [49], [66]).

Request Queuer: orders requests by packing policy: *First-Come-First-Serve (FCFS)* or *Least Work Left*.

KV Manager: manages KV blocks for ongoing requests, prefix caching, and KV preemption to storage.

All three are modular, accepting custom batching, queuing, or scheduling strategies with minimal effort.

2) *RAG Scheduler*: For each RAG request, the scheduler needs to take two steps: convert the prompt to an embedding using DPR and retrieve relevant context from the Vector DB. If both steps are performed on the same HW (e.g., CPU), we complete both steps from the given batch before starting another batch. If there is a dedicated accelerator in the HW cluster (e.g., CPU+GPU), we use dual batching: one batch runs DPR on GPU, and the second batch runs DB lookups on CPU.

3) *KV Scheduler*: The KV Manager records KV cache blocks and their metadata, including access history and permissions. KV reads are independent, so we do not batch them by default; the request queue sets their order, and users can enable batching on clients supporting parallel reads.

F. HW Layer: Cluster Modeling

For each client in MIST, we have a corresponding HW cluster. These clusters are logical groups of HW nodes, and we simulate these through one of four methods:

TABLE II: Hardware cluster simulation methods supported in MIST.

HW Cluster	Real Execution	Empirical Runtime	Analytical Model	External Simulator
LLM Client	✓	✓	✓	✓
RAG Client	✓	✓	✗	✗
KV Client	✓	✓	✗	✗

- **Real Execution Runtime:** Execute the actual pipeline stage/step on real hardware. E.g., DecodeBatch[Req1, Req2]. We maintain a local database of execution times. If the *same* batch’s runtime is needed, we can look it up instead of running again⁵.
- **Empirical Runtime:** Profile various combinations of requests on real hardware and create a database of runtimes [14], [63]. Train an ML prediction model to create a real HW runtime database. The key difference from the previous method is that one does not need access to real HW at all times.
- **Analytical Modeling:** We can use empirical runtime-inspired analytical models to model runtime for components with a smaller contribution to overall runtime.
- **External Simulator:** We can plug in an external hardware simulator [15], [17], [59] to get the hardware runtime of unavailable systems.

Table II shows the methods supported for different hardware clusters. We detail the implementation of each hardware cluster in Sec. IV-A.

G. Output Metrics

MIST collects detailed metrics during simulation to analyze how requests are processed across the system. These metrics inform performance insights and guide system design decisions. We categorize the collected data as follows:

Individual Request Metrics: For every request, we record fine-grained statistics, including associated stage metrics (client assignment time, stage start time, stage end time). For prefill and decode, we also record per-token metrics (scheduled time, token start time, and token end time). Based on these measurements, we derive the standard inference quality metrics, including TTFT (Time-To-First-Token) and TPOT (Time-Per-Output-Token), to evaluate whether requests satisfy the target SLO (Service-Level Objective).

Scheduler-Level Metrics: These metrics track the request load queued and processed at each simulation step. These include instantaneous and average queue length, arrival volume, scheduling rate, step-wise memory load, and finished requests.

Client-Level Metrics: Each client instance maintains operational statistics through its scheduler. Tracked metrics include load and queue size at specific time points, request service rate over time, and estimated power consumption.

Coordinator Metrics: To capture holistic system behavior, we log aggregate statistics collected from individual requests, such as serviced-request information, latency breakdowns

(mean, P50, P90, P99), and communication metrics. These global insights enable comprehensive evaluation of system performance and comparative analysis across configurations and scheduling strategies.

Request Tracing and Visualization: All request-level execution details are encoded in JSON format, capturing each processed stage. This format enables seamless integration with visualization tools, such as Chrome Tracing. Together, the input datasets and output metrics constitute the foundation of our request modeling pipeline, enabling rigorous, end-to-end analysis of inference system behavior under a wide range of workload conditions.

H. Extending MIST

MIST has a highly modular and hierarchical design (illustrated in Fig. 1(b)). This allows for the decoupling of the AI workload, system & software, and hardware layers, enabling seamless integration across the stack. MIST employs base-class abstractions for pipeline stages and hardware clients, ensuring extensibility as workloads evolve. Adding a new stage only requires specifying its parameters and a latency model (Sec. III-F).

IV. MIST IMPLEMENTATION AND VALIDATION

A. Implementation of HW Layer

For the purposes of our case studies (Sec. V), we developed and integrated the following models for the hardware clusters (shown in Table II).

1) *LLM Client HW Executor:* We model prefill and decode runtime from *empirical runtime* data, collecting over 200k datapoints from vLLM across input size, batch size, chunk size (for chunked batching), and tensor parallelism (TP1/TP2/TP4/TP8). Per model, hardware, and stage, we build estimators from an ensemble of regressors, each trained on a distinct pre-specified data subset rather than stochastic samples, yielding 2.5% average error and under 1% median error, while running 20–50× faster and far cheaper than real execution.

Additionally, MIST supports *external simulators* (specifically AIConfigurator [59] and GenZ [16]) to model hypothetical HW configurations (e.g., Etched [6]) for the HW Executor of the LLM Client. MIST also provides *real execution runtime* via vLLM model execution if the hardware is available. This can be helpful in testing the potential performance of optimizations before real implementation.

2) *RAG Client HW cluster:* RAG runtime consists of i) converting the input query into a search-space embedding, ii) retrieving documents, and iii) re-ranking the top-k documents. We use the LangChain implementation to collect *empirical runtime* data for running RAG.⁶ The RAG profiling script ships with MIST, so users can register runtime data for other configurations (including a fully CPU-bound pipeline that runs embedding and lookup on the same host) by re-profiling under their own hyperparameters (document count, embedding dimension, and number of clusters probed).

⁵This is specific to the LLM inference stack; e.g., runtimes from vLLM and SGLang require separate databases.

⁶The RAG retrieval latency depends on the number of documents and the retrieval algorithm. For this work, we use the default LangChain configuration running on a 64-core x86 CPU.

3) *KV Retrieval HW Cluster*: Sec. III-E3 defines retrieval order. MIST models the KV retrieval cluster as a multi-level memory hierarchy (e.g., GPU HBM $\mapsto$ CPU $\mapsto$ SSD), with runtime data collected across levels by KV block organization and tokens retrieved, and a linear regression model trained on the resulting latencies.

Keeping runtime models current. Serving stacks and hardware evolve quickly, so MIST’s empirical models must be refreshed as new accelerators and engine releases appear. MIST ships an automated forward-pass profiling script that collects the runtime database for a given (model, hardware, engine) triple in about 2 hours; retraining the ML predictor on the collected data takes under an hour. We maintain a separate versioned database per major serving-engine release (e.g., vLLM v0.21.0 and v0.22.0, and likewise for SGLang), so software-level optimizations such as updated attention kernels or graph-capture changes are captured by re-profiling rather than by re-deriving an analytical model. Because the pre-collected databases and trained predictors are distributed with MIST, users only need to run the profiling and training scripts when targeting a custom model, engine, or accelerator. For hardware that is not yet available, this profiling step is replaced by the *external simulator* path described above.

B. Accelerated Large-scale Simulation

MIST was built to explore thousands of configurations for streaming, end-to-end inference workloads, so simulation efficiency is a core design goal: wall-clock runtime grows with both client count and simulated steps. Runtime also depends on the cluster backend: MIST’s default ML-based models are fast, while cycle-accurate backends trade runtime for fidelity. To accelerate execution, we cache per-step LLM cluster latencies in hash maps and SQL databases, which is especially effective for long decode lengths where consecutive steps have similar latencies; users can trade fidelity for speed by controlling how many consecutive steps reuse cached values. MIST thus enables rapid design-space exploration across diverse hardware and software configurations without prohibitive GPU cost.

C. MIST Validation

Next, we demonstrate MIST’s fidelity with different cases. We perform end-to-end validations (Aggregated, Disaggregated) of MIST on various platforms and models. We also validate the implementation of LLM Cluster and KV retrieval on different hardware. Broken down by pipeline stage, the mean absolute error of MIST’s models is 4.2% for prefill, 1.9% for decode, 3.2% for chunked prefill, 5.7% for KV retrieval, and 4.0% for inter-client communication. The end-to-end errors reported below are lower than several of the individual stage errors because per-stage deviations are partially uncorrelated, and because decode, the most accurately modeled stage, dominates end-to-end runtime for the workloads we study.

1) *Aggregated Validation*: We evaluate MIST using an online workload that has a Poisson distribution at the arrival rate of 5 requests per second (RPS). Using vLLM, we profile

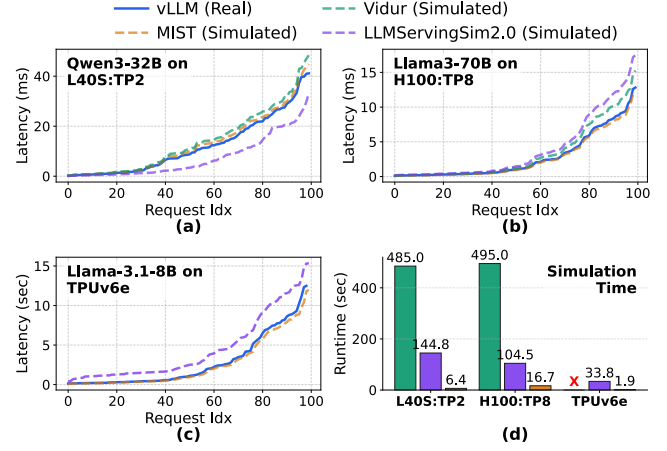


Fig. 4: (a-c) Comparing the end-to-end latency on real HW against LLM inference simulators. (d) Simulation time of different LLM inference simulators.

Llama-3-70B on **H100x8**, Qwen3-32B on **L40Sx2**, and Llama-3.1-8B on **TPUv6e**, all running the ShareGPT [10] dataset. Fig. 4 shows the end-to-end runtime of 100 requests. MIST achieves high fidelity, with less than 2% average error across different combinations of HW and models. MIST’s accuracy is significantly better than that of Vidur⁷ and LLMervingSim 2.0 because they profile only the major kernels individually. In real deployments, the vLLM serving engine uses pre-compiled graphs [55] to reduce LLM inference forward pass time. This cannot be captured when reconstructing the latency from individual operators. Additionally, since both Vidur and LLMervingSim 2.0 use operator-level ML predictors, the error is aggregated across layers and tokens, hence impacting end-to-end latency. Fig. 4(d) shows the simulation time for each LLM inference simulator running the three benchmarks. Vidur requires a significant amount of time: about 400 seconds for pre-training before each run. MIST achieves the fastest simulation runtime while maintaining fidelity due to (i) smarter caching of HW cluster runtimes and (ii) avoiding repeated loading of individual kernels by capturing inference forward-pass latency at the granularity of precompiled graphs.

2) *Disaggregated Validation*: To validate the effectiveness of disaggregated client serving, we utilize real request traces collected from the Azure platform [13]. As a reference, we compare against Splitwise [56], which implements disaggregated serving on top of vLLM for small-scale evaluation. While direct experiments on 80- and 160-GPU clusters are not available to us, splitwise-sim extrapolates from their vLLM prototype to evaluate the same architecture at larger scales. We therefore treat it as a valid simulation-based baseline. We simulate two different models (Llama-2-70B and Bloom-176B) on an **80xH100** node configuration with 8 prefill clients and 2 decode clients under different request distributions (RPS=20 and RPS=40). Across use cases, we observe minor differences in modeling (< 6% maximum error) as shown in Fig. 5. This difference in runtime arises

⁷Vidur does not support Google TPUs.

from communication, as splitwise-sim employs a dummy link-based communication model with a specified lower-bound bandwidth. In contrast, we use ASTRA-sim to model client communication, which introduces slight differences in overall runtime.⁸

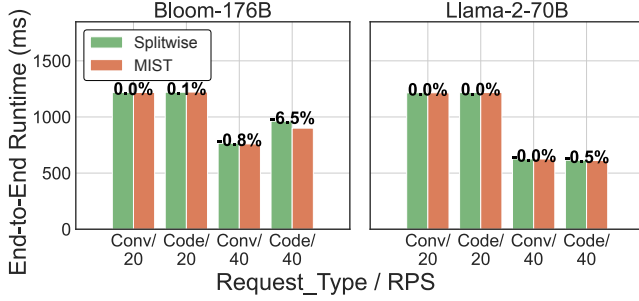


Fig. 5: End-to-end validation results comparing Splitwise and MIST on an 80-GPU system configured with TP8.

3) *LLM Cluster Modeling Validation*: To evaluate the accuracy of our ML-Based LLM Cluster Modeling (Sec. IV-A1), we compare MIST’s per-step runtime predictions and Vidur’s predicted runtimes against ground-truth values from vLLM running Llama-2-70B on **H100x8**. Fig. 6a compares the error of MIST and Vidur. MIST’s ML-Assisted LLM cluster modeling accuracy is significantly better than Vidur’s runtime prediction. Upon further inspection, we identified two main causes of error in Vidur: (i) using operator-level ML predictors causes error to accumulate across layers and operators, and (ii) Vidur ignores kernel launch overheads and smaller operations. MIST uses a predictor trained on the LLM engine’s model execution time, which accounts for all the CPU and kernel launch overheads and avoids error cascading.

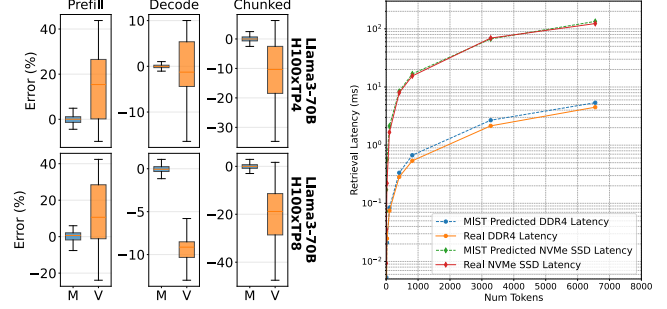
4) *KV Retrieval Cluster Modeling Validation*: To evaluate the accuracy of our KV retrieval cluster in MIST, we compare the measured and MIST predicted latencies for both NVMe SSDs and DDR4 main memory. Sequential reads were performed using the `fio` benchmarking tool [26] across a range of block sizes, from 256 KB to 1GB. Average read latency was recorded for each size. As shown in Fig. 6b, MIST predicts the trend of measured latency closely across both memory tiers.

5) *RAG Cluster Modeling Validation*: We validate RAG performance against Google’s RAGO cost model [32], observing strong correlation with less than 15% error on large vector databases containing over 100 million vectors.

V. DESIGN-SPACE EXPLORATION CASE STUDIES

We demonstrate the value of MIST across two use cases: (i) exploring LLM inference deployments on heterogeneous hardware and (ii) KV cache storage hardware design exploration. Both are intended as *illustrations of the design-space exploration MIST enables*, rather than as deployment

⁸For this validation we used splitwise-sim’s LLM runtime predictor, as our aim is to show the correctness of request orchestration by the global coordinator.



(a) Comparison of MIST (M) and Vidur (V) predicted runtime against vLLM runtime for 1K requests (150K tokens). (b) Validating KV Retrieval cluster against real memory devices. We use Llama-3-70B KV cache retrieval as an example.

Fig. 6: Hardware Cluster Validation

recommendations. Each conclusion below holds under a specific set of assumptions (accelerator and cloud pricing, network bandwidth, workload mix, and, for unreleased hardware, projected device capabilities and the maturity of the supporting software stack), and we state those assumptions alongside the results so that readers can judge which findings would transfer to their own setting and which would not.

A. Exploring LLM Inference on Heterogeneous Hardware

In this section, we demonstrate how MIST can efficiently guide the deployment of multi-stage LLM use cases with diverse token distributions across off-the-shelf heterogeneous hardware. Modern LLM serving engines [2], [37], [65] expose a wide variety of configurable parameters (tuning knobs) that significantly impact deployment efficiency. Using MIST, we systematically explore combinations of these knobs to maximize generated tokens per second per dollar, for requests that meet their latency SLOs.

Search Space: We search the deployment space across the following configurable dimensions: (i) **hardware SKUs** spanning Nvidia (H200-SXM, B200-SXM, GB300), AMD (MI350X, MI355X), Google (TPUv6e, TPUv7), and Etched (Sohu)⁹; (ii) **model parallelism** with tensor parallelism $TP \in \{1, 2\}$; (iii) **batching strategies** including chunked and disaggregated prefill; and (iv) **prefill-to-decode client ratios** for disaggregated deployments. We use cloud rental pricing¹⁰ (per accelerator per hour) to estimate search and deployment costs: H200-SXM \$6.31, B200-SXM \$8.60, GB300 \$12.00, Etched Sohu \$12.00, MI350X \$7.50, MI355X \$9.50, TPUv6e \$3.00, and TPUv7 \$8.00. We use the roofline estimator in Nvidia’s AConfigurator [59] as the LLM Cluster runtime simulator, ensuring all HW SKUs are modeled with the same backend and based on their raw capabilities.

Assumptions. The configurations reported below are *exploration outcomes under a stated set of assumptions*,

⁹We assume Etched Sohu has similar memory and network characteristics to Nvidia GB300, with $5\times$ the compute units [50].

¹⁰We use pricing from CoreWeave for Nvidia GPUs, TensorWave for AMD GPUs, and Google Cloud for TPUs, as listed in November 2025. We estimate prices for publicly unavailable HW from similarly sized, similarly specified chips.

TABLE III: LLM use cases with different multi-stage pipelines.

Use Case	Datasets	Stages	# Queries	# Prefill Tokens			# Decode Tokens		
				Mean	Median	P95	Mean	Median	P95
Chat Conversation	Narrative QA [35]	Prefill Decode	288	252	484	1597	218	232	332
Code Generation	Github-Code [52]	KV Retrieval Prefill Decode	465k	26979	16204	83920	1088	329	2753

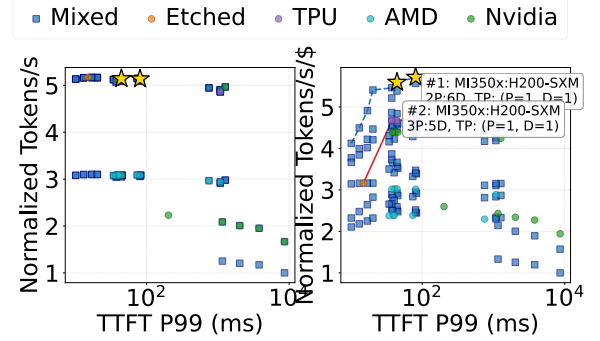
not deployment recommendations, and we make those assumptions explicit here. First, GB300, TPUv7, MI355X, and Etched Sohu are not publicly benchmarked at the time of writing; Sohu in particular is a *projection* built from public analysis [50] rather than vendor-confirmed silicon, and its rental price is an estimate by analogy to similarly sized parts. Second, tokens/\$ depends directly on rental prices, which move frequently and differ across providers and commitment terms; a sustained shift in the relative price of two SKUs can reorder the cost-efficiency ranking even when the underlying performance model is unchanged. The performance ranking, which depends only on the hardware models, is unaffected by such shifts. Third, the heterogeneous configurations assume the KV cache can be moved between vendors without conversion cost. In practice, cross-vendor transfers may incur additional latency for KV-cache dtype, layout, and FP8-format translation. We do not add this penalty to the numbers below; MIST’s KV-transfer stage accepts it as a configurable latency term, which lets a user ask the inverse question: how large a translation penalty a given cross-vendor pairing can absorb before it loses to the best single-vendor alternative.

We evaluate MIST across two representative multi-stage use cases (Table III). Fig. 8 compares the best output generation throughput/\$ and deployment cost of the top configuration across vendors for both use cases.

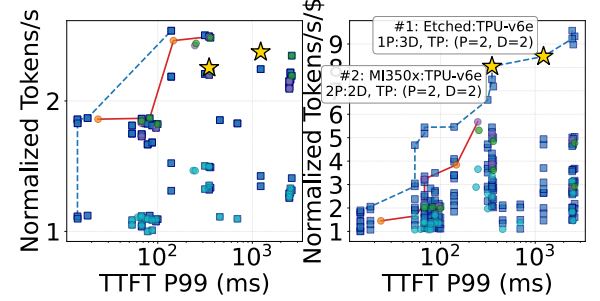
1) *Chat Conversation: Use case: Regular chat-based conversation with user prompts.* We use the Narrative QA dataset [35] as a representative workload, where queries pose questions about movies and books to elicit free-form answers.

Experimental Setup: We use Qwen3-32B [54] and search over deployment configurations (up to 8 nodes) at a query injection rate of 100 RPS. Given the streaming chat generation use case, we enforce a TTFT SLO of $P99 \leq 250$ ms.

Our search yields 1,567 valid configurations. Fig. 7a shows the token generation throughput across all explored deployments. Among single-vendor configurations, TPUv7 (2P:6D) and B200-SXM (2P:6D) achieve the highest raw throughput owing to their superior TFLOPS and memory bandwidth. However, optimizing for generated tokens per second per dollar favors a heterogeneous, multi-vendor mix: 2xMI350X (prefill) paired with 6xH200-SXM (decode). Under the pricing and hardware assumptions stated above, this configuration produces 22% more tokens/\$ at 17.4% lower per-hour deployment cost than the best single-vendor configuration, illustrating that cross-vendor hardware pairings are worth including in the search space, since they can expose cost-efficiency operating points unavailable within any single vendor’s product line.



(a) Chat Conversation.



(b) Code generation with High Prefix-KV Reuse.

Fig. 7: Search-space exploration for different use cases. Output generation throughput is normalized against the lowest throughput configuration. Stars mark the two best configurations found by MIST.

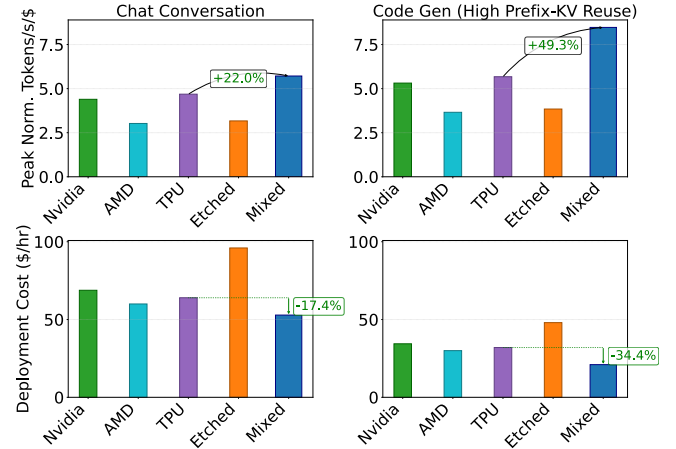


Fig. 8: LLM deployment DSE for different use cases.

2) *Code Generation with High Prefix-KV Reuse: Use case: LLM-assisted code generation has become a common practice in modern software engineering. When generating code with multi-file dependencies, the relevant files must be supplied as context for the prompt. The KV cache for most files can be pre-computed offline and reused across requests via prefix caching [4]. We use the GitHub Code dataset [52]*

to simulate requests that edit small files using pre-calculated KV context from other files in the same repository.

Experimental Setup: We use Qwen3-32B [54] and search over deployment configurations (up to 8 nodes) at 100 RPS. Given the large historical context window, we enforce a TTFT SLO of $P99 \leq 1.5$ s.

The search yields 1,878 valid configurations (Fig. 7b). While most requests benefit from a large fraction of pre-computed KV context, roughly 10% of requests still require full KV recomputation over long context sequences (so-called *whale* requests), making high prefill compute capacity critical. Among single-vendor deployments, the Etched Sohu chip achieves the highest raw throughput owing to its exceptional compute density. However, optimizing for generated tokens/\$ reveals a more nuanced picture: a heterogeneous Etched + TPUv6e configuration (1 prefill client $\times$ TP2 + 3 decode clients $\times$ TP2) delivers the best cost-efficiency. The intuition is clear: Etched Sohu absorbs the compute-intensive KV recomputation for whale requests, while TPUv6e, which offers the highest memory bandwidth per dollar among the candidates, handles the decode stage efficiently. Optimizing for generated tokens per second per dollar, and again under the assumptions stated above, this Etched-TPUv6e pairing outperforms the best single-vendor alternative, TPUv7 with chunked batching (4 clients, TP2), generating **49.3% more tokens/\$** at **34.4% lower deployment cost**.

Software support required in practice. Even where MIST identifies a favorable cross-vendor pairing, today’s serving stacks cannot execute it directly. vLLM’s V1 KV-connector architecture already separates prefill and decode into independent instances, but every connector assumes a single homogeneous platform, non-interoperable collective libraries, and matching KV-cache dtype and layout. Splitting one request across vendors would require changes in five areas: the KV connector and transport layer, the platform abstraction, KV-cache layout and FP8-format translation, the executor/worker model, and the disaggregation scheduler; none of these are upstream today, though cross-vendor prefill/decode disaggregation has been demonstrated outside of upstream vLLM by staging the KV cache through host memory over RDMA [44]. This is precisely the setting in which we see MIST being useful: it lets a team estimate whether a given cross-vendor pairing would repay that engineering investment before committing to it.

B. MIST for KV Cache Storage Hardware Design Exploration

Efficient KV cache retrieval is crucial for LLM serving, as faster KV retrieval means lower TTFT.

Key Question: Where should we store the KV cache?

Target Use Case: The analysis addresses two principal scenarios: *i) private key-value (KV) caches*, designed for individual user contexts (e.g., ChatGPT). These private KV caches can be accessed by future queries from the same user. *ii) shared KV caches*, often used in multi-user settings (e.g., enterprise AI or shared codebases) to enable multiple users to access a large corpus ($\sim O(10^{10})$ tokens) of documents or

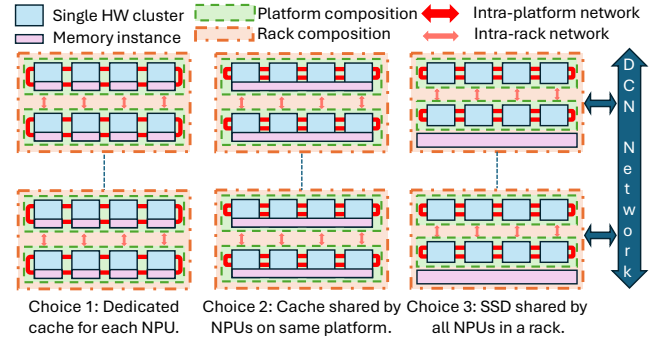


Fig. 9: DESIGN CHOICES FOR KV CACHE STORAGE.

code. Generally, these caches have hotspots that are accessed much more frequently than the rest of the KV cache.

Hardware design space: Current AI serving racks with storage memory use different caching solutions. We classify these into three choices (Fig. 9): (1) a dedicated cache per client, inspired by dedicated DDR on the GPU host; (2) a platform-level shared cache with shared access by 2-8 clients; and (3) a rack-level shared cache with shared access by 32-64 clients, inspired by SSD memory in each rack. These racks are connected by a data center network (DCN). Each configuration offers distinct trade-offs among capacity, bandwidth, and latency.

Experimental Setup: We evaluate these architecture choices on 256 GPUs (128 clients with H100:TP2) distributed across 4 racks, interconnected via NVLink and PCIe. The experiments simulate short (4K tokens) and long (24K tokens) KV cache retrieval workloads for both private and shared contexts. Requests follow the AzureConv trace [13]. We examine five distinct storage architectures, detailed in Table IV, ranging from dedicated local memory to a baseline of full recomputation. End-to-end request serving latency serves as the primary evaluation metric to assess system performance across these configurations.

TABLE IV: Storage Architecture Configs. Bandwidth (BW) represents the speed of data retrieval or transfer.

Case	Architecture Type	Capacity / BW	Access / Notes
A	Dedicated Per-Client (Choice 1)	1 TB / 128 GB/s	LPDDR-based; Local access
B	Platform-Shared (Choice 2)	4 TB / 32 GB/s	Shared by 4 clients
C	Rack-Shared (Choice 3)	32 TB / 2 GB/s	Shared by 32 clients
D	Rack-Shared (Choice 3) + DCN	32 TB / 2 GB/s	Inter-rack transfer @ 128GB/s
E	No Cache	N/A	Full KV recomputation

Fig. 10 shows the end-to-end latency CDF for the design space. **For private KV caches**, we find that the platform-level shared cache (Case B) or dedicated LPDDR (Case A) offers the best P90 request latency for both short and long requests. This is because, under normally distributed request arrivals, clients’ loads remain balanced. In contrast, the performance of Cases C, D, and E degrades as KV context length increases because of lower bandwidth and higher latency.

For shared KV caches, storing the KV cache at the platform level (B) or in host memory (A) is a very poor

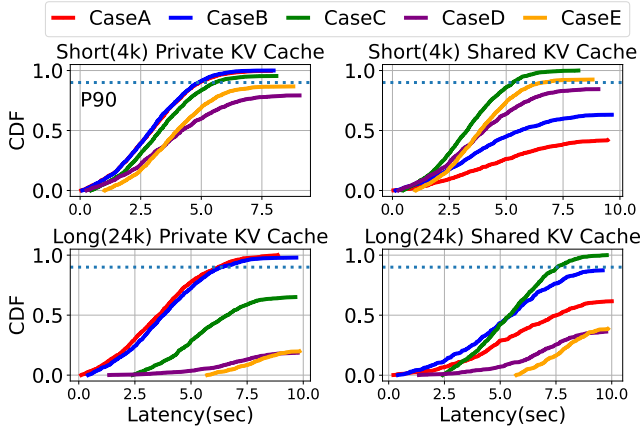


Fig. 10: Comparing different platform architectures for KV cache storage, serving 128 clients of Llama-3.1-70B on 4 HGX racks (64xH100 per rack).

choice as it leads to severe imbalance in the system; instead, rack-level SSD storage (Case C) is the best option, as it delivers higher capacity and maintains acceptable latency despite a modest reduction in per-client bandwidth. For larger KV caches, storing them at the platform level (B) becomes attractive due to superior memory BW, resulting in lower latency.

- For the private-KV workloads we model, dedicated (A) or platform-level shared cache (B) balances speed and resource sharing best.
- For shared global KV caches at this rack scale, rack-level shared cache (C) performs best: it provides low-latency access and efficient inter-client sharing.

VI. RELATED WORK

Table V compares various frameworks simulating the LLM inference serving stack. LLMCompass [64] and GenZ [16] provide detailed modeling capabilities with optimizations but are limited to single-client configurations. Vidur [7] supports multi-client discrete-event simulation with ML-based runtime prediction, but its analytical roofline models abstract away framework-specific behavior; it assumes homogeneous clients and real hardware with aggregated batching only, lacking disaggregated prefill-decode support. Splitwise-sim [45] models three pools of hardware clients representing prefill, decode, and mixed pools. It does not model chunked batching or end-to-end pipeline stages. AIConfigurator [59] automates configuration selection across serving frameworks (vLLM, TensorRT-LLM) from extensive real GPU measurements. However, it assumes fixed batch sizes and sequence lengths, so it cannot evaluate streaming requests from traces such as Azure [13], and models only prefill and decode on independent GPUs, without heterogeneous network routing or multi-stage pipelines. Notably, MIST integrates AIConfigurator as an *external simulator*, leveraging its extensive measurement dataset collected across diverse hardware platforms. LLMservingSim 2.0 [20] adds heterogeneous accelerator pools (e.g., TPUs, CIMS), profile-based execution, and finer modeling of KV movement, prefix reuse, and

TABLE V: COMPARISON OF LLM SERVING SIMULATORS

Simulator	Workload			SW		HW		
	TR	PS	MM	RR	B	HT	MP	TKV
AIConfigurator [59]	–	–	–	–	✓	△	△	–
Vidur [7]	✓	–	–	✓	△	–	–	–
LLMservingSim [21]	✓	–	–	–	△	△	△	✓
Splitwise-sim [45]	✓	–	–	–	△	△	△	–
LLMservingSim 2.0 [20]	✓	–	–	✓	✓	✓	△	✓
MIST (ours)	✓	✓	✓	✓	✓	✓	✓	✓

TR: Trace Replay, **PS:** Pipeline stages (end-to-end inference workflow beyond prefill/decode, such as KV Cache Retrieval and RAG), **MM:** Multi-Model Serving, **RR:** Multi-Node Routing, **B:** Batching (Agg, PD Disagg), **HT:** Heterogeneous Systems, **MP:** Multiple HW Pools, **TKV:** Tiered KV Cache Hierarchy.
 ✓: fully supported, –: not supported, △: limited or partial support.

expert-parallel MoE routing. Despite improved hardware diversity and routing, it remains single-model and does not model multi-stage pipelines. Overall, Vidur, Splitwise-sim, AIConfigurator, and LLMservingSim 2.0 all fall short in modeling advanced, multi-stage LLM inference.

VII. CONCLUSION

We present MIST (<https://github.com/MIST-Simulator/MIST>), an event-driven simulation framework modeling the full LLM serving stack across workload, system-software, and hardware layers, achieving end-to-end fidelity within 6% of real deployments. Through design-space exploration case studies, MIST identifies multi-vendor PD-disaggregated configurations yielding up to 49.3% higher tokens/\$ than homogeneous deployments under our stated pricing and hardware assumptions, and shows that KV storage hierarchy and eviction policy choices significantly shape tail latency.

Limitations. MIST’s cost-efficiency conclusions inherit the assumptions of their inputs: rental prices that fluctuate across providers and over time, projected capabilities for accelerators that are not yet publicly benchmarked, and the absence of a cross-vendor KV-cache translation penalty that today’s serving stacks would incur. The case studies should therefore be read as demonstrations of what the framework can explore, not as deployment guidance. Reported fidelity is likewise established for the platforms, models, and stages we validate in Sec. IV-C, with RAG and KV retrieval modeled at coarser granularity than the prefill and decode paths. Our roadmap includes SGLang support, additional hardware backends, and agentic workload pipelines. Looking ahead, MIST can guide future hardware roadmap decisions, help develop adaptive cross-stage schedulers, and model emerging multi-agent LLM orchestration patterns.

ACKNOWLEDGMENT

We have used LLM tools (Claude, Gemini, Grammarly) for proofreading, grammar correction, and continuity checking across sections. All the AI-generated text has been verified by the authors. This work was supported in part by CoCoSys, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and also by the Georgia Research Alliance based in Atlanta, Georgia under award GRA.25.071.GT.01.a. We also thank the reviewers for their valuable feedback, which improved

this work, and Ishita Vohra for her help in running baseline validation experiments.

REFERENCES

- [1] “Automatic prefix caching (vLLM),” [Online; accessed 2025-04-11]. [Online]. Available: https://docs.vllm.ai/en/latest/design/v1/prefix_caching.html
- [2] “Dynamo inference framework (NVIDIA Developer),” [Online; accessed 2025-04-10]. [Online]. Available: <https://developer.nvidia.com/dynamo>
- [3] “Triton TensorRT-LLM backend.” [Online]. Available: <https://github.com/triton-inference-server/tensorrtllm%5Fbackend>
- [4] “What is prefix caching? A beginner’s guide (AI Resources),” [Online; accessed 2025-04-11]. [Online]. Available: <https://www.modular.com/ai-resources/what-is-prefix-caching-a-beginner-s-guide>
- [5] “Choose the k-NN algorithm for your billion-scale use case with OpenSearch (AWS Big Data Blog),” 9 2022, [Online; accessed 2025-04-11]. [Online]. Available: <https://aws.amazon.com/blogs/big-data/choose-the-k-nn-algorithm-for-your-billion-scale-use-case-with-opensearch/>
- [6] “Etched is making the biggest bet in AI,” 2024, (Accessed on 2/21/2025). [Online]. Available: <https://www.etched.com/announcing-etched>
- [7] A. Agrawal, N. Kedia, J. Mohan, A. Panwar, N. Kwatra, B. Gulavani, R. Ramjee, and A. Tumanov, “Vidur: A large-scale simulation framework for LLM inference,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2405.05465>
- [8] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, “Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2403.02310>
- [9] J. Ahn, R. Verma, R. Lou, D. Liu, R. Zhang, and W. Yin, “Large language models for mathematical reasoning: Progresses and challenges,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2402.00157>
- [10] anon8231489123, “ShareGPT_Vicuna datasets at Hugging Face,” 2024. [Online]. Available: https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered
- [11] Astra-Sim, “ns-3 network backend (ASTRA-sim 2.2 documentation),” [Online; accessed 2025-06-19]. [Online]. Available: <https://astra-sim.github.io/astra-sim-docs/network-backend/ns3-network-backend.html>
- [12] K. Aubrey, “Inside the NVIDIA Vera Rubin platform: Six new chips, one AI supercomputer (NVIDIA Technical Blog),” 1 2026, [Online; accessed 2026-04-07]. [Online]. Available: <https://developer.nvidia.com/blog/inside-the-nvidia-rubin-platform-six-new-chips-one-ai-supercomputer/>
- [13] M. Azure, “Azure Public Dataset: Azure LLM Inference Trace 2023,” <https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2023.md>, 2023, accessed: 2025-04-10.
- [14] A. Bambhaniya, G. Jeong, J. Park, J. Yu, J. Lee, P. Wang, C. Kim, C. Tang, and T. Krishna, “Scaling multi-node mixture-of-experts inference using expert activation patterns,” 2026. [Online]. Available: <https://arxiv.org/abs/2604.23150>
- [15] A. Bambhaniya, R. Raj, G. Jeong, S. Kundu, S. Srinivasan, S. Subramanian, M. Elavazhagan, M. Kumar, and T. Krishna, “Demystifying AI platform design for distributed inference of next-generation LLM models,” 2025. [Online]. Available: <https://arxiv.org/abs/2406.01698>
- [16] A. Bambhaniya, R. Raj, G. Jeong, S. Kundu, S. Srinivasan, S. Subramanian, M. Elavazhagan, M. Kumar, and T. Krishna, “Demystifying AI platform design for distributed inference of next-generation LLM models,” 2025. [Online]. Available: <https://doi.org/10.48550/arXiv.2406.01698>
- [17] A. R. Bambhaniya, S.-C. Kao, G. Jeong, S. Subramanian, A. Yazdanbakhsh, and T. Krishna, “Demystifying the cost versus benefits of sparse large language model acceleration,” *IEEE Micro*, vol. 46, no. 2, pp. 38–50, 2026.
- [18] Y. Cheng, K. Du, J. Yao, and J. Jiang, “Do large language models need a content delivery network?” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2409.13761>
- [19] A. Chirkin, “Accelerating vector search: NVIDIA cuVS IVF-PQ part 1, deep dive (NVIDIA Technical Blog),” 7 2024, [Online; accessed 2025-03-08]. [Online]. Available: <https://developer.nvidia.com/blog/accelerating-vector-search-nvidia-cuvs-ivf-pq-deep-dive-part-1/>
- [20] J. Cho, H. Choi, G. Heo, and J. Park, “LLMServingSim 2.0: A unified simulator for heterogeneous and disaggregated LLM serving infrastructure,” 2026. [Online]. Available: <https://doi.org/10.48550/arXiv.2602.23036>
- [21] J. Cho, M. Kim, H. Choi, G. Heo, and J. Park, “LLMServingSim: A HW/SW co-simulation infrastructure for LLM inference serving at scale,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2408.05499>
- [22] S. K. Dam, C. S. Hong, Y. Qiao, and C. Zhang, “A complete survey on LLM-based AI chatbots,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2406.16937>
- [23] A. Francis, “Gimlet Labs targets AI’s inference cost problem,” 3 2026, [Online; accessed 2026-04-07]. [Online]. Available: <https://www.channelinsider.com/ai/gimlet-labs-80m-series-a-ai-inference/>
- [24] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, “Retrieval-augmented generation for large language models: A survey,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2312.10997>
- [25] G. Gerganov, “ggerganov/llama.cpp: Port of Facebook’s Llama model in C/C++,” [Online]. Available: <https://github.com/ggerganov/llama.cpp>
- [26] Google Cloud, “Benchmark Persistent Disk performance on a Linux VM,” <https://cloud.google.com/compute/docs/disks/benchmarking-pd-performance-linux>, 2025, last updated: 2025-08-07; Accessed: 2025-08-20.
- [27] S. Hao, Y. Gu, H. Ma, J. J. Hong, Z. Wang, D. Z. Wang, and Z. Hu, “Reasoning with language model is planning with world model,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2305.14992>
- [28] Hao AI Lab, “DistServe retrospective: Attention-FFN disaggregation, the next frontier,” <https://haailab.com/blogs/distserve-retro/>, 2025, [Online; accessed 2026-08-25].
- [29] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin, A. Bakhtiari, L. Kurilenko *et al.*, “DeepSpeed-FastGen: High-throughput text generation for LLMs via MII and DeepSpeed-Inference,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2401.08671>
- [30] K. Jain, A. Parayil, A. Mallick, E. Choukse, X. Qin, J. Zhang, Íñigo Goiri, R. Wang, C. Bansal, V. Rühle, A. Kulkarni, S. Kofsky, and S. Rajmohan, “Intelligent router for LLM workloads: Improving performance through workload-aware load balancing,” 2025. [Online]. Available: <https://doi.org/10.48550/arXiv.2408.13510>
- [31] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A survey on large language models for code generation,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2406.00515>
- [32] W. Jiang, S. Subramanian, C. Graves, G. Alonso, A. Yazdanbakhsh, and V. Dadu, “RAGO: Systematic performance optimization for retrieval-augmented generation serving,” 2025. [Online]. Available: <https://arxiv.org/abs/2503.14649>
- [33] J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with GPUs,” 2017. [Online]. Available: <https://doi.org/10.48550/arXiv.1702.08734>
- [34] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W. tau Yih, “Dense passage retrieval for open-domain question answering,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online: Association for Computational Linguistics, 2020, pp. 6769–6781. [Online]. Available: <https://doi.org/10.18653/v1/2020.emnlp-main.550>
- [35] T. Kočiský, J. Schwarz, P. Blunsom, C. Dyer, K. M. Hermann, G. Melis, and E. Grefenstette, “The NarrativeQA reading comprehension challenge,” *Transactions of the Association for Computational Linguistics*, vol. 6, pp. 317–328, 2018. [Online]. Available: https://doi.org/10.1162/tacl_a_00023
- [36] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, “Efficient memory management for large language model serving with PagedAttention,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 611–626. [Online]. Available: <https://doi.org/10.1145/3600006.3613165>
- [37] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, “Efficient memory management for large language model serving with PagedAttention,” in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2309.06180>

- [38] Y. Leviathan, M. Kalman, and Y. Matias, “Fast inference from transformers via speculative decoding,” 2023. [Online]. Available: <https://arxiv.org/abs/2211.17192>
- [39] Y. Li, F. Wei, C. Zhang, and H. Zhang, “EAGLE: Speculative sampling requires rethinking feature uncertainty,” 2024, [Online]. Available: <https://arxiv.org/abs/2401.15077>.
- [40] H. Lightman, V. Kosaraju, Y. Burda, H. Edwards, B. Baker, T. Lee, J. Leike, J. Schulman, I. Sutskever, and K. Cobbe, “Let’s verify step by step,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2305.20050>
- [41] C. Lin, Z. Han, C. Zhang, Y. Yang, F. Yang, C. Chen, and L. Qiu, “Parrot: efficient serving of LLM-based applications with semantic variable,” in *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’24. USA: USENIX Association, 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2405.19888>
- [42] Y. Liu, H. Li, Y. Cheng, S. Ray, Y. Huang, Q. Zhang, K. Du, J. Yao, S. Lu, G. Ananthanarayanan *et al.*, “CacheGen: KV cache compression and streaming for fast large language model serving,” in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 38–56. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.07240>
- [43] Microsoft, “GitHub Copilot · your AI pair programmer.” [Online]. Available: <https://github.com/features/copilot>
- [44] Moreh, “Cross-vendor disaggregated inference: GPT-OSS-120B across NVIDIA H100 and AMD MI300X,” <https://moreh.io/technical-report/cross-vendor-disaggregated-inference-gpt-oss-120b-across-nvidia-h100-and-amd-mi300x-260318/>, 2026, [Online; accessed 2026-08-25].
- [45] Mutinifni, “SplitwiseSim: LLM Serving Cluster Simulator,” <https://github.com/Mutinifni/splitwise-sim>, 2024, accessed: 2025-04-10.
- [46] NVIDIA, “GitHub - NVIDIA/FasterTransformer: Transformer related optimization, including BERT, GPT,” [Online; accessed 2025-04-10]. [Online]. Available: <https://github.com/NVIDIA/FasterTransformer>
- [47] NVIDIA, “TensorRT-LLM,” 2023. [Online]. Available: <https://github.com/NVIDIA/TensorRT-LLM>
- [48] OpenAI, “ChatGPT.” [Online]. Available: <https://openai.com/chatgpt>
- [49] P. Patel, E. Choukse, C. Zhang, Ñiño Goiri, A. Shah, S. Maleki, and R. Bianchini, “Splitwise: Efficient generative LLM inference using phase splitting,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2311.18677>
- [50] Peak FLOPS, “Breaking down Etched’s Sohu,” 4 2026, [Online; accessed 2026-04-09]. [Online]. Available: <https://wafer.substack.com/p/breaking-down-etcheds-sohu>
- [51] S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, “ASTRA-SIM: Enabling SW/HW co-design exploration for distributed DL training platforms,” in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020. [Online]. Available: <https://www.doi.org/10.1109/ISPASS48437.2020.00018>
- [52] N. Saga, “nick007x/github-code-2025 · datasets at Hugging Face,” 10 2025, [Online; accessed 2025-11-17]. [Online]. Available: <https://huggingface.co/datasets/nick007x/github-code-2025>
- [53] G. Team, “Gemini: A family of highly capable multimodal models,” 2025. [Online]. Available: <https://doi.org/10.48550/arXiv.2312.11805>
- [54] Q. Team, “Qwen3 technical report,” 2025. [Online]. Available: <https://doi.org/10.48550/arXiv.2505.09388>
- [55] vllm, “CUDA graphs (vLLM),” 3 2026, [Online; accessed 2026-04-07]. [Online]. Available: https://docs.vllm.ai/en/stable/design/cuda_graphs/
- [56] vLLM contributors, “Add Splitwise Implementation to vLLM,” <https://github.com/vllm-project/vllm/pull/2809>, 2024, accessed: 2025-04-10.
- [57] J. Wang, “Cerebras,” 3 2026, [Online; accessed 2026-04-07]. [Online]. Available: <https://www.cerebras.ai/blog/cerebras-is-coming-to-aws>
- [58] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, “Chain-of-thought prompting elicits reasoning in large language models,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2201.11903>
- [59] T. Xu, Y. Liu, X. Lu, Y. Zhao, X. Zhou, A. Feng, Y. Chen, Y. Shen, Q. Zhou, X. Chen, I. Sherstyuk, H. Li, R. Thakkar, B. Hamm, Y. Li, X. Huang, W. Wu, A. Shanbhag, H. Kim, C. Chen, and J. Lai, “AIConfigurator: Lightning-fast configuration optimization for multi-framework LLM serving,” 2026. [Online]. Available: <https://doi.org/10.48550/arXiv.2601.06288>
- [60] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, “SWE-agent: Agent-computer interfaces enable automated software engineering,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2405.15793>
- [61] J. Yao, H. Li, Y. Liu, S. Ray, Y. Cheng, Q. Zhang, K. Du, S. Lu, and J. Jiang, “CacheBlend: Fast large language model serving with cached knowledge fusion,” *arXiv preprint arXiv:2405.16444*, 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2405.16444>
- [62] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, “Orca: A distributed serving system for Transformer-Based generative models,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 2022, pp. 521–538. [Online]. Available: <https://www.usenix.org/conference/osdi22/presentation/yu>
- [63] Z. Yu, Y. Guan, Z. Yu, C. Zhou, Z. Hu, S. Pei, Y. Kang, Y. Ding, and P.-A. Tsai, “Patterns behind chaos: Forecasting data movement for efficient large-scale MoE LLM inference,” 2026. [Online]. Available: <https://arxiv.org/abs/2510.05497>
- [64] H. Zhang, A. Ning, R. Prabhakar, and D. Wentzlaff, “A hardware evaluation framework for large language model inference,” *arXiv preprint arXiv:2312.03134*, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2312.03134>
- [65] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, “SGLang: Efficient execution of structured language model programs,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2312.07104>
- [66] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, “DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2401.09670>