KEN: An Execution Engine for Unstructured Database Systems

Ferdi Kossmann
MIT CSAIL
ferdik@csail.mit.edu

Ziniu Wu
MIT CSAIL
ziniuw@csail.mit.edu

Alex Turk
Intel Labs
alex.turk@intel.com

Nesime Tatbul
MIT CSAIL, Intel Labs
tatbul@csail.mit.edu

Lei Cao
University of Arizona
lcao@csail.mit.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

Unstructured database management systems (UDBMSes) leverage machine learning to apply the relational model to modalities beyond tables, such as documents, images and videos. Queries in a UDBMS consist of *logical operators* for which the UDBMS chooses *physical implementations* (e.g., different models) with the goal to optimize both query latency and accuracy. However, many operators only expose a coarse-grained set of implementations, forcing the UDBMS to excessively sacrifice either accuracy or latency without middle-ground options. For example, an entity matching operator can either be implemented through small, specialized models or large, general-purpose models (e.g., Large Language Models) — while the former struggles on challenging inputs, the latter is more accurate but incurs orders of magnitude more computation. In this work, we aim to address this issue with *model cascades*, which seek to process "easy" inputs with small models and only resort to large models when necessary. However, cascades incur higher memory usage and additional data transfer between GPU memory and arithmetic units, which often slows queries compared to single models. To address this issue, we introduce Ken, a dedicated UDBMS *execution engine* that dynamically adapts its use of cascades to the query load, and optimizes the GPU placement and invocation scheduling of the cascade models. Compared to baselines, Ken achieves $1.7\times - 3.3\times$ latency reductions when combining similar models for a single operator, and 122× latency reductions when combining models with orders of magnitude size differences in a multi-operator query.

PVLDB Reference Format:
Ferdi Kossmann, Ziniu Wu, Alex Turk, Nesime Tatbul, Lei Cao, and Samuel Madden. KEN: An Execution Engine for Unstructured Database Systems. PVLDB, 19(5): 902 - 916, 2026.
doi:10.14778/3796195.3796204

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ferdiko/ken_submission.

1 INTRODUCTION

Relational database systems have matured into indispensable tools for managing structured data. However, most of the world's data is unstructured, such as documents, images, videos, and more. To query such data, recent work introduces *unstructured database management systems* (UDBMSes) that employ machine learning (ML) models to handle a wide range of data modalities [9, 11, 12, 14, 23, 38, 39, 51, 54, 59, 62, 64, 68, 69, 89]. As in relational databases, queries to UDBMSes are composed of logical operators, each of which can be implemented in multiple ways (for example, using different ML models). UDBMS query optimizers choose an implementation for each operator but, unlike in traditional systems, their choices don't only affect the runtime of a query but also the *accuracy* of the query output. As a result, UDBMSes must navigate a trade-off between latency and accuracy, with the optimal balance depending on the application's requirements.

For example, imagine a UDBMS application that lets users upload PDF invoices, which are then parsed to populate a structured database. The query may involve two operators: (1) parsing the document to extract relevant fields, and (2) matching extracted entities (e.g., vendor names) against a reference database. For either of the operators, the UDBMS may choose between different general-purpose Large Language Models (LLMs). Alternatively, the UDBMS might employ specialized models, such as GOT-OCR [83] for parsing and AnyMatch [98] for entity matching.

In this example, state-of-the-art LLMs achieve higher accuracy than the specialized models but are three orders of magnitude larger. This results in an enormous gap in the size of chosable models. Such gaps don't only exist between models that are designed for different use cases—like general-purpose models versus task-specific ones—but are generally common due to the high cost of training, which discourages developers from producing models that only slightly vary from existing ones. Unfortunately, such disparities limit the UDBMS's ability to strike effective trade-offs: if a small model fails on challenging inputs, the next viable option may be significantly larger, forcing the system to incur substantial runtime overheads to handle the challenging inputs with higher accuracy.

To bridge such mismatches between operator needs and implementation offerings, we introduce Ken, a UDBMS *query execution engine* that exposes a fine-grained latency–accuracy trade-off curve to the query optimizer. Ken leverages *model cascades* [80], which directly address the above problem by aiming to use small models for easy inputs and only resorting to large models when necessary. Specifically, cascades first process inputs with a small model which produces both a prediction and a confidence score. If the confidence exceeds a predefined threshold, the prediction is accepted as final output; otherwise, the input is escalated to a more powerful model for reprocessing. Through their tunable certainty thresholds, cascades expose a high-resolution trade-off between accuracy

Proceedings of the VLDB Endowment, Vol. 19, No. 5 ISSN 2150-8097.
doi:10.14778/3796195.3796204

and computational cost, effectively creating many models at main points in the trade-off space. Furthermore, cascades can significantly save computation over using a single model at little accuracy degradation. The left side of Figure 1 compares different cascades of BERT classifiers to single BERT model instances. Each sample point represents the average number of FLOPs per sample and accuracy over a test set of 20,000 Sentiment-140 samples. Figure 1 shows that cascades can match the accuracy of the largest BERT model, while spending 2.5× fewer FLOPs. However, translating these savings in computation into lower latencies is not trivial.

While prior work has shown that cascades can improve the performance of UDBMSes, such as Lotus [59] for classification and NoScope [40] for video processing, they either only support models behind pay-per-request APIs (e.g., OpenAI), or only support small models and homogeneous workloads (scanning a video). Prior works do not leverage cascades in a more general setting, where workloads may be dynamic (e.g., have fluctuations in request arrival rates), and the UDBMS may query any model, including large ones or ones that are not hosted behind pay-per-request APIs. Such scenarios introduce several challenges which can even cause cascades to slow queries compared to always using a large model:

Cascades incur additional data transfer. Computing a model's output requires the GPU to transfer the model weights and inputs from memory to its arithmetic units. The execution engine can *batch* inputs together and compute their predictions inside the same forward pass, only requiring the model weights to be moved once for all predictions. For small batch sizes, the data movement takes longer than the arithmetic. As batch sizes grow, a forward pass involves more arithmetic but only incurs slightly more data transfer, eventually making the arithmetic the bottleneck.

When predicting a batch with a cascade, the weights of multiple models need to be transferred to the arithmetic units. As a consequence, cascades degrade performance for small batch sizes where memory transfers dominate the runtime. For large batch sizes, cascades can improve performance as they save arithmetic operations (e.g, FLOPs), as shown on the right of Figure 1, where obtaining predictions with an example cascade takes longer than directly using Llama-70B, even though the cascade saves FLOPs. However, forming large batches may require the system to queue samples — cascades therefore only improve performance when their computational savings make up for the queuing delay.

Cascades incur a higher memory footprint. Model cascades require multiple models to reside in GPU memory. Often, their combined weights cannot fit onto a single GPU. As a result, individual models cannot be replicated as extensively as if they were served alone, reducing their achievable throughput. For generative LLMs, throughput further depends on the amount of *free* GPU memory, which is reduced if memory needs to be reserved for additional model weights (§2).

In conclusion, cascades only reduce latencies in certain scenarios. Ken introduces three mechanisms optimize the use of cascades and account for the above limitations:

Adaptive cascade selection. User-facing systems, such as UDBMSes, typically face fluctuations in query arrival rates [46, 56, 90]

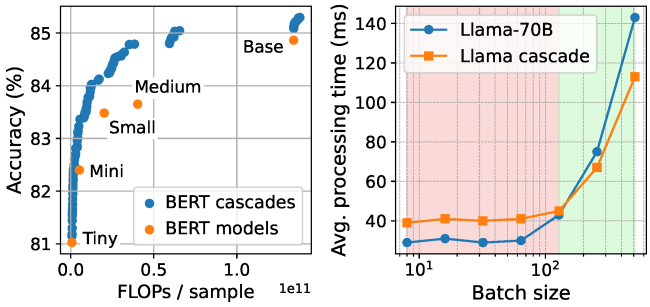


Figure 1: Left: Average FLOPs per sample versus accuracy of BERT classifiers on Sentiment-140 [30]. Right: Runtimes of Llama-70B and a cascade incurring 84% of its FLOPs.

(e.g., users upload PDF invoices during work hours). Different cascades suit different arrival rates and Ken dynamically adapts which cascade to use. High arrival rates allow for large batch sizes, and Ken therefore favors FLOP-efficient cascades. For low arrival rates, Ken prioritizes cascades with high accuracy (e.g., just using the most expensive model, a one-model cascade).

Joint model placement. Loading models onto GPUs takes seconds to minutes, so Ken preloads all models instead of loading them at runtime. Cascade selection determines which models to preload and how intensively each model is used for a given query arrival rate. Given the load on the models, Ken optimizes how models are replicated and parallelized across the GPU cluster.

Multi-model scheduling. Different models might be placed onto the same GPU, which requires Ken to multiplex GPU time between them and optimize queueing, batching, and load balancing.

While each mechanism is an optimization problem on its own, their interdependence requires them to be co-optimized. For example, the chosen cascades must allow for a GPU placement that meets their throughput requirements. Similarly, the model placements must allow for good load balancing. However, load balancing is defined with respect to a placement, and a placement is defined with respect to the cascade choices. To formalize the optimization and capture its results, Ken parametrizes each mechanism and packages the parameter configurations of all mechanisms into a *gear plan*.

Gear plans represent latency-accuracy trade-off points for an operator, and the UDBMS's query optimizer hence chooses a gear plan for each operator. While computing the Pareto frontier of gear plans is NP hard, Ken introduces a search algorithm that we find to be effective in practice. Once the gear plan search is complete and the UDBMS has selected a query plan, computing model predictions and operating a gear plan online incurs negligible overheads.

We find that Ken achieves 1.7 × −3.3× latency reductions over state-of-the-art baselines when combining models *from the same family* for a single logical operator. Furthermore, Ken enables the query optimizer to effectively utilize models that differ by orders of magnitude in size, achieving 122× lower latencies than existing baselines at comparable accuracy.

In summary, our contributions are as follows:

• We propose Ken, a UDBMS execution engine that uses model cascades to enable fine-grained latency–accuracy trade-offs. This allows UDMBSes to avoid excessive latency or accuracy degradations

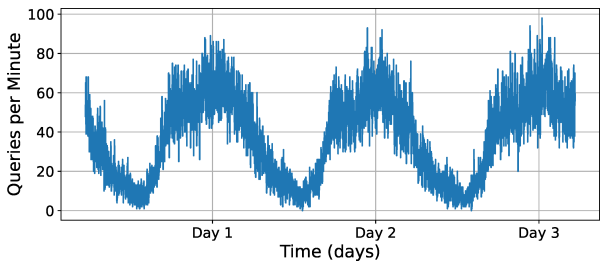


Figure 2: Excerpt of a workload trace [67] that is representative for model serving.

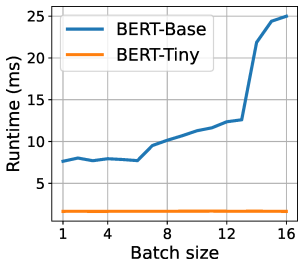


Figure 3: Runtimes of a forward pass with different batch sizes for BERT classifiers.

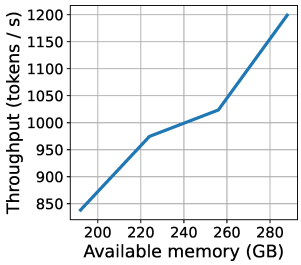


Figure 4: Request throughput of Llama-3.1 70B [26] as a function of available memory.

that arise when only faced with coarse-grained trade-off points where none meet the precise application needs.

- Achieving low latency response times with cascades comes with caveats, which Ken addresses by optimizing query-aware cascade selection, GPU placement, and invocation scheduling.
- We evaluate Ken and find that it exposes trade-offs to the UDBMS query optimizer that outperform strong baselines by $1.7\times - 3.3\times$ when combing similar models. When combining models with large size gaps, Ken achieves up to 122× latency improvements over baselines for multi-operator UDBMS queries.

2 MODEL CASCADES: PROMISES AND PITFALLS

In this section, we overview model cascades, describing their potential to enhance UDBMSes but also the challenges to build a query execution engine around them.

Background on model cascades. A model cascade [80] consists of a sequence of models $\{m_1, \ldots, m_k\}$, each of which consumes the same inputs and produces similar outputs. Generally, m_i is smaller (in terms of parameters and inference time) than m_{i+1}. Given this sequence of models, a set of thresholds $\{\tau_1, \ldots, \tau_{k-1}\}$ dictate how samples are conditionally forwarded for inference. Specifically, for a given sample x, m_1 first produces a prediction $f_1(x)$ and a *prediction certainty* $\delta_1(x)$. If $\delta_1(x)$ exceeds τ_1, $f_1(x)$ is used as the final output. Otherwise, sample x is *cascaded* to m_2, where the inference process is repeated recursively. Many existing approaches (e.g., the entropy of the output logits) can quantify prediction certainty without significant inference overhead [40].

Promising properties of model cascades. Model cascades have two properties that are promising for query execution engines.

First, cascades can significantly reduce the incurred FLOPs while largely maintaining accuracy over a prediction task. For example, Figure 1 (left) shows the tradeoff between FLOPs and accuracy for different model cascades in comparison to the stand-alone models that are used to build the cascades. The models are fine-tuned BERT classifiers and are running on a test set of 20,000 samples from the Sentiment-140 classification benchmark [30]. In this example, a model cascade can match the accuracy of BERT-Base while only using 45% of its FLOPs per prediction. We have found that the Pareto-optimal cascades in terms of FLOP savings typically use 1-3 cascades — more models merely add FLOPs in the worst case (when a sample needs to be cascaded until the largest model), while not showing benefits by having a more fine-grained model short-circuiting mechanism. Prior work has shown that cascades can reduce the average FLOP count on a variety of ML tasks [16, 24, 33, 40, 60, 80, 87, 92, 97]. While some cascades in Figure 1 even outperform BERT-Base in accuracy, this is not typical (although similar gains have been reported in prior work [79]). Conversely, for workloads where queries have largely uniform difficulty, multi-model cascades do not achieve meaningful FLOP reductions and can be unnecessarily complicated.

Second, model cascades offer a high-resolution trade-off between FLOPs and accuracy. Instead of choosing between individual models, cascades allow to choose points on the FLOP-accuracy trade-off curve at a finer granularity. This makes them well-suited for workload adaptation. In this work, we model the query load to a UDBMS after production request traces to Azure serverless functions [67], which are commonly used to model the query loads issued to user-facing applications such as UDBMSes [49, 63, 95]. Figure 2 shows an excerpt of this production workload trace. The workload shows large short-term variations in query load which are typical for user-facing applications [46, 56, 90]. Cascades are well-suited to adapt to such changes because they (i) offer a trade-off point that precisely fits the momentary query load and (ii) transitioning between cascades can be as simple as varying the certainty threshold τ without changing the models.

In summary, model cascades offer great potential to reduce the computational cost of ML inference and adapt to dynamic and variable workloads. However, in real-world applications, users are more concerned with latency and throughput than pure computational cost (i.e., FLOPs). Thus, to leverage the promising properties of cascades, the query execution engine must translate savings in computation cost to lower latencies and higher throughput. In the following, we discuss the challenges of optimizing these metrics.

Batching effects. Model inference systems typically propagate several samples through a neural network as one *batch*. The runtime for processing a batch does not increase proportionally with the batch size, especially for small batch sizes. For example, Figure 3 shows the time taken to propagate one batch with two BERT classifiers on a V100 GPU. The runtime for BERT-Tiny essentially stays constant for all batch sizes up to 512. The reason in this particular case is that BERT Tiny's per-batch computational load is so low on Sentiment-140,that system-level overheads (e.g., tokenization, moving tokenized samples from CPU to GPU, kernel launches and synchronization) hide GPU scaling effects. Propagating 16 samples

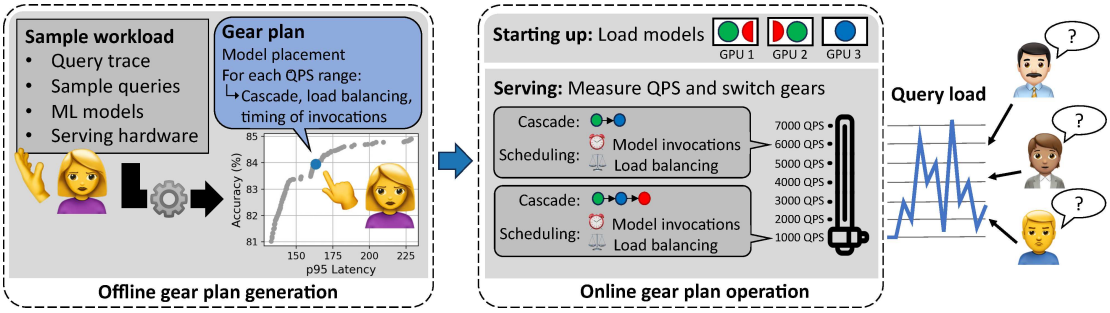


Figure 5: Workflow of Ken.

through the BERT-Base only triples the runtime when compared to propagating one sample through it.

Thus, while cascades save computation, batching effects can actually cause them to lower throughput. For example, consider a cascade that first propagates four samples through a BERT-Tiny classifier and cascades just one sample to a BERT-Base classifier. It takes $2ms + 8ms$ to process the four samples, resulting in a throughput of 400 samples per second. However, directly propagating the four samples through BERT-Base only takes $8ms$, resulting in a throughput of 500 samples per second. The cascade therefore decreases the throughput by 20% despite incurring 4× less computation. As we describe in Section 4, Ken directly optimizes such effects.

Memory constraints. With the rise of larger models, GPU memory constraints become a key consideration for inference systems. For example, the weights of a Llama 70B model occupy 140GB of memory, requiring the model to be split across several GPUs. Furthermore, in the case of auto-regressive LLMs, the amount of available memory directly affects the model's throughput: LLMs produce an intermediate state for each request called the *Key-Value (KV) cache*. The number of KV caches that can be held in memory determines how many requests the LLM can process in parallel, therefore directly affecting the LLM's throughput. Figure 4 shows the throughput of Llama-3.1 70B as a function of the memory available to store KV caches.

Because model cascades require loading multiple models into GPU memory, some configurations may exceed available capacity. For LLMs, even when additional models fit, loading their weights further reduces the GPU memory available for KV caches, restricting the LLM to smaller batch sizes and potentially lowering throughput. Ken mitigates this by selecting cascades that account for both memory limits and their effect on LLM throughput.

3 SYSTEM OVERVIEW

Ken targets the common scenario where models are served on a provisioned set of GPUs which is exclusively managed by Ken. Ken's workflow mirrors the current workflow of UDBMSes and therefore allows for seamless integration. Specifically, applications that run on UDBMSes often feature similar types of repeated queries (e.g., populate fields in a structured database whenever a user uploads a PDF invoice). A UDBMS will optimize such an application by first asking the user for a workload sample, configuring each operator in the query (i.e., running the query optimizer), and finally processing requests "online" according to the query plan.

System design. The inputs to Ken are a set of models that are relevant for executing the operator. These models must have the same input and output formats — for example, they may belong to the same *model family* (e.g., Llama models, BERT models, etc), and the hardware that the UDBMS will use to run these models. Ken targets a set up where a fixed number of hardware resources (e.g., a provisioned cluster) are allocated to it.

Ken then follows the typical workflow of a UDBMS. Ken first searches for *gear plans* where a gear plan contains (i) an assignment of cascades (including 1-model cascades) to query arrival rates, (ii) a placement of models onto GPUs, and (iii) a scheduling policy on how to route models to GPUs and when to trigger model invocations. Ken exposes these gear plans to the query optimizer, which selects one. This search happens "offline" and leverages the sample workload that is given to the UDBMS. Once the query optimizer has chosen a gear plan and the UDBMS enters the online phase, Ken starts serving requests according to chosen gear plan.

Ken leverages the UDBMS workflow in several ways: first, by having access to a sample workload, Ken searches for optimized gear plans by evaluating them against it, effectively tailoring its execution to the workload. Second, Ken uses the offline phase to offload all of its decision making to minimize online overheads. Specifically, Ken optimizes query execution by jointly deciding on: (i) which cascade to serve under varying query loads; (ii) how to replicate cascade models (i.e., serve the same model on different GPUs), and parallelize them across GPUs; and (iii) how to schedule requests (i.e., balance load across model replicas and time model invocations to maximize batching). While a model placement (ii), and a scheduling policy (iii) are fundamentally needed to run inference, adapting the cascades to variations in query load (i) is an optimization introduced by Ken.

The offline phase of UDBMSes allows Ken to use slower, but more accurate, algorithms to search this decision space. Ken parametrizes these decisions and stores them inside a gear plan, which precisely describes how to serve the workload online and allows Ken to apply these optimizations with negligible overhead. Finally, by knowing the choice of gear plan before online serving starts, Ken can pre-load all models onto the GPUs and avoid the overheads of loading them dynamically in response to workload changes. These overheads would quickly become impractical as it often takes seconds to minutes to load models but typical inference serving workloads can fluctuate from second to second (Figure 2).

Optimization approach. Before searching for a gear plan, Ken performs a one-off, up-front search for Pareto-optimal cascades

(FLOPs vs. accuracy) using the sampling method from [82], which effectively solves the problem.

Given these cascades, finding optimal gear plans is NP-hard[1]. To approximate the latency–accuracy Pareto frontier, Ken begins with two anchor points: the highest-accuracy gear plan (i.e., always selecting the most accurate cascade or model) and the lowest-latency gear plan (i.e., always choosing the cheapest model). It then iteratively generates gear plans that approximate the frontier between these points, greedily trading off minimal accuracy for maximum latency reduction.

Each gear plan is built via a cost-based search that leverages the problem's hierarchical structure. Specifically, model placement P_A can only be optimized once a cascade assignment A is fixed (i.e., it's known which models are used and what throughput they require). Similarly, a scheduling policy $S_{P,A}$ can only be determined given both the cascade assignment and the model placement (i.e., how many requests go to a model and which GPUs host it). Although these decisions follow a fixed pipeline, the performance of the earlier stages affects the outcome of the later ones.

Figure 6 illustrates this with an example where three static cascades process a workload on two V100 GPUs. We measure cascade latency under different model placements and scheduling policies, showing that cascade design must jointly consider placement and scheduling to assess performance. Cascade 1 uses BERT Mini and forwards 60% of samples to BERT Medium. Cascade 2 uses BERT Mini, forwards 30% of samples to BERT Medium, and 20% of those to BERT Base. Cascade 3 uses BERT Mini and forwards 9% of samples directly to BERT Base. In the example, we use two V100 GPUs and under Placement 1, all three models are replicated on both GPUs, with samples split evenly. At high QPS, BERT Base fails to meet throughput targets for Cascades 2 and 3, making Cascade 1 the lowest-latency option. In Placement 2, all models run on GPU 1, while GPU 2 hosts only BERT Medium and Base. Load is balanced across GPUs for all QPS levels. This substantially boosts BERT Base throughput, greatly benefiting Cascades 2 and 3 but only marginally helping Cascade 1. BERT Medium becomes a relatively significant latency source and slightly lags behind when processing queries at high QPS. Finally, Schedule 3 increases BERT Base batch sizes at higher QPS, enabling it to fully meet throughput targets. BERT Medium remains the latency bottleneck, making Cascade 1 the slowest and Cascade 2 the second slowest.

Ken uses recursive search to jointly optimize all subproblems. For each, it introduces a dedicated module that generates and scores multiple candidate solutions using simulated latency and accuracy as cost-based guidance. Specifically, the workload adapter (§4.3.1) proposes cascade assignments; for each candidate assignment A, the placement optimizer (§4.3.2) explores multiple model placements P_A; and for each P_A, the scheduling optimizer (§4.3.2) explores scheduling policies $S_{P,A}$. Each complete candidate gear plan—i.e., a combination of A, P_A, and $S_{P,A}$—is simulated on the user's sample workload to estimate accuracy and latency. Scoring is done bottom-up: Scheduling candidates $S_{P,A}$ are scored by calling into the simulator directly. After converging on an optimized scheduling policy $S^*_{P,A}$, the candidate placement P_A is scored considering the latency achieved by $S_{P,A}$. Finally, after converging to an optimized

[1]E.g., Multiple Knapsack Problem reduces to GPU placement

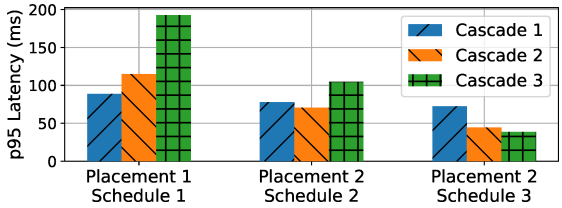


Figure 6: p95 latencies of BERT cascades when changing model placement and the timing of model invocations.

placement P^*_A, the cascade assignment A is scored by the accuracy and latency achieved by P^*_A. As shown in Section 5.6, this approach is efficient because each submodule exploits structural properties of its subproblem to quickly narrow down high-quality candidates.

When approximating the gear plan Pareto frontier for an operator embedded in a pipeline with further operators, Ken's cost-based search naturally applies. It incorporates not only the models and load defined by the current gear plan's cascade assignment, but also the ones of cascade assignments of other gear plans—i.e., the models they use, and the loads that their models face—integrating all into the simulation. To optimize the current gear plan, the three submodules then proceed as usual, but their cost evaluations now account for the interactions with these fixed gear plans.

Likewise, Ken's cost-based approach generalizes across model types, including auto-regressive LLMs and models that require only a single forward pass. Combined with the broad applicability of cascades to ML tasks, Ken provides a general framework capable of supporting a wide range of models.

System workflow. Figure 5 shows the workflow of Ken. First, the user registers a sample workload with Ken, containing (i) a sample query arrival trace, such as the one depicted in Figure 2, (ii) some labeled queries that are similar to the queries to be served online, (iii) access to the provisioned hardware for serving the requests, and (iv) a set of models that Ken may choose to construct cascades (e.g., BERT-Tiny, BERT-Small, BERT-Base). Given these inputs, Ken generates several gear plans that approximate the latency-accuracy Pareto frontier on the user's workload in the *offline phase*.

After generating this approximate Pareto frontier, the user picks the gear plan that best matches their application requirements (e.g., service-level objectives). The result of the offline phase is a selected set of gear plans. As shown in Figure 5, the gear plans consist of a cascade for different ranges of input queries per second (QPS). Each cascade has an accompanying assignment of models to hardware and load balancing schedule. This gear plan is fed to the online phase for inference serving. Prior to the online phase all models in the gear plan are loaded into GPU memory.

Pivotal workload changes. Prior work has found that model serving workloads have high repeatability from the perspective of the serving system [49, 95]. Furthermore, users typically know the size and domain of their business operations. Nevertheless, workloads may pivot or some users might provide the UDBMS with an unrepresentative sample workload in the offline phase. While Ken functions under such circumstances, its gear plan will be optimized for a different workload.

To deal with such scenarios, Ken allows to monitor system performance online and swap the gear plan out if deemed necessary.

Section 5.6 shows that the offline phase runs within a couple of minutes, which makes re-running the offline phase very practical. Since the offline phase runs on the CPU and is significantly faster than real-time, the UDBMS query optimizer might even decide to run the offline phase over a moving window of the workload.

4 CONSTRUCTING THE GEAR PLAN FRONTIER

This section describes how Ken generates gear plans that approximate the latency–accuracy Pareto frontier on a user-provided sample workload. We begin with the case of a single operator in isolation. Section 4.1 defines the optimization problem. Section 4.2 outlines the iterative frontier search, where each iteration produces one gear plan. Section 4.3 details how a gear plan is constructed.

4.1 Optimization Problem

Gear plan parametrization. Gear plans encode three decisions: (i) which cascade to use at each request arrival rate, (ii) how to replicate and parallelize the involved models, and (iii) how to balance load across GPUs and schedule model invocations. While a model placement (ii), and a scheduling policy (iii) are fundamentally needed to run inference, adapting the cascades to variations in query load (i) is an optimization introduced by Ken. For single-forward-pass models, each request comprises of one forward pass and QPS is therefore measured simply by the number of arriving queries. However, for auto-regressive models, forward passes are requested by new user queries but also by running, auto-regressive queries. In such cases, Ken measures system load (i.e., QPS) as the number of requested *forward passes*, i.e., the number of *running* queries and queued queries at the time. We now describe the parametrization of a gear plan.

(i) Ken must handle a wide range of query arrival rates, measured in queries per second (QPS). Given a minimum rate (e.g., 0) and maximum rate QPS_{max}, Ken uniformly partitions the range into $|\mathcal{Q}|$ QPS intervals. Each interval $q_i \in \mathcal{Q}$ is assigned a cascade, parameterized by its cascade ID.

(ii) Let $\mathcal{M}$ be the set of models used in the selected cascades. A model placement $\mathcal{P}$ specifies a set of replicas for each model. Each replica is assigned to one or more GPUs (indicating parallelization). For example: $\mathcal{P} = \{\mathcal{R}[m_1], \mathcal{R}[m_2], \ldots\}$, where $\mathcal{R}[m] = \{\langle gpu_1, gpu_2, \ldots\rangle, \ldots\}$. $\mathcal{R}[m]$ is a set because one model may have several replicas and each element in the set is one replica.

(iii) The scheduling policy defines, for each QPS range $q_i \in \mathcal{Q}$: (a) q_i^r, the QPS routed to replica r under rate q_i, and (b) bs_i^r, the queue length required before invoking replica r under load q_i,

Optimization problem. The overall optimization problem of the offline phase is to approximate the latency-accuracy Pareto frontier of gear plans. This involves finding optimized parameter combinations that allow gear plans to achieve high performance while also, collectively, spanning the latency-accuracy trade-off curve at a high resolution. Ken targets a set up where a fixed number of hardware resources (e.g., a provisioned cluster) is allocated to it. This is a common scenario for running ML inference and suits Ken as overheads from loading additional model weights are incurred once offline, and not at runtime, e.g., as additional nodes are spun up. In Supplementary Material B we show how Ken saves cost in such a set up, as it allows the user to reach trade-off points in the latency-accuracy trade-off space with fewer GPUs than baselines.

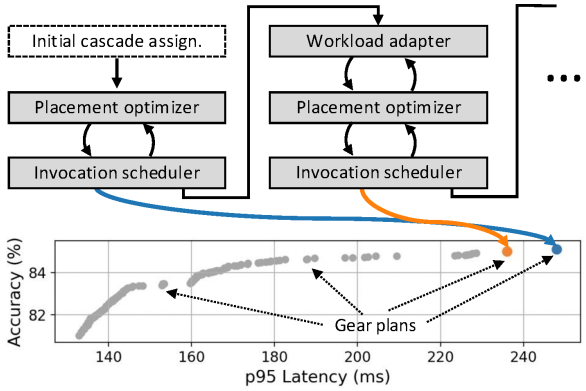


Figure 7: Ken iteratively generates gear plans. Ken starts from the gear plan that achieves the highest accuracy and iteratively trades accuracy off for lower latency.

The main challenge in optimizing a gear plan is that it involves three sub-problems — cascade assignment, model placement, and invocation scheduling — that need to be optimized jointly, resulting in an immense exponential search space and an NP-hard problem. For instance, optimal model placement depends on the cascade's QPS allocation: a frequently-invoked, expensive model at high QPS ranges must be replicated across GPUs to meet throughput. Conversely, optimal cascade assignment depends on both placement and scheduling: Figure 6 illustrates this on an example of three BERT-Cascades whose latency is measured on a sample workload of Sentiment-140 queries. For the initial placement and scheduling, Cascade 1 incurs the lowest p95 latency. However, after changing both placement and scheduling policy, Cascade 1 actually incurs the largest latency of the three. As a result, cascades can only be chosen and assigned with respect to a placement and scheduling policy. The interdependence of the sub-components of a gear plan call for an approach that optimizes them jointly.

4.2 Iterative Generation Process

The offline phase of Ken *iteratively* generates a set of gear plans that approximates the latency-accuracy Pareto frontier on the user-provided workload.

In each iteration, Ken generates a gear plan by selecting a cascade assignment, model placement, and scheduling policy. It begins with a Pareto-optimal assignment that maps the most accurate cascade (e.g., most expensive model) to each QPS range, then optimizes placement and scheduling accordingly. In subsequent iterations, Ken proposes modifications to this assignment by replacing some cascades with cheaper alternatives, aiming to gradually trade-off accuracy for lower latency.

Because changes to the cascade assignment can impact the feasibility of placement and scheduling, all three components must be co-optimized (§ 4.1). Ken uses dedicated modules for each sub-problem and greedily searches for cascade modifications that reduce latency with minimal accuracy loss. Within each iteration that produces a new gear plan, the workload adapter performs a cost-based search over candidate cascade assignments. For each candidate,

it re-optimizes placement and scheduling, simulates the resulting gear plan on a sample workload, and uses the observed latency and accuracy to guide further refinements. This process continues until the adapter converges on a candidate, which is then selected as that iteration's final gear plan. It will then move on to the next iteration, which generates the next gear plan by modifying the assignment of the current one. This process is repeated until Ken reaches the assignment that assigns the cheapest model to each QPS range.

Figure 7 illustrates this process: it begins with the initial cascade assignment, optimizes placement and scheduling with respect to it, and then enters an iterative loop where each round produces a new gear plan by co-optimizing cascade assignment, model placement, and scheduling policy.

Section 4.3.1 provides a detailed description of the full gear plan generation process within one iteration, including the optimization modules and the simulation procedure.

4.3 Workflow of a Single Iteration

We now describe how a gear plan is generated within one iteration of the Pareto frontier search (§ 4.2). The generation process involves three optimization modules: the *workload adapter*, *placement optimizer*, and *invocation scheduler*.

4.3.1 Workload adapter. Ken uniformly partitions the full range of QPS into *QPS ranges* Q. The workload adapter assigns a cascade $c_i \in C$ to each $q_i \in Q$, forming a mapping from QPS ranges to cascades. The set of candidate cascades C is precomputed in a one-time step using the method described in Section 3 and [82].

Exhaustively searching the $|C|^{|Q|}$ possible assignments is intractable. Instead, the workload adapter picks out one of the QPS ranges $q_i \in Q$ and explores a new cascade assignment by replacing the cascade c_i of q_i with the next cheaper one in C — i.e., a new cascade $c_i' \in C$ with lower average compute cost but potentially reduced accuracy. This change is then propagated to all higher-QPS ranges $q_j > q_i$ such that, if the current cascade of q_j is more expensive than c_i', it will be replaced by c_i'. This enforces an invariant where a higher-load regime is never assigned a more expensive cascade than any lower-load one. This heuristic reflects the need for lightweight cascades under high QPS to maintain throughput.

Note that, given the previous plan's cascade assignment and the set of cascades C, the choice of QPS range q_i uniquely defines the new cascade assignment (i.e., global mapping of QPS ranges to cascades). After picking a candidate range q_i, the model placement (§ 4.3.2) and scheduling policy (§ 4.3.3) are optimized with respect to the new cascade assignment, rendering a gear plan gp. This gear plan gp is then simulated on a user-provided sample workload to obtain its latency $lat(gp)$ and accuracy $acc(gp)$. The workload adapter then computes the plan's score as $score(g) = {}^{acc(gp)}/_{lat(gp)}$.

The adapter evaluates all $|Q|$ choices of q_i and selects the one yielding the highest-scoring gear plan g^*. It outputs g^* as the final gear plan for the current generation iteration.

The module runs in $O(|Q|)$ time. Over the course of generating the full Pareto frontier, it is invoked $O(|C| \cdot |Q|)$ times. This is practical given typical values (e.g., $|C| = 100$, $|Q| = 50$). In practice, the number of module invocations is also much lower than $|C| \cdot |Q|$.

4.3.2 Placement Optimizer. Given a cascade assignment, the goal of the *placement optimizer* is to find a model placement that minimizes latency when serving the assignment. Let $\mathcal{M} = m_1, \ldots, m_k$ be the set of models in the cascade. A placement $\mathcal{P} = p(m_1), \ldots, p(m_k)$ specifies, for each model m_i, a set of replicas $r(m_i)$. A replica r comprises of a set of GPUs that the model is placed on — the set can contain several GPUs if the model is parallelized across GPUs. For example, $p_1 = \langle g_1 \rangle, \langle g_2, g_3 \rangle$ denotes two replicas of m_1: one placed on GPU g_1, the other parallelized across GPUs g_2 and g_3 (where g_1, g_2, and g_3 are GPU identifiers).

Currently, Ken's online serving system only supports tensor parallelism [57] within a single node, and restricts each replica to span 1, 2, 4, or 8 GPUs. However, these are not fundamental limitations — Ken's optimizer can support further parallelization techniques, combinations thereof, and multi-node placements, which becomes evident from the algorithm below. Under the current constraints, each model m_i can be parallelized in $\sum_{k \in 1,2,4,8} \binom{8}{k} = 107$ ways on a node with 8 GPUs. When adding support for additional parallelization techniques, these would simply be added as additional points to this set of choosable parallelization configurations.

Ken uses beam search to find a low-latency placement $\mathcal{P}$ for the user-provided sample workload. Each beam candidate is a placement, and placements are expanded by adding replicas. The search begins by sampling a set of initial placements, each containing exactly one replica per model $m_i \in \mathcal{M}$.

When expanding a placement $\mathcal{P}$, new replicas are sampled uniformly. Samples violating constraints—e.g., memory limits or duplicating model components on the same GPU—are discarded. For each of the bw beam candidates, k new placements $\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(k)}$ are generated by adding sampled replicas. The resulting $(k+1) \cdot bw$ candidates are evaluated using the scheduling optimizer (§ 4.3.3) to simulate latency, and the top bw placements are retained.

Since placement only affects latency, accuracy is not considered in the search. The beam search terminates when no improvement is observed across the beam.

4.3.3 Invocation scheduler. Given a cascade assignment and a model placement $\mathcal{P}$, the *invocation scheduler* determines (i) how load should be balanced across GPUs, and (ii) when to invoke a model replica. A *scheduling policy* is specific to a QPS range q_i. The invocation scheduler is invoked on all QPS ranges $q_i \in Q$, and we describe the optimization for one such q_i below.

Load balancing. The *invocation scheduler* first optimizes how load is distributed across GPUs by determining how many queries q_i^r should be processed per second by each replica $r \in \mathcal{P}$.

The variables q_i^r are optimized via the linear program in Equation 1. To formulate this as a linear program, we assume runtime scales linearly with batch size—this simplification is used only for load balancing; batching effects are accounted for later in invocation scheduling. Let $time(r)$ denote the runtime of replica r for a batch size of 1. Then, $time(r) \cdot q_i^r$ represents the time taken by r to process q_i^r queries.

To express the target load per GPU, we define *avg_load* as the total work if uniformly distributed: each replica of model m processes $q_i^r = {}^{q_i^m}/_{|\mathcal{R}[m]|}$ queries, where q_i^m is the QPS assigned to model m and $\mathcal{R}[m]$ denotes its replicas.

Equation 1 minimizes the deviation of each GPU's load from the average, where load is measured as total processing time per second. The constraint in Equation 2 ensures that each model's total query volume is fully served: the combined throughput of its replicas must equal the incoming rate q_i^m.

$$\text{minimize} \quad \sum_{g \in \mathcal{G}} \left| \sum_{r \in \mathcal{R}[g]} time(r) * q_i^r - avg\_load \right| \tag{1}$$

$$\text{subject to} \quad \sum_{r \in \mathcal{R}[m]} q_i^r = q^m \qquad \forall m \in M,\ q_i^r \geq 0 \tag{2}$$

Note that q_i^m is known from the average forwarding rate of the cascade on the user's sample workload.

Invocation timing. Given how to balance the load, Ken will now determine the optimal timing of model invocations. Recall from Section 2 that propagating several samples through a model as one *batch* can significantly increase throughput. Therefore, the online phase of Ken maintains a queue of queries for each model replica r, and r will not be invoked until the queue holds bs_i^r many queries. A small batch size bs_i^r will incur frequent model invocations and a large bs_i^r increases waiting times for the queue samples. Both scenarios could increase the query latency.

To optimize parameters bs_i^r for each replica $r \in \mathcal{P}$, Ken uses up-front profiles of each replica r when run with different batch sizes. Such profiling are feasible since the number of possible configurations for a replica is limited.

Initially, Ken sets $bs_i^r = 1$ for all replicas, avoiding unnecessary queuing overhead in case that the QPS to r is sufficiently low. Then, if the total compute time across all replicas on a GPU g exceeds $1s$ — implying g cannot sustain its assigned QPS — Ken incrementally increases bs_i^r for replicas on g in a round-robin manner until the total runtime falls below $1s$. This heuristic ensures all replicas scale batch size proportionally until the assigned load can be sustained.

Note that the load balancer does not guarantee that a GPU can handle its load in one second; it only balances QPS across GPUs. If all bs_i^r reach their limits and runtime still exceeds $1s$, Ken deems the gear plan *infeasible*, prompting the placement optimizer and workload adapter to revise their decisions.

Otherwise, if a valid set of bs_i^r is found, Ken completes the gear plan gp and simulates it (§ 4.3.4) to obtain latency $lat(gp)$ and accuracy $acc(gp)$ on the input workload. These metrics serve as feedback in the cost-based searches of the *placement optimizer* and *workload adapter* modules.

4.3.4 Simulator. Ken uses a continuous-time, discrete-event simulator [61] to efficiently explore candidate gear plans in the offline phase. We evaluate the simulator in supplementary material A.

Before simulation, an upfront profiling step gathers: (i) model predictions and certainties for each $m \in \mathcal{M}$ on the user-provided sample queries, and (ii) runtimes for different models and parallelization strategies across a few batch sizes. Both profiling steps are fast and embarrassingly parallel.

Given this data, runtime is largely deterministic for networks not split across nodes—forward passes with the same batch size follow a fixed sequence of operations [49]. The simulator then replays the workload, recording per-sample predictions and latencies to compute the gear plan's aggregate metrics: $lat(gp)$ and $acc(gp)$ for plan gp.

5 EVALUATION

Given a logical operator, Ken aims to provide the UDBMS query optimizer with Pareto-efficient accuracy-latency trade-off points to chose from. We now evaluate the trade-offs that Ken achieves for different operators and aim to answer the following questions.

- §5.3 How do Ken's trade-off curves compare to alternatives?
- §5.4 Can a UDBMS query optimizer leverage Ken's operator trade-offs to assemble efficient end-to-end pipelines?
- §5.5 How optimal are the gear plans found by Ken's optimizer?
- §5.6 How optimal are the gear plans found by Ken's optimizer?
- §5.7 How do different optimizations affect performance?

5.1 Workloads

Each workload in our evaluation consists of (i) the set of models that may be used to implement a UDBMS operator, (ii) an ML benchmark to assess the prediction quality for different operator tasks (e.g., sentiment classification), and (iii) a trace to emulate query timing, i.e., when user queries trigger the execution of an operator.

Sentiment-140 with BERT. Sentiment-140 [30] is a popular sentiment classification benchmark. We use a family of fine-tuned BERT classifiers [25, 78] to build cascades. Training and serving samples are mutually exclusive. Ken is implemented atop the Huggingface Transformers inference engine [84] and runs on V100 GPUs. The request trace is derived from Tweet timestamps.

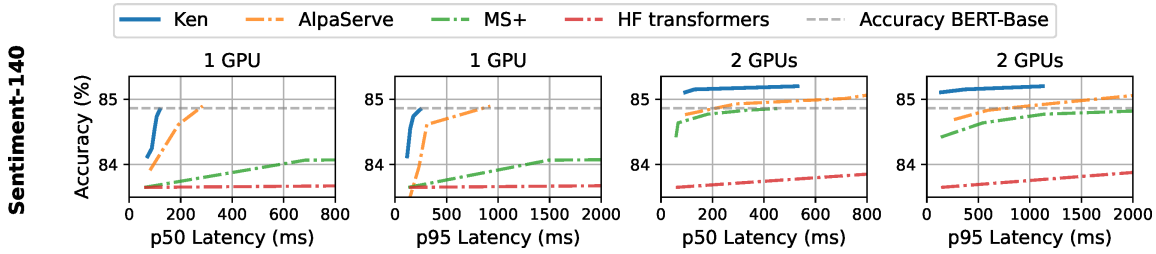
HellaSwag with Llama-2. HellaSwag [93] is a popular commonsense reasoning benchmark. We build cascades from the Llama-2 model family [27] and OpenLlama-3B [29], quantized to int4 with GPTQ [28]. Models are served on V100 GPUs. Ken is implemented atop the ExLlama inference engine [77]. The request trace is derived from an Azure Functions invocation log [67], representative of real-world model serving [63, 95].

MT-Bench with Llama-3. MT-Bench [99] is a multi-turn conversation benchmark judged by another LLM ("LLM-as-a-judge"). We follow this benchmark protocol and use GPT-4o [74] as the judge. Cascades are built from FP8-quantized Llama-3.1 and Llama-3.2 models [26]. Models run on A100 GPUs (80GB) using the vLLM inference engine [45]. The request trace matches the Azure Functions trace [67] used in the HellaSwag workload.

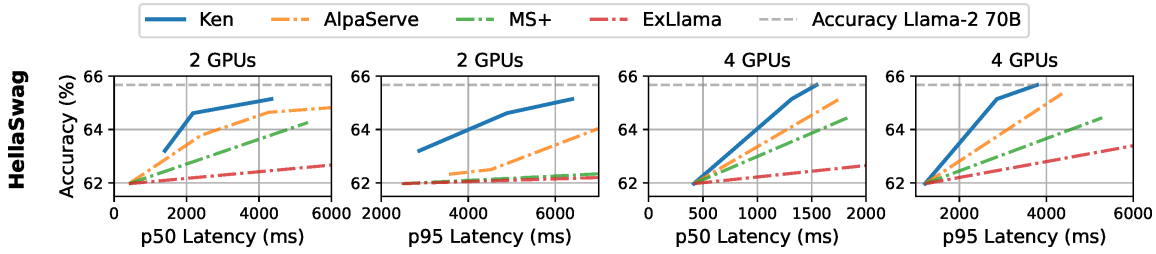
5.2 Baselines

UDBMSes typically rely on generic *model serving systems* as their execution engine. For this reason we compare against state-of-the-art model serving systems and popular production systems. For each baseline, we construct a Pareto frontier by running it with different models and/or system-specific configurations.

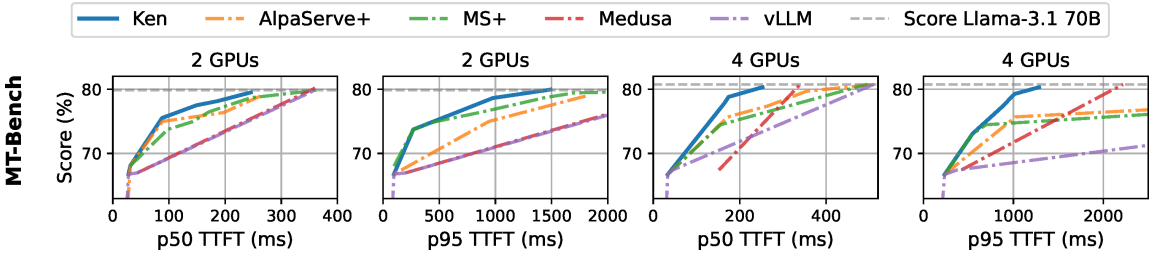
Stand-alone engines. We compare Ken to stand-alone models served directly via popular inference engines (vLLM [45], Huggingface Transformers [84], ExLlama [77]), the most common serving approach today.



(a) Latency-score trade-off of different baselines on the Sentiment-140 workload.



(b) Latency-accuracy trade-off of different baselines on the HellaSwag workload.



(c) Latency-accuracy trade-off of different baselines on the MT-Bench workload.

Figure 8: The trade-off between latency and quality of Ken and different baselines.

Cascades on top of AlpaServe. We exhaustively search all cascades to find the best one for the exact workload used in the experiment, then serve it using AlpaServe [49] with access to the actual workload trace. This baseline is heavily overfitted and has access to the ground-truth optimal cascade. For MT-Bench, we extend AlpaServe to support autoregressive LLMs.

ModelSwitching+. We enhance Model Switching (MS) [96] — which assumes all models fit in memory — by integrating Ken's placement optimizer, and workload adapter. The resulting version, MS+, therefore adopts a lot of Ken's optimizations but switches between individual models (not cascades).

Cocktail+. In supplementary material C, we compare Ken to Cocktail [32], which combines auto-scaling with model ensembles. Since Cocktail scales the number of servers, we exclude it from fixed-GPU experiments. We enhance it by removing instance startup delays and providing ground-truth workload forecasts. We refer to this version as Cocktail+.

Medusa. For MT-Bench, we include Medusa [15], a recent technique for efficient decoding in LLMs. We use fine-tuned accelerators for Llama-3 [26] and use vLLM for running them.

5.3 Trade-Off Efficiency

We first compare Ken to the baselines from Section 5.2, evaluating latency and output quality on the workloads described in Section 5.1. We choose set ups with limited GPU resources as Ken is especially stressed when provisioned with little GPU memory. While Ken produces a fine-grained latency–accuracy Pareto frontier, we only evaluate a subset of gear plans due to the runtime of full evaluations.

Figure 8 shows the latency–accuracy trade-offs for Ken and all baselines across varying GPU counts. For MT-Bench, we report time-to-first-token (TTFT). TTFT mainly captures the queuing delay of a request whereas the alternative latency measure, Total-Generation-Time, mainly captures how many tokens the response contains (i.e., is query-specific).

Across workloads, Ken consistently outperforms all baselines by adaptively using model cascades. On classification tasks, it improves p95 latency at the same accuracy by up to 3.3× on Sentiment-140 and 2.1× on HellaSwag over the strong AlpaServe+ baseline. On MT-Bench, Ken improves p95 latency over Medusa by up to 1.7×. While additional GPUs improve all systems, Ken remains the most efficient across all settings.

We note a few observations:

• AlpaServe+ generally outperforms MS+, highlighting the value of model cascades under load.

• On Sentiment-140, some cascades outperform the most accurate standalone model (BERT-Base) because cheaper models confidently classified some inputs correctly that larger models would have misclassified. This effect only appears for some workloads [79].

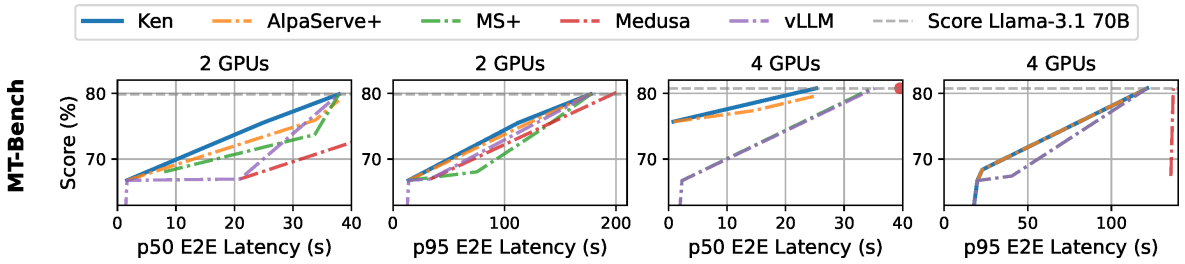


Figure 9: Total-Generation-Time latencies on MT-Bench. While Ken does not optimize this metric, it does not degrade it.

• Medusa does not consistently outperform baseline models, as it disables its optimizations under high load.

• For Sentiment-140, Ken loads all fine-tuned BERT models onto each GPU as all models jointly occupy <5% of one GPU's memory. This allows Ken to precisely choose the cascade that is optimal for each QPS range without needing to compromise based on memory constraints. For different QPS ranges Ken selects cascades of 1, 2, and 3 models. For HellaSwag, Ken loaded {OpenLlama-3B, Llama-13B, Llama-70B} with Llama-70B parallelized across 2 GPUs and each GPU additionally either containing an entire OpenLlama-3B or Llama-13B model. This reflects that high QPS arrival rates require a cheap model (OpenLlama-3B) but cascades using Llama-13B can achieve further accuracy gains at medium-level arrival rates. Ken uses 67% of GPU memory for weights, which is compatible with HellaSwag because its input sequences are relatively short and only generate one output token. Ken selects cascades of length 1 and 2. For MT-Bench, Ken loaded either {Llama-1B, Llama-70B}, {Llama-3B, Llama-70B}, or {Llama-3B} and loaded each model with 2-way tensor parallelism. These placements use < 46% of GPU memory for model weights, accounting for the fact that input and output sequences in MT-Bench may be of moderate length and require more memory for KV caches. Ken selects cascades of length 1 or 2.

Figure 9 additionally shows the Total-Generation-Time (TGT) latency-quality trade-off for the MT-Bench workload. TGT is dominated by inter-token generation speed (i.e., the decoding phase), which Ken does not aim to optimize. Nevertheless, Figure 9 shows that Ken does not harm TGT and even slightly outperforms the baselines in some settings.

5.4 UDBMS Query Optimization over Gear Plans

We now evaluate whether a simple UDBMS query optimizer can select operator configurations that improve the query pipeline's latency–accuracy trade-off. Specifically, we examine how the optimizer's performance changes as it is given more alternative implementations produced by the Ken cascade enumeration process.

We evaluate on the Multi-Modal Question Answering (MultiModalQA) benchmark [72], following prior work on UDBMS query optimizers [64]. We adopt the best-performing pipeline from the MultiModalQA paper (*ImplDecompose*) to define the logical operators in each query, allowing the query optimizer to configure them. Execution configurations may use the fine-tuned RoBERTa models [52] (335M parameters) from the original paper, as well as Llama-3.2 3B, Llama-3.1 8B, and Llama-3.3 70B [26].

We report per-operator latency to account for differences in operator counts across queries. Our evaluation focuses on text and table modalities, as Ken currently does not support image inputs (though this is not a fundamental limitation).

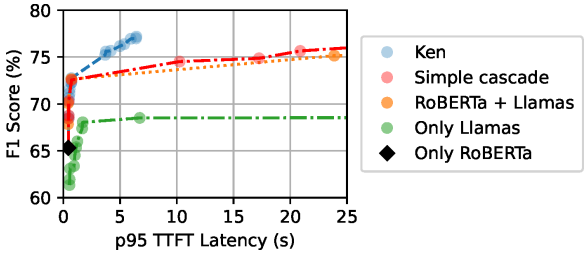


Figure 10: Optimizing a query with multiple operators.

We use a simple UDBMS optimizer that uniformly samples operator configurations. We sample for one hour per baseline, though it typically converges to a Pareto frontier within a few minutes. To efficiently explore many query pipelines, we simulate execution results. The simulator's accuracy is validated in supplementary material A and found to be sufficient.

We compare Ken to several baselines. Ken placees all cascade models using its optimizer, while in baselines, each LLM runs on a dedicated GPU and smaller RoBERTa models are collocated to avoid under-utilization. The baselines' scheduler uses a round-robin load balancer that interleaves RoBERTa and Llama model invocations. Baselines do not adapt dynamically to query load. The "RoBERTa + Llamas" baseline allows the optimizer to choose either a RoBERTa or a Llama model for each operator. The "Simple Cascade" baseline enumerates RoBERTa → Llama cascades. Finally, we include baselines using only Llama or only RoBERTa models.

Figure 10 compares Ken with the baselines. The "RoBERTa + Llamas" baseline illustrates the difficulty of combining accurate but slow LLMs (Llama) with faster, less accurate models (RoBERTa). It achieves favorable trade-offs up to an F1 score of 73%, but beyond that, replacing key operators with LLMs sharply increases latency. To match the maximum F1 score of the Ken pipeline, this baseline reaches a p95 TTFT latency of 778s — 122× higher than Ken's 6.4 s at comparable accuracy. The "Simple Cascade" baseline explores more fine-grained workload configurations but suffers from inefficient model placement. It loads only a single Llama model shared across all operators. Furthermore, in this particular workload, cascades were not able to achieve significant FLOP savings at given accuracy levels. As a result, it performs only marginally better than "RoBERTa + Llamas" and its main advantage lies in providing more trade-off points. Ken outperforms both by adapting to load and optimizing model placement. It makes more Llama invocations within the same latency regime than the baselines, yet maintains low latency by issuing them during low-load periods, avoiding queuing delays. This adaptive behavior enables Ken to sustain high accuracy with substantially better efficiency.

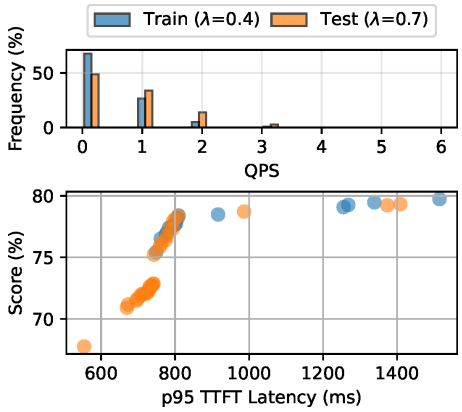


Figure 11: Gear plan performance is evaluated on the orange QPS distribution. Blue gear plans are optimized on the blue train distribution and compared against ones that are optimized on the orange test distribution directly.

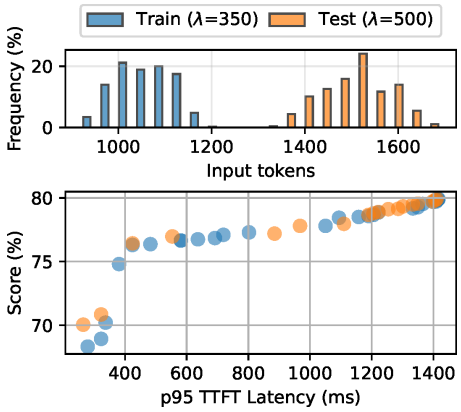


Figure 12: Gear plan performance is evaluated on the orange input length distribution. Blue gear plans are optimized on the blue train distribution and compared against ones that are optimized on the orange test distribution directly.

5.5 Robustness to Workload Misanticipation

Ken optimizes gear plans with respect to a sample workload (§4). We now evaluate Ken's robustness to deviations between the offline and online workloads using the MT-Bench benchmark. Specifically, we test two cases: (i) unrepresentative QPS distribution and (ii) unrepresentative prompt lengths. We model the deviating workloads using Poisson processes, which is commonly used to model workload properties [76]. We show our results in Figures 11 and 12. The bar charts at the top if each plot illustrate the sampled distributions of the respective workload characteristics. For reference, we analyze daily variations in production traces in in supplementary material D; compared to these variations, the ones shown in Figures 11 and 12 are more pronounced.

Figures 11 and 12 show gear plans (blue) that are evaluated on a test workload after being optimized with respect to a significantly different train workload. We evaluate these gear plans (blue) against gear plans that were optimized with respect to the ground truth test distribution (orange). The comparable performance of the two sets of gear plans indicates that Ken is robust to deviations between offline and online workloads. This robustness arises because Ken matches the throughput requirements for each workload strain (i.e., QPS range and number of queued tokens), making the same gear plan effective across differing workloads. Likewise, the model placements within the same regime are consistent between the orange and blue sets of gear plans, as Ken identifies the optimal configuration to sustain high QPS ranges.

However, Figure 11 shows that gear plans derived from the blue trace do not fully cover the Pareto frontier achieved by the Oracle on the orange trace. This occurs because the blue trace has lower query volumes, making cheaper plans irrelevant. Consequently, Ken omits these plans, as their benefits are not observable in the offline workload. This gap is an artifact of the large deviation between the two workloads, which operate in different QPS regimes.

5.6 Gear Plan Optimization

We evaluate the gear plan optimizer using the simulator from Section 4.1 to measure accuracy and latency.

Figure 13 compares plans produced by Ken's planner to two baselines: exhaustive search and random sampling. To make exhaustive search feasible, we constrain the search space by (i) assuming all models fit on each GPU (maximal replication), (ii) fixing batch size to 1, and (iii) using a short, 8-second workload trace. All methods operate under these constraints. The exhaustive baseline tries all possible cascade-to-QPS assignments; the random baseline samples assignments uniformly, with a runtime budget 2× that of Ken.

Ken's planner completes in 0.1s (Llama) and 1s (BERT). The longer runtime for BERT is due to higher QPS, which increases simulation cost. Exhaustive search takes 9 minutes (Llama) and 16 minutes (BERT). While Ken may miss some frontier plans compared to exhaustive search, it closely approximates the Pareto frontier and significantly outperforms random sampling.

Figure 14 shows the runtime of Ken across full search spaces for both workloads. Planning time scales with the number of QPS ranges $|Q|$ (§4). We report submodule call counts and wall-clock times, marking the $|Q|$ used in our experiments with the dashed vertical line. For realistic settings, planning completes within minutes. Even with larger $|Q|$, the cost remains acceptable, as planning is a one-time offline step.

5.7 Ablation Study

Finally, Figure 15 analyzes the factors driving Ken's performance. "No Switching" uses a static cascade to serve inferences and "No cascade" switches between single models. Both baselines use Ken optimizer for model placement and scheduling. "Placement heuristic" runs static cascades but either places each model on exclusive GPUs or, if all models fit onto one GPU, collocates all. Figure 15 shows that the ablated optimizations significantly contribute to Ken's final performance. "Placement heuristic" places all models onto the same GPU for Sentiment-140, precisely matching what "No switching" does. For the HellaSwag workload, "Placement heuristic" sees significant throughput regressions for multi-model cascades where each model cannot be replicated as frequently (due to the exclusive placement policy). At the plotted latency regime, "Placement heuristic" therefore regresses to using a single model.

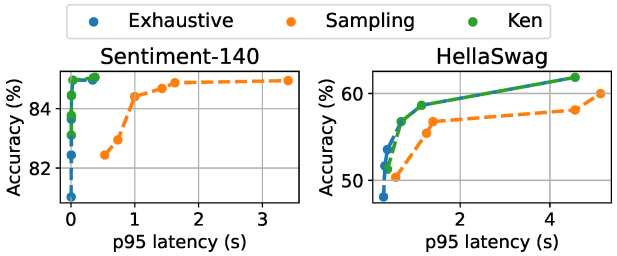


Figure 13: Quality of plans found by gear planner.

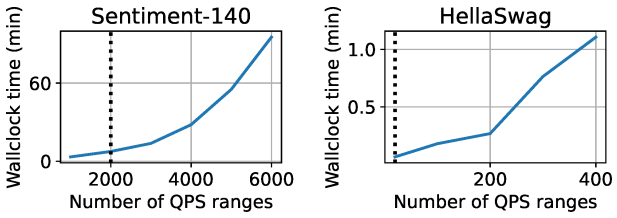


Figure 14: Cost of offline planning.

6 RELATED WORK

Ken aims to leverage the well-studied mechanism of model cascades to (i) achieve fine-grained latency-accuracy trade-offs, and (ii) improve the latency-accuracy Pareto frontier over existing systems. To this end, Ken is the first work that identifies and overcomes the challenges of serving general online requests via cascades on GPUs. In this section, we discuss related work to Ken.

Optimizing model requests. There is a large corpus of works on efficient ML inference. We discuss a subset of these works, focusing on the ones that are most relevant to Ken.

Some model serving systems leverage bagging ensembles [22, 32, 35, 81], which run multiple models independently in parallel and improve accuracy by aggregating their predictions. However, unlike cascades, bagging ensembles are not suited for fine-grained latency–accuracy trade-offs, as they adjust computation only at a coarse granularity (a model is always executed or never). Consequently, unlike Ken, these systems do not target fine-grained latency–accuracy trade-offs and avoid the challenges of conditional cascading described in Section 1.

Other works dynamically adapt their service to the incoming query rate [41, 55, 63, 96]. Unlike these approaches, Ken leverages model cascades to achieve finer-grained adaptation. We compare Ken to MS [96] in Section 5 and find that this finer granularity gives Ken performance advantages.

Most prior works optimize the performance of a fixed single model. Some works focus on multi-tenant settings, where multiple models share common hardware resources [20, 31, 34, 36, 37, 48, 49, 58, 71, 85, 91, 94, 95]. In addition, several systems explore complementary approaches to improve the efficiency of serving a single ML model [1–7, 42, 45, 75, 100–102]. These works are orthogonal to Ken and some could be integrated to further enhance performance.

Optimizing ML applications. There is a growing corpus of research around end-to-end optimization of ML applications.

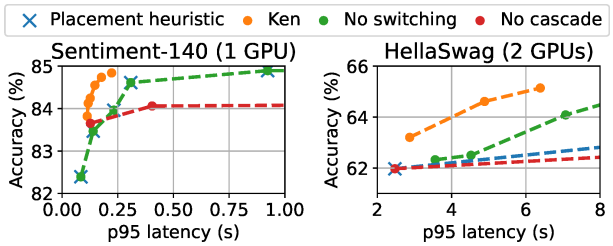


Figure 15: Performance when disabling optimizations.

This paper focuses on *UDBMSes*, which optimize semantic queries over data [9, 51, 59, 66, 68]. Prior works have explored several directions towards efficiently running semantic queries, including query optimization [39, 59, 64, 65, 69], programming models [14, 62, 68] and operator implementations [10, 13, 40, 53]. For example, *Probabilistic Predicates* [53] aim to filter samples and avoid predicting some of them entirely (e.g., skip frames in video object detection). Generally, these works optimize components other than the *execution engine*, which runs ML models and is the focus of this work.

Beyond UDBMSes, several systems optimize applications with multiple model invocations. Teola [73] and Parrot [50] address general ML applications, leveraging application-level knowledge to jointly schedule and optimize dependent LLM requests. Other efforts target multi-stage ML pipelines [8, 17, 21, 43, 70, 86], focusing on optimizing execution for a *fixed* application logic. In contrast, Ken optimizes individual ML requests by *selecting* cascades and *co-optimizing* them with model placement and invocation scheduling.

Model Cascades. Model cascades are a well-established approach for reducing computational cost while preserving accuracy across diverse tasks including object detection [16, 80], classification [47, 82, 88], regression [87], and text generation [24, 92, 97]. NoScope [40] accelerates video object search, FrugalGPT [18] and FrugalML [19] apply cascades to reduce the cost of querying black-box model APIs, and Willump [44] optimizes feature computation for classification queries. However, these systems are domain- or interface-specific and cannot serve general ML requests as Ken does. Ken can leverage their specialized cascade designs to enhance its performance.

7 CONCLUSION

In this work, we address the crucial problem that machine learning models often only allow for coarse-grained trade offs between cost and accuracy, such as choosing between a small, specialized model or a Large Language Model that incurs orders of magnitude more computation. We introduce Ken which leverages model cascades to interpolate trade-off points between models. Cascades can harm query latencies since they incur larger memory footprints and additional data transfer from GPU memory to the arithmetic units. Ken addresses these challenges by dynamically selecting which cascade to run and optimizing model placement on GPUs and model invocation scheduling.

ACKNOWLEDGMENTS

We thank the Data Systems and Artificial Intelligence Lab (DSAIL) for supporting this work. Lei Cao is supported by the NSF (DBI-2327954 and SHF-2106621) and Amazon Research Awards.

REFERENCES

[1] [n.d.]. AWS SageMAker. https://aws.amazon.com/sagemaker/ (Accessed on 24 Mar 2024).

[2] [n.d.]. AzureML model serving. https://learn.microsoft.com/en-us/azure/machine-learning/tutorial-deploy-model?view=azureml-api-2 (Accessed on 23 Jan 2024).

[3] [n.d.]. Databricks Model Serving. https://docs.databricks.com/en/machine-learning/model-serving/index.html (Accessed on 24 Mar 2024).

[4] [n.d.]. NVIDIA Triton Inference Server. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html (Accessed on 24 Mar 2024).

[5] [n.d.]. TensorFlow Serving. www.tensorflow.org/serving (Accessed on 24 Mar 2024).

[6] [n.d.]. TorchServe. https://pytorch.org/serve/ (Accessed on 21 Jan 2024).

[7] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00073

[8] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2022. Optimizing inference serving on serverless platforms. *Proc. VLDB Endow.* 15, 10 (June 2022), 2071–2084. https://doi.org/10.14778/3547305.3547313

[9] Jaeho Bang, Gaurav Tarlok Kakkar, Pramod Chunduri, Subrata Mitra, and Joy Arulraj. 2023. Seiden: Revisiting Query Processing in Video Database Systems. *Proc. VLDB Endow.* 16, 9 (May 2023), 2289–2301. https://doi.org/10.14778/3598581.3598599

[10] Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. 2020. MIRIS: Fast Object Track Queries in Video. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1907–1921. https://doi.org/10.1145/3318464.3389692

[11] Favyen Bastani, Songtao He, Ziwen Jiang, Osbert Bastani, and Sam Madden. 2021. SkyQuery: an aerial drone video sensing platform. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Chicago, IL, USA) *(Onward! 2021)*. Association for Computing Machinery, New York, NY, USA, 56–67. https://doi.org/10.1145/3486607.3486750

[12] Favyen Bastani and Sam Madden. 2021. MultiScope: Efficient Video Preprocessing for Exploratory Video Analytics. arXiv:2103.14695 [cs.DB] https://arxiv.org/abs/2103.14695

[13] Favyen Bastani and Samuel Madden. 2022. OTIF: Efficient Tracker Preprocessing over Large Video Datasets. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2091–2104. https://doi.org/10.1145/3514221.3517835

[14] Favyen Bastani, Oscar Moll, and Sam Madden. 2020. Vaas: video analytics at scale. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 2877–2880. https://doi.org/10.14778/3415478.3415498

[15] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads. *arXiv preprint arXiv: 2401.10774* (2024).

[16] Zhaowei Cai, Mohammad Saberian, and Nuno Vasconcelos. 2015. Learning Complexity-Aware Cascades for Deep Pedestrian Detection. In *2015 IEEE International Conference on Computer Vision (ICCV)*. 3361–3369. https://doi.org/10.1109/ICCV.2015.384

[17] Chaokun Chang, Eric Lo, and Chunxiao Ye. 2024. Biathlon: Harnessing Model Resilience for Accelerating ML Inference Pipelines. *Proc. VLDB Endow.* 17, 10 (June 2024), 2631–2640. https://doi.org/10.14778/3675034.3675052

[18] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. arXiv:2305.05176 [cs.LG]

[19] Lingjiao Chen, Matei Zaharia, and James Y Zou. 2020. FrugalML: How to use ML Prediction APIs more accurately and cheaply. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 10685–10696. https://proceedings.neurips.cc/paper_files/paper/2020/file/789ba2ae4d335e8a2ad283a3f7effced-Paper.pdf

[20] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[21] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 477–491. https://doi.org/10.1145/3419111.3421285

[22] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: a low-latency online prediction serving system. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 613–627.

[23] Hanjun Dai, Bethany Yixin Wang, Xingchen Wan, Bo Dai, Sherry Yang, Azade Nova, Pengcheng Yin, Phitchaya Mangpo Phothilimthana, Charles Sutton, and Dale Schuurmans. 2024. UQE: A Query Engine for Unstructured Databases. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 29807–29838. https://proceedings.neurips.cc/paper_files/paper/2024/file/34b3a40ec9752c1ae48fe85fef8fe8dc-Paper-Conference.pdf

[24] Jasper Dekoninck, Maximilian Baader, and Martin Vechev. 2024. A Unified Approach to Routing and Cascading for LLMs. arXiv:2410.10347 [cs.CL] https://arxiv.org/abs/2410.10347

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[26] Aaron Grattafiori et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[27] Hugo Touvron et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] https://arxiv.org/abs/2307.09288

[28] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. OPTQ: Accurate Quantization for Generative Pre-trained Transformers. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=tcbBPnfwxS

[29] Xinyang Geng and Hao Liu. 2023. OpenLLaMA: An Open Reproduction of LLaMA. https://github.com/openlm-research/open_llama

[30] A. Go, R. Bhayani, and L. Huang. 2009. *Twitter Sentiment Classification Using Distant Supervision*. CS224N Project Report 1(2009). Stanford University. 12 pages.

[31] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati

[32] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. 2022. Cocktail: A Multidimensional Optimization for Model Serving in Cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1041–1057. https://www.usenix.org/conference/nsdi22/presentation/gunasekaran

[33] Neha Gupta, Harikrishna Narasimhan, Wittawat Jitkrittum, Ankit Singh Rawat, Aditya Krishna Menon, and Sanjiv Kumar. 2024. Language Model Cascades: Token-Level Uncertainty And Beyond. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=KgaBScZ4VI

[34] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[35] Shenda Hong, Yanbo Xu, Alind Khare, Satria Priambada, Kevin Maher, Alaa Aljiffry, Jimeng Sun, and Alexey Tumanov. 2020. HOLMES: Health OnLine Model Ensemble Serving for Deep Learning Models in Intensive Care Units. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 1614–1624. https://doi.org/10.1145/3394486.3403212

[36] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. 2018. Dynamic Space-Time Scheduling for GPU Inference. In *32nd Conference on Neural Information Processing Systems (NeurIPS 2018)*. Montreal, Canada. arXiv:1901.00041 [cs.DC]

[37] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. 2023. Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 249–265. https://doi.org/10.1145/3552326.3567508

[38] Gaurav Tarlok Kakkar, Jiashen Cao, Pramod Chunduri, Zhuangdi Xu, Suryatej Reddy Vyalla, Prashanth Dintyala, Anirudh Prabakaran, Jaeho Bang, Aubhro Sengupta, Kaushik Ravichandran, Ishwarya Sivakumar, Aryan Rajoria, Ashmita Raju, Tushar Aggarwal, Abdullah Shah, Sanjana Garg, Shashank Suman, Myna Prasanna Kalluraya, Subrata Mitra, Ali Payani, Yao Lu, Umakishore Ramachandran, and Joy Arulraj. 2023. EVA: An End-to-End Exploratory Video

Analytics System. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning* (Seattle, WA, USA) *(DEEM '23)*. Association for Computing Machinery, New York, NY, USA, Article 8, 5 pages. https://doi.org/10.1145/3595360.3595858

[39] Gaurav Tarlok Kakkar, Jiashen Cao, Aubhro Sengupta, Joy Arulraj, and Hyesoon Kim. 2024. Hydro: Adaptive Query Processing of ML Queries. arXiv:2403.14902 [cs.DB] https://arxiv.org/abs/2403.14902

[40] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: optimizing neural network queries over video at scale. *Proc. VLDB Endow.* 10, 11 (aug 2017), 1586–1597. https://doi.org/10.14778/3137628.3137664

[41] Alind Khare, Dhruv Garg, Sukrit Kalra, Snigdha Grandhi, Ion Stoica, and Alexey Tumanov. 2025. SuperServe: Fine-Grained Inference Serving for Unpredictable Workloads. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 739–758. https://www.usenix.org/conference/nsdi25/presentation/khare

[42] Ferdi Kossmann, Bruce Fontaine, Daya Khudia, Michael Cafarella, and Samuel Madden. 2025. Is the GPU Half-Empty or Half-Full? Practical Scheduling Techniques for LLMs. arXiv:2410.17840 [cs.LG] https://arxiv.org/abs/2410.17840

[43] Ferdi Kossmann, Ziniu Wu, Eugenie Lai, Nesime Tatbul, Lei Cao, Tim Kraska, and Sam Madden. 2023. Extract-Transform-Load for Video Streams. *Proc. VLDB Endow.* 16, 9 (may 2023), 2302–2315. https://doi.org/10.14778/3598581.3598600

[44] Peter Kraft, Daniel Kang, Deepak Narayanan, Shoumik Palkar, Peter Bailis, and Matei Zaharia. 2020. Willump: A Statistically-Aware End-to-end Optimizer for Machine Learning Inference. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 147–159. https://proceedings.mlsys.org/paper_files/paper/2020/file/d9e5bd751997cffa6bc2d0e31ebdc048-Paper.pdf

[45] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[46] Katrina LaCurts. 2014. *Application Workload Prediction and Placement in Cloud Computing Environments.* Ph.D. Dissertation. Massachusetts Institute of Technology. https://people.csail.mit.edu/katrina/papers/thesis.pdf

[47] Luzian Lebovitz, Lukas Cavigelli, Michele Magno, and Lorenz K Muller. 2023. Efficient Inference With Model Cascades. *Transactions on Machine Learning Research* (2023). https://openreview.net/forum?id=obB415rg8q

[48] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 611–626. https://www.usenix.org/conference/osdi18/presentation/lee

[49] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 663–679. https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[50] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 929–945. https://www.usenix.org/conference/osdi24/presentation/lin-chaofan

[51] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baille Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. 2024. A Declarative System for Optimizing AI Workloads. arXiv:2405.14696 [cs.CL] https://arxiv.org/abs/2405.14696

[52] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL] https://arxiv.org/abs/1907.11692

[53] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/3183713.3183751

[54] Samuel Madden, Michael Cafarella, Michael Franklin, and Tim Kraska. 2024. Databases Unbound: Querying All of the World's Bytes with AI. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4546–4554. https://doi.org/10.14778/3685800.3685916

[55] Daniel Mendoza, Francisco Romero, and Caroline Trippel. 2024. Model Selection for Latency-Critical Inference Serving. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1016–1038. https://doi.org/10.1145/3627703.3629565

[56] Arko Mondal, M E Hoque, and Mosharaf Chowdhury. 2021. Scheduling of Time-Varying Workloads Using Reinforcement Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. AAAI, 499–506. https://ojs.aaai.org/index.php/AAAI/article/view/17088/16895

[57] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. https://doi.org/10.1145/3458817.3476209

[58] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (<conf-loc>, <city>Koblenz</city>, <country>Germany</country>, </conf-loc>) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 595–610. https://doi.org/10.1145/3600006.3613163

[59] Liana Patel, Siddharth Jha, Carlos Guestrin, and Matei Zaharia. 2024. LOTUS: Enabling Semantic Queries with LLMs Over Tables of Unstructured and Structured Data. *arXiv preprint arXiv:2407.11418* (2024). https://arxiv.org/abs/2407.11418

[60] Miguel Ângelo Rebelo, Duarte Coelho, Ivo Pereira, and Fábio Fernandes. 2022. A New Cascade-Hybrid Recommender System Approach for the Retail Market. In *Innovations in Bio-Inspired Computing and Applications*, Ajith Abraham, Ana Maria Madureira, Arturas Kaklauskas, Niketa Gandhi, Anu Bajaj, Azah Kamilah Muda, Dalia Kriksciuniene, and João Carlos Ferreira (Eds.). Springer International Publishing, Cham, 371–380.

[61] Stewart Robinson. 2004. *Simulation: The Practice of Model Development and Use.* John Wiley & Sons, Inc., Hoboken, NJ, USA.

[62] Francisco Romero, Johann Hauswald, Aditi Partap, Daniel Kang, Matei Zaharia, and Christos Kozyrakis. 2022. Optimizing Video Analytics with Declarative Model Relationships. *Proc. VLDB Endow.* 16, 3 (Nov. 2022), 447–460. https://doi.org/10.14778/3570690.3570695

[63] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[64] Matthew Russo, Sivaprasad Sudhir, Gerardo Vitagliano, Chunwei Liu, Tim Kraska, Samuel Madden, and Michael Cafarella. 2025. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. arXiv:2505.14661 [cs.DB] https://arxiv.org/abs/2505.14661

[65] Matthew Russo, Sivaprasad Sudhir, Gerardo Vitagliano, Chunwei Liu, Tim Kraska, Samuel Madden, and Michael Cafarella. 2025. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. arXiv:2505.14661 [cs.DB] https://arxiv.org/abs/2505.14661

[66] Mohammed Saeed, Nicola De Cao, and Paolo Papotti. 2023. Querying Large Language Models with SQL. arXiv:2304.00472 [cs.DB] https://arxiv.org/abs/2304.00472

[67] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[68] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. arXiv:2410.12189 [cs.DB] https://arxiv.org/abs/2410.12189

[69] Ted Shaowang, Nilesh Jain, Dennis D. Matthews, and Sanjay Krishnan. 2021. Declarative data serving: the future of machine learning inference on the edge. *Proc. VLDB Endow.* 14, 11 (July 2021), 2555–2562. https://doi.org/10.14778/3476249.3476302

[70] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

[71] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Nineteenth European Conference on Computer Systems (EuroSys '24)*. ACM, Athens, Greece, 1–18. https://doi.org/10.1145/3627703.3629578

[72] Alon Talmor, Ori Yoran, Amnon Catav, Dan Lahav, Yizhong Wang, Akari Asai, Gabriel Ilharco, Hannaneh Hajishirzi, and Jonathan Berant. 2021. MultiModal{QA}: complex question answering over text, tables and images. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ee6W5UgQLa

[73] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. 2025. Towards End-to-End Optimization of LLM-based Applications with Ayo. In *Proceedings of the 30th ACM*

International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. https://doi.org/10.1145/3676641.3716278

[74] OpenAI team. 2024. GPT-4o System Card. arXiv:2410.21276 [cs.CL] https://arxiv.org/abs/2410.21276

[75] TensorRT-LLM Team. 2024. *TensorRT-LLM Repository*. https://github.com/NVIDIA/TensorRT-LLM, accessed on 17 Sep 2024.

[76] Marlin U. Thomas. 1976. Queueing Systems. Volume 1: Theory (Leonard Kleinrock). *SIAM Rev.* 18, 3 (1976), 512–514. https://doi.org/10.1137/1018095 arXiv:https://doi.org/10.1137/1018095

[77] turboderp. 2024. ExLlama Github Repository. https://github.com/turboderp/exllama (accessed on May 7, 2024).

[78] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-Read Students Learn Better: On the Importance of Pre-training Compact Models. arXiv:1908.08962 [cs.CL]

[79] Neeraj Varshney and Chitta Baral. 2022. Model Cascading: Towards Jointly Improving Efficiency and Accuracy of NLP Systems. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 11007–11021. https://doi.org/10.18653/v1/2022.emnlp-main.756

[80] P. Viola and M. Jones. 2001. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, Vol. 1. I–I. https://doi.org/10.1109/CVPR.2001.990517

[81] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. 2018. Rafiki: machine learning as an analytics service system. *Proc. VLDB Endow.* 12, 2 (oct 2018), 128–140. https://doi.org/10.14778/3282495.3282499

[82] Xin Wang, Yujia Luo, Daniel Crankshaw, Alexey Tumanov, Fisher Yu, and Joseph E. Gonzalez. 2018. IDK Cascades: Fast Deep Learning by Learning not to Overthink. arXiv:1706.00885 [cs.CV]

[83] Haoran Wei, Chenglong Liu, Jinyue Chen, Jia Wang, Lingyu Kong, Yanming Xu, Zheng Ge, Liang Zhao, Jianjian Sun, Yuang Peng, et al. 2024. General OCR Theory: Towards OCR-2.0 via a Unified End-to-end Model. *arXiv preprint arXiv:2409.01704* (2024).

[84] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. arXiv:1910.03771 [cs.CL]

[85] Xiaorui Wu, Hong Xu, and Yi Wang. 2020. Irina: Accelerating DNN Inference with Efficient Online Scheduling. In *Proceedings of the 4th Asia-Pacific Workshop on Networking* (Seoul, Republic of Korea) *(APNet '20)*. Association for Computing Machinery, New York, NY, USA, 36–43. https://doi.org/10.1145/3411029.3411035

[86] Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. 2022. Serving and Optimizing Machine Learning Workflows on Heterogeneous Infrastructures. *Proc. VLDB Endow.* 16, 3 (Nov. 2022), 406–419. https://doi.org/10.14778/3570690.3570692

[87] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. arXiv:2403.02286 [cs.DB]

[88] Yanbo Xu, Alind Khare, Glenn Matlin, Monish Ramadoss, Rishikesan Kamaleswaran, Chao Zhang, and Alexey Tumanov. 2022. UnfoldML: Cost-Aware and Uncertainty-Based Dynamic 2D Prediction for Multi-Stage Classification. arXiv:2210.15056 [cs.LG]

[89] Zhuangdi Xu, Gaurav Tarlok Kakkar, Joy Arulraj, and Umakishore Ramachandran. 2022. EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 602–616. https://doi.org/10.1145/3514221.3526142

[90] Xifeng Yan, Amol Goel, Chuljin Kim, Rodrigo Fonseca, and Randy H. Katz. 2012. Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium*. IEEE, 1287–1294. https://sites.cs.ucsb.edu/~xyan/papers/noms12_cloudman.pdf

[91] Peifeng Yu and Mosharaf Chowdhury. 2019. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. arXiv:1902.04610 [cs.DC]

[92] Murong Yue, Jie Zhao, Min Zhang, Liang Du, and Ziyu Yao. 2024. Large Language Model Cascades with Mixture of Thoughts Representations for Cost-efficient Reasoning. arXiv:2310.03094 [cs.CL] https://arxiv.org/abs/2310.03094

[93] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a Machine Really Finish Your Sentence?. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Anna Korhonen, David Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, Florence, Italy, 4791–4800. https://doi.org/10.18653/v1/P19-1472

[94] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1049–1062. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang

[95] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 787–808. https://www.usenix.org/conference/nsdi23/presentation/zhang-hong

[96] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. 2020. Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association. https://www.usenix.org/conference/hotcloud20/presentation/zhang

[97] Kai Zhang, Liqian Peng, Congchao Wang, Alec Go, and Xiaozhong Liu. 2024. LLM Cascade with Multi-Objective Optimal Consideration. arXiv:2410.08014 [cs.CL] https://arxiv.org/abs/2410.08014

[98] Zeyu Zhang, Paul Groth, Iacer Calixto, and Sebastian Schelter. 2025. ANYMATCH - Efficient Zero-Shot Entity Matching with a Small Language Model. In *Proceedings of the GOOD-DATA Workshop at AAAI 2025*. https://openreview.net/forum?id=Nees1OD5td Accepted at the GOOD-DATA Workshop, AAAI 2025.

[99] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. arXiv:2306.05685 [cs.CL]

[100] 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. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104 [cs.AI] https://arxiv.org/abs/2312.07104

[101] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. 2025. BatchLLM: Optimizing Large Batched LLM Inference with Global Prefix Sharing and Throughput-oriented Token Batching. arXiv:2412.03594 [cs.CL] https://arxiv.org/abs/2412.03594

[102] Lixi Zhou, Jiaqing Chen, Amitabh Das, Hong Min, Lei Yu, Ming Zhao, and Jia Zou. 2022. Serving deep learning models with deduplication from relational databases. *Proc. VLDB Endow.* 15, 10 (June 2022), 2230–2243. https://doi.org/10.14778/3547305.3547325