

DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency

Jovan Stojkovic, Chaojie Zhang[†], Íñigo Goiri[†], Josep Torrellas, Esha Choukse[†]
 University of Illinois at Urbana-Champaign [†]Microsoft Azure Research - Systems

Abstract—The rapid evolution and widespread adoption of generative large language models (LLMs) have made them a pivotal workload in various applications. Today, LLM inference clusters receive a large number of queries with strict Service Level Objectives (SLOs). To achieve the desired performance, these models execute on power-hungry GPUs causing the inference clusters to consume large amount of energy and, consequently, result in excessive carbon emissions. Fortunately, we find that there is a great opportunity to exploit the *heterogeneity in inference compute properties and fluctuations in inference workloads*, to significantly improve energy-efficiency. However, such a diverse and dynamic environment creates a *large search-space* where different system configurations (e.g., number of instances, model parallelism, and GPU frequency) translate into different energy-performance trade-offs. To address these challenges, we propose *DynamoLLM*, the first energy-management framework for LLM inference environments. DynamoLLM automatically and dynamically reconfigures the inference cluster to *optimize for energy* and cost of LLM serving under the service’s performance SLOs. We show that at a service-level, DynamoLLM conserves 53% energy and 38% operational carbon emissions, and reduces 61% cost to the customer, while meeting the latency SLOs.

I. INTRODUCTION

The exponential growth in the adoption of generative large language models (LLMs) has positioned them at the core of numerous technological advancements and applications. Today, we see use-cases of LLMs in various domains, such as healthcare [52], developer productivity [13], data analytics [68], education [5] and other. As the popularity of LLMs increases among users, the inference clusters receive millions of queries per day [27] resulting in large infrastructures with sophisticated software and expensive hardware systems.

To meet these ever increasing computing demands, researchers proposed various software [9], [26], [35], [73], [81] and hardware [4], [50], [78] techniques. Such techniques improve the performance efficiency of LLM inference clusters. However, one aspect that has been largely overlooked is the energy consumption of these environments [58], [60]. The substantial energy requirements of serving LLMs running on power-hungry GPUs have emerged as a significant concern. As these models become integral to various services, minimizing their energy consumption and, consequently, carbon emissions while maintaining high performance is paramount.

To address this gap, this paper starts by characterizing the *energy-efficiency properties of LLM inference workloads*. Our characterization underscores that such environments present a distinct set of challenges, divergent from existing energy management schemes tailored for traditional datacenters appli-

cations [7], [17], [21], [31], [61], [80]. Specifically, we observe that *heterogeneity in LLM inference compute properties and fluctuations in LLM inference workloads* create a dynamic environment with large variations. Such variations arise from: (1) requests with varying input/output token lengths, (2) distinct compute properties of different LLMs, and (3) different SLOs required by the services using an LLM.

Requests with a large number of input tokens are compute intensive, thus, sensitive to GPU frequency. Conversely, requests with a few input tokens and many output tokens have low compute, but high memory requirements. Reducing their GPU frequency would save the energy without significantly impacting the performance. Moreover, the number of model parameters also affects the LLM’s sensitivity to the number of GPUs and GPU frequency. Finally, depending on the service currently using the LLM, the *SLO requirements* can be strict requiring high-performance configurations, or loose allowing for lower-performance but more energy-efficient configurations. Importantly, *these characteristics rapidly change* due to load fluctuations and dynamic distributions of requests. Such dynamic changes cause a system configuration that is energy-efficient at a given point, to quickly become sub-optimal. This requires a *dynamic approach* to resource management.

To pave the way towards energy-efficient and sustainable LLM inference clusters, this paper introduces *DynamoLLM*, the first energy-management framework for LLM inference environments. DynamoLLM exploits the unique properties of LLM inference workloads to reduce their energy consumption while meeting the performance SLOs. The system uses *energy-performance profiles of models and their workloads* to *automatically and dynamically select the energy-efficient configuration*. It leverages multiple knobs, including scaling in/out the number of server instances, model parallelism across GPUs, and GPU frequency scaling.

To handle workload heterogeneity, DynamoLLM *maintains differently configured pools of LLM instances* that are optimal for different types of incoming requests. For instance, compared to a request with many input and output tokens, a request that processes and outputs fewer tokens runs more efficiently on a model parallelized across fewer GPUs running at a lower frequency. As request distribution varies over time, DynamoLLM *dynamically sizes the pools*. These pools can be merged into fewer pools or divided into multiple pools over time, *providing a balance between right-sizing and fragmentation of resources*. To efficiently manage the

resources, DynamoLLM uses a hierarchy of controllers that **reduces computation complexity** and **eliminates centralized bottlenecks**. The controller at each level operates under the conditions imposed by the upper level, computes its dedicated knob, and forwards further constraints to the controllers at a lower level. Finally, to enable frequent and smooth transition across different configurations, DynamoLLM includes techniques to **minimize or hide the reconfiguration overheads**. As a result, the system maintains high levels of efficiency and service quality under changing workload demands.

An evaluation of DynamoLLM with a large GPU cluster running production-level traces from a major cloud provider shows that DynamoLLM is very effective: it conserves 53% energy, 38% operational carbon emissions, and reduces 61% cost to the customer, while meeting the latency SLOs.

The contributions of this paper are as follows:

- An analysis of the opportunities for energy-efficient LLM serving, rooting in the heterogeneity and fluctuations within the inference workloads.
- Design and implementation of DynamoLLM, a high performance and energy-optimized framework for LLM inference.
- An evaluation of DynamoLLM on a large-scale platform using production-level traces.

II. BACKGROUND

Computational phases of LLMs Generative LLMs [34], [38], [56], [67], [80] are auto-regressive: they process the whole input in parallel, and serially generate the output tokens. This property leads to two computationally distinct phases [49], [50]. First is the prefill phase, where the input tokens are computed in parallel. This is a compute-intensive phase and scales with the number of input tokens. Second is the decode phase, where each output token is generated serially, based on all the tokens seen so far. This is a memory-intensive phase, and scales with the number of output tokens.

Performance metrics for LLMs To evaluate the performance, we use: time to first token (TTFT), time between tokens (TBT), and throughput [50], [63]. TTFT is the latency of generating the first output token; while TBT is the latency to generate each new output token. To quantify the energy efficiency, we measure the energy consumption in Watt-hours (Wh) while meeting certain latency SLOs. The SLOs vary depending on their use cases for different tasks. For latency-sensitive tasks, both TTFT and TBT are important metrics with strict SLOs. We define SLOs for TTFT and TBT based on maximum achievable performance, described in Table IV.

LLM parallelism A single model can be divided across GPUs to improve performance and allow larger memory footprints. LLM inference typically uses pipeline and tensor parallelism. Pipeline parallelism (PP) partitions the LLM layers among GPUs, while keeping all the operators/tensors of a layer on the GPU. GPUs then communicate only in between two consecutive stages. Tensor parallelism (TP) allocates a slice of each layer to each GPU. This requires aggregation across all the GPU for each layer, in turn needing high bandwidth

communication. TP performs better for GPUs within the same server, connected with high bandwidth interconnects (e.g., NVLink [45]), while PP is preferred across servers. Since most open source models [34], [38], [67] fit on 8 GPUs in a single server, we consider only TP in the rest of the paper; the ideas can easily extend to PP. We denote tensor parallelism across 2, 4 and 8 GPUs as TP2, TP4 and TP8, respectively.

Power and energy in datacenters A rich body of work explored power/energy efficiency in traditional datacenters [7], [29], [31], [62]. However, the rapid growth of LLMs has posed new challenges that have not yet been extensively studied. LLM inference workloads comprise a swiftly increasing percentage of datacenter load [49]. This, coupled with the power-dense hardware like DGX A100s and H100s being deployed to serve these workloads makes them power, energy, and carbon-intensive [12], [49], [58]. To effectively address this challenge, it is important to have a comprehensive framework for managing energy in these systems.

III. OPPORTUNITIES FOR ENERGY EFFICIENCY

To understand the energy-efficiency properties of LLM inference environments, we characterize open-source models [33], [38], [39], [66] on an NVIDIA DGX H100 server [44] using vLLM [26] inference engine. We analyze the energy properties of LLMs by varying the request lengths, request load, model, and service SLO. Additionally, we analyze how the profiled variables change over time in a real-production environment using the invocation traces of two LLM services from *Azure: Coding* and *Conversation*. The traces include a subset of invocations received by the profiled services during one week, and contain the timestamp of the invocation, along with the number of input and output tokens. These traces are a super-set of our open-source traces for the same services [50].

A. Heterogeneous Energy-Performance Profiles

Request lengths The prefill and decode phases in an LLM inference exhibit distinct execution behaviors (Section II), suggesting that requests of *different input and output lengths* possess different compute and energy characteristics. We categorize the requests based on the number of input/output tokens into 9 buckets: SS (short input, short output), SM (short input, medium output), SL (short input, long output), MS, MM, ML, LS, LM, and LL. Table IV shows the thresholds and corresponding TTFT/TBT SLOs. We set the thresholds for request lengths using the 33rd, 66th and 100th percentiles of the input/output lengths from a trace for a *Conversation* service from *Azure*. We set the SLOs to 5× the latency of a single request running isolated on a system [30].

We use these categories to characterize the energy consumption of different request types running the Llama2-70B [33] model with a medium system load of 2000 tokens per second (TPS) under various GPU frequencies and model parallelisms. Table I shows our results in the form of a heat map. Since shorter requests are not computationally intensive, they meet their SLOs with any tensor parallelism, and generally at lower

Tensor Parallelism		TP2				TP4				TP8			
GPU Frequency (GHz)		0.8	1.2	1.6	2.0	0.8	1.2	1.6	2.0	0.8	1.2	1.6	2.0
Input	Output												
Short	Short		0.77	0.97	1.03	0.94	0.79	0.91	1.01	1.35	1.19	1.29	1.49
Short	Medium		2.78	3.45	3.68	3.39	2.82	3.37	3.81	4.55	4.15	4.43	4.74
Short	Long					4.84	4.17	4.97	5.52	6.37	5.62	5.59	6.95
Medium	Short			1.02	1.09		1.08	1.07	1.20	1.51	1.29	1.34	1.73
Medium	Medium						4.23	3.91	4.08	5.34	4.39	4.56	5.44
Medium	Long						4.99	4.66	4.53	6.86	5.79	6.52	7.12
Long	Short						1.51	1.64	1.76	2.55	2.53	2.83	2.94
Long	Medium										7.71	8.81	9.17
Long	Long										12.99	11.89	13.21

TABLE I: Energy consumption in Watt×hours (Wh) for Llama2-70B varying request lengths, frequency, and model parallelism with medium system load (2K tokens per second). Configurations that violate the SLO are shown as empty gray boxes, while the acceptable configurations are colored as a heat map according to their energy consumption, per row.

Tensor Parallelism		TP2				TP4				TP8			
GPU Frequency (GHz)		0.8	1.2	1.6	2.0	0.8	1.2	1.6	2.0	0.8	1.2	1.6	2.0
Low Load				3.41	3.75	3.44	2.93	3.71	3.73	4.49	3.76	4.52	4.64
Medium Load							4.23	3.91	4.08	5.34	4.39	4.56	5.44
High Load								4.22	4.13	5.86	5.24	5.42	6.62

TABLE II: Energy consumption in Wh for LLama2-70B medium-sized input and output (MM) requests varying frequency and model parallelism under different system loads: low (650 TPS), medium (2K TPS) and high (4K TPS).

Tensor Parallelism		TP2				TP4				TP8			
GPU Frequency (GHz)		0.8	1.2	1.6	2.0	0.8	1.2	1.6	2.0	0.8	1.2	1.6	2.0
Llama2-13B [32]		1.05	0.99	1.14	1.24	1.52	1.27	1.58	1.65	2.61	2.35	2.74	3.45
Mixtral-8x7B [39]		1.03	0.98	1.21	1.32	1.39	1.51	2.09	2.31	2.57	3.06	3.71	4.66
Llama2-70B [33]							4.23	3.91	4.08	5.34	4.39	4.56	5.44
Llama3-70B [34]							4.32	4.28	4.57	6.11	5.18	5.42	6.45
Mixtral-8x22B [38]										3.83	3.23	3.65	4.03
Falcon-180B [66]										9.56	7.94	8.57	10.34

TABLE III: Energy consumption in Wh for medium-sized (MM) requests of different LLM architectures varying the frequency and model parallelism with medium system load (2K TPS).

	Input	Output	TTFT SLO	TBT SLO
Short	S	<256	<100	250 ms
Medium	M	<1024	<350	400 ms
Long	L	≤8192	≥350	2000 ms

TABLE IV: Thresholds for classifying the requests based on input/output lengths and corresponding TTFT/TBT SLOs.

frequencies compared to the rest. As an example, the least-energy configuration for SS requests is TP2 at 1.2 GHz. Conversely, LL requests can only run with TP8 without violating the SLO. With TP8, the least-energy configuration for LL requests is 1.6 GHz. Note that the lowest power configuration that meets SLOs (TP8 at 1.2 GHz), is not the energy-optimal one due to the increased execution time. Running all the requests together would require the system to run with the most constrained SLO configuration, in this case, as per the LL configuration. This would make the system energy inefficient.

To exploit this heterogeneity for energy-efficiency, the system would need to separate requests based on their input/output lengths, and process different request types with different server configurations. However, on request arrival, the input length is known, but, due to the auto-regressive LLM nature,

the output length is unknown. Thus, the system needs to predict the output length. DynamoLLM will rely on prior work that efficiently performs such operation with relatively high precision [19], [55], [79], and will have a mechanism to mitigate the impact of occasional mis-predictions.

Request loads In addition to the request length, the incoming load of the LLM inference server drives the compute requirements. During periods of low load, the system has a larger SLO slack to exploit and can run the requests at low-frequency configurations to save energy. Conversely, during periods of high load, the system does not have enough SLO slack, and needs to run at high-frequency configurations.

Table II shows the energy consumed when running Llama2-70B medium-sized input and output (MM) requests while varying the number of processed prompt tokens per second (TPS). The system can run low load with any TP at almost any frequency. Among all feasible configurations, the lowest-energy configuration is TP4 with 1.2 GHz. TP8 requires more GPUs to operate in parallel and, thus, consumes more energy. TP2 uses fewer GPUs but increases the execution time and forces individual GPUs to operate at high frequency to meet SLOs, leading to high energy. Conversely, under high load,

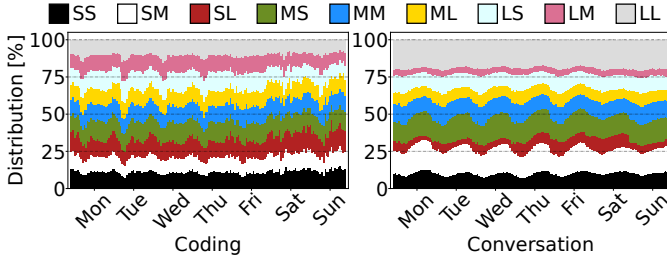


Fig. 1: Distribution of requests based on input and output lengths categorized into three groups: short, medium, and long.

the system cannot operate on TP2 and requires TP4 or TP8. The lowest energy configuration is TP4 with 2 GHz. Overall, to minimize the energy consumption while operating under performance constraints, we need to consider the incoming load to set the correct parallelism and GPU frequency.

Requested model The diversity of the compute properties of an LLM directly translates into its energy profile. Table III shows the energy consumption of different LLMs when running medium-sized requests at medium system load. Smaller models, such as Llama2-13B and Mixtral-8x7B, can run with any TP (even with a single GPU); their lowest-energy configuration is TP2 at 1.2 GHz. Mixtral-8x22B and Falcon-180B are much larger and can only run with TP8. Their lowest-energy configuration is TP8 at 1.2 GHz.

Compute-bound models with large number of parameters are more sensitive to the GPU frequency and model parallelism. Hence, they often need to operate at high-frequency and high-energy modes. Sparse models with relatively smaller numbers of parameters tolerate lower frequencies and lower model parallelism. Hence, they meet the performance requirements even with lower-performance modes.

Service SLO Different services often use the same model with different SLO requirements [57]. As indicated before, we assume an SLO such that the P99 tail latency is within $5\times$ of the execution time of a request on an unloaded system [30]. However, some services have more relaxed SLOs, at $10\times$ or even $20\times$ of a single request execution [10], [37]. For different SLO requirements, the system may need different energy-optimal configurations. For example, Table I shows that, with strict SLO ($5\times$), short-input long-output sized Llama2-70B requests at medium load have the optimal configuration at TP4 and 1.2GHz. However, if with loose SLO ($10\times$), the requests may even operate with TP2 at 1.6GHz.

Insight #1 LLM workloads are highly heterogeneous in their energy-performance profiles. To achieve the optimal energy under performance SLOs, different requests (sizes, models and SLOs) need to be processed separately and differently.

B. Dynamic LLM Inference Workloads

Changing request-length distribution We measure the distribution of request types for *Coding* and *Conversation* services. Figure 1 shows the distribution of requests for each workload over a week. The distribution differs across services. *Conversation* has typically longer outputs and shorter inputs, while

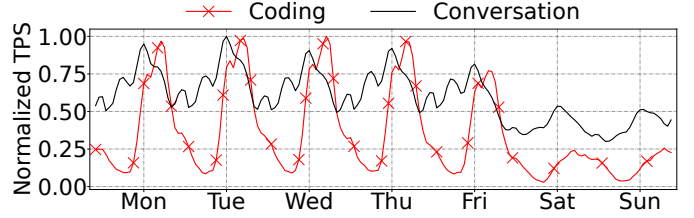


Fig. 2: Load over a week for *Coding* and *Conversation* LLM inference workloads.

Coding shows the opposite trend. However, both services have a significant fraction of each request type, and importantly, the popularity of request types changes over time.

As observed earlier, different request types require different energy-optimal configurations. Thus, the system needs to split its resources into per request-type pools, configure pools individually, and dynamically adapt the pools' configurations based on the current request distribution. However, if the system classifies the requests into too few classes, it will not be able to fine-tune the system for best energy. On the other hand, too many classes may lead to fragmentation and negatively impact energy efficiency. Thus, the system has to find the right number of resource pools. In DynamoLLM, we will use historical data to set the number of pools such that requests with distinct SLO requirements (TTFT or TBT bound) and compute properties (compute or memory bound) have separate pools. Moreover, as the load of a given request type reduces, DynamoLLM will avoid fragmentation by merging the pool with the next available pool that serves longer requests.

Request load fluctuations LLM inference workloads, as user-facing applications, exhibit a typical diurnal pattern with peaks during working hours and valleys at night and weekends. Figure 2 shows the load in tokens per second of the two workloads over a week. The load is normalized to the peak of the individual workloads. The *Coding* trace shows a clear diurnal pattern, with peaks every day, lower load at night, and much lower load during weekends. *Conversation* shows a less extreme, but still significant, diurnal pattern.

The peak load of *Conversation* is $1.7\times$ and $3.3\times$ higher than its average and valley loads, respectively. The peak load of *Coding* is $2.8\times$ and $34.6\times$ higher than its average and valley loads, respectively. This large slack indicates that the LLM inference servers can frequently operate in a less performant but energy-optimized configuration without violating the SLO. Once the load starts building up, the server needs to switch to a more performant mode of operation.

LLM service SLO and model diversity Finally, different services may time-share the same LLM model instance [14]. They may have different SLOs, requiring the configuration to be adapted based on the current service-user. On the other hand, the same service may concurrently use multiple different models [11]. This requires different execution plans for the optimal energy consumption of the individual queries. Thus, it is not trivial for service providers to operate in an energy-optimal setting while meeting the performance SLOs.

Overhead source	Time
Create a new H100 VM [36]	~1-2 min
Initialize distributed multi-GPU environment	~2 min
Download model weights (Llama2-70B [67])	~3 min
Set up the engine configuration	~18 sec
Install weights and KV cache on GPUs	~15 sec
Total	~6-8 min

TABLE V: Measured overheads of creating a new $8\times$ H100 instance of an LLM inference server VM.

Insight #2 LLM workloads are highly dynamic and, thus, an energy-optimal configuration can quickly become sub-optimal. However, the complexity of a large search space requires an automatic and user-transparent configuration selection.

C. Reconfiguration Overheads

To capture the fast changes in LLM inference workloads, we need to quickly transition between configurations. However, there are overheads to change (1) number of inference server instances, (2) model parallelism, and (3) GPU frequency.

Changing instance number To adjust to fluctuating load, it is cost-beneficial to dynamically adjust the number of LLM instances to serve the requests (*i.e.*, scale in and out). However, the overheads of adding a new inference server are too large to be tolerable on the critical path of inference loads. Table V shows the breakdown of the overheads to: (1) instantiate a new GPU VM in the cloud (such as H100 VM [36]), (2) initialize the distributed multi-GPU environment (*e.g.*, Ray, MPI), (3) download the model weights, (4) setup the inference engine, and (5) install the weights and key-value cache on the GPUs. In total, these overheads can take even a few minutes. Hence, the conventional LLM inference environments typically provision the static number of instances to handle their peak load resulting in heavy underutilization. In DynamoLLM, we will propose techniques to efficiently scale the number of instances (with the current load) while minimizing most of the scale-out overheads.

Changing model parallelism To modify the model parallelism of an LLM inference server, we need to perform two operations. First, we need to re-shard the model weights and transfer them to the memory of the right GPUs. Second, the inference engine needs to synchronize the involved GPUs. Current systems stop the engine, unload the weights from GPUs, load the weights from the host to the new set of GPUs, and re-start the engine from the scratch. This adds intolerable overheads (around 1-2 minutes) if performed on the critical path. In DynamoLLM, we will show how to minimize the re-sharding overheads by smartly mapping the logical to physical GPUs, exploiting inter-GPU direct NVLink connections and moving the weights between GPUs in the background.

Changing GPU frequency Setting the GPU frequency (*e.g.*, via `nvidia-smi` [46]) incurs non-negligible overheads. It involves invoking the OS, communicating with the GPU driver via system calls, and performing hardware interactions via firmware. On average, setting the GPU frequency takes

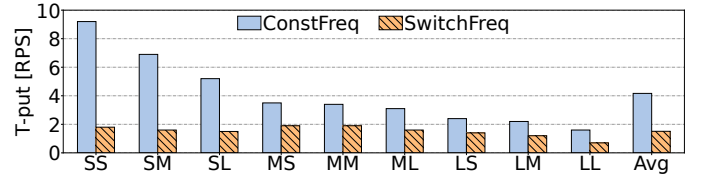


Fig. 3: Throughput for different request types with constant frequency (1980MHz) and with re-setting the frequency (to 1980MHz) on every iteration in the background.

around 50-80ms. In comparison, one decode iteration of the LLM inference process takes 20-30ms. Consequently, the time spent adjusting the GPU frequency can significantly impact the overall performance, potentially doubling the latency of individual inference steps. Figure 3 shows the throughput for different request types when constantly running at the highest frequency (1980 MHz) and when re-setting the frequency (to 1980 MHz) in the background on every LLM inference iteration. Due to the software overheads, the throughput of LLM inference system significantly drops. Therefore, optimizing or minimizing frequency changes during LLM inference is crucial for maintaining efficient and responsive performance.

Insight #3 Transitioning between LLM server configurations incurs significant overheads. For energy-efficiency, such overheads need to be minimized and considered when computing the energy/performance trade-offs.

IV. DYNAMOLLM: AN ENERGY MANAGEMENT FRAMEWORK FOR LLM INFERENCE CLUSTERS

We use the insights to design DynamoLLM, the first energy management framework for LLM inference environments. DynamoLLM seamlessly integrates with existing inference platforms, enabling LLM workloads to operate energy-efficiently and cost-effectively while meeting their performance SLOs. DynamoLLM has four key principles. First, it is *energy-optimized and SLO-aware*, leveraging model profiles to automatically select the most energy-efficient configuration for specific LLMs and inference workloads within their SLO requirements. Second, DynamoLLM fine-tunes configurations for heterogeneous LLM workloads by dividing cluster resources into *instance pools* tailored to specific request types. Third, DynamoLLM accommodates fluctuating LLM inference loads by *dynamically reconfiguring* the chosen organization. Finally, to ensure frequent and smooth reconfiguration, DynamoLLM *minimizes reconfiguration overheads*.

Architecture Figure 4 shows the DynamoLLM architecture. The system is organized hierarchically at the cluster, pool, and instance levels. At each level, the controllers tune their assigned configuration knob, and communicate their decisions with the controllers from the upper and lower levels. The controllers use energy-performance models generated in the profiling phase to determine the number of instances, model parallelization, and GPU frequency for an energy-optimized operation given the current system state. (1) *Cluster Manager* receives inference requests, predicts their type, and forwards them to the appropriate instance pool. Additionally, it peri-

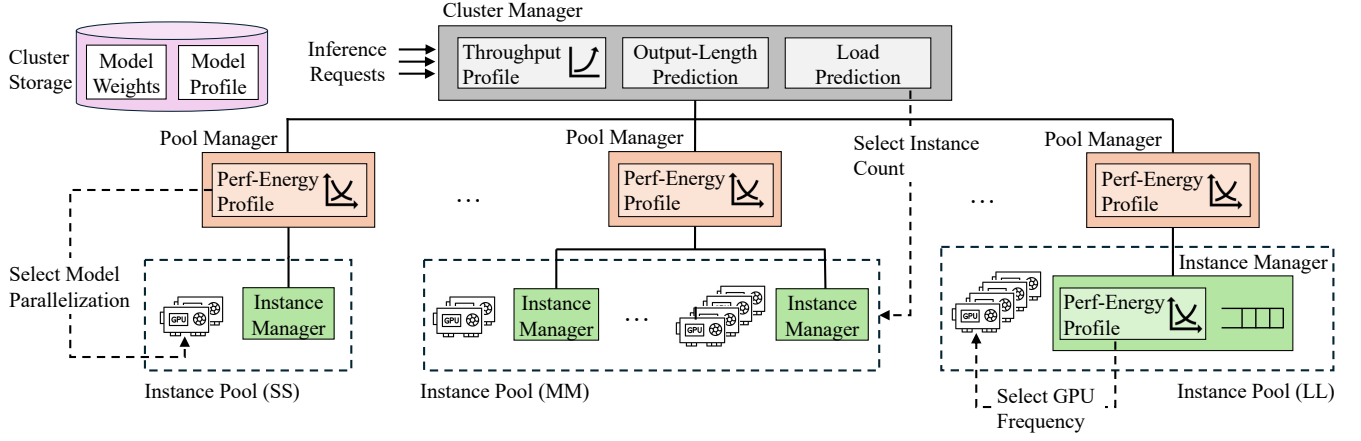


Fig. 4: DynamoLLM architecture: a hierarchy of controllers with cluster resources split into per request-type pools.

odically re-evaluates how many pools and how many model instances per pool are needed based on the system load. (2) *Pool Manager* schedules the requests to model instances in a manner that minimizes per-pool energy consumption. It also periodically checks if its instances need to be re-shared into a more energy-efficient parallelization. (3) *Instance Manager* schedules the requests to the inference engine and periodically checks if the instance’s GPU frequency needs to be adjusted.

A. Configuring Instances for Energy-Efficiency

Generating LLM profiles When deploying their service to DynamoLLM, users specify the LLM used by the service and the expected performance SLOs. Then, the system characterizes the model and generates its energy-performance profile. DynamoLLM profiles the model by running loads of different request lengths at different model parallelisms (TP2, TP4 and TP8) and GPU frequencies (800–1980MHz, with a step of 200MHz). The system profiles a few load levels, up to the maximum throughput, and then extrapolates the behavior for the loads in between the measured ones. The profiling result is a function that takes the load, request length, model parallelism and GPU frequency as inputs and outputs the expected energy consumption and TTFT/TBT latencies.

As many services may use the same model, DynamoLLM can reuse the profiles across services, minimizing the profiling overheads. Such profiles are stored in a global DynamoLLM repository, and then cached in a cluster-local storage when a given service is deployed in the cluster.

Selecting the energy-optimized configuration Given the current load and available resources, DynamoLLM uses the generated profiles to minimize energy consumption while staying within performance constraints. The system formulates this task as an optimization problem for the mixed integer linear programming (MILP) solver. The solver needs to output how many instances of each tensor parallelism (N^{TP_2} , N^{TP_4} and N^{TP_8}) are needed, at which frequency they should run (f^{TP_2} , f^{TP_4} and f^{TP_8}), and which load should be assigned to each instance (L^{TP_2} , L^{TP_4} and L^{TP_8}). We assume that all instances of a given parallelism run at the same frequency and receive fair-share amount of work.

The optimization target of the solver is to minimize the total energy consumption (E), while the constraints are: 1) the total number of GPUs used by all instance types does not exceed the assigned number of GPUs (N); 2) the load assigned to individual instances sums up to the total expected load (L); and 3) the expected performance of all instances with the assigned load is within the requirements (SLO). Functions $Energy^{TP_i, f_i}(L^{TP_i})$ and $Performance^{TP_i, f_i}(L^{TP_i})$ output the expected energy and performance, respectively, when running the load L^{TP_i} with TP_i parallelism at f_i GPU frequency. Then, the optimization task can be formulated as:

$$\begin{aligned}
 \min \quad & \left(\sum_i (N^{TP_i} \times Energy^{TP_i, f_i}(L^{TP_i})) \right) \quad \forall i \in \{2, 4, 8\} \\
 \text{s.t.} \quad & \sum_i i \times N^{TP_i} \leq N \\
 & \sum_i (N^{TP_i} \times L^{TP_i}) \geq L \\
 & Performance^{TP_i, f_i}(L^{TP_i}) \leq SLO
 \end{aligned} \tag{1}$$

This approach guarantees the energy optimal configuration. However, it introduces non-negligible overheads (*i.e.*, ~ 100 s of ms) due to the large search-space for the solver. Hence, it cannot be used to select the correct system configuration at fine-grain intervals (*e.g.*, every few seconds). Next we show how to break the task into a hierarchy of subtasks and use an approximation heuristic to reduce the computation complexity.

B. Hierarchical Control for Dynamic Load

DynamoLLM simplifies computations by assigning specific optimization tasks to individual controllers. Instead of searching for a globally optimal configuration, controllers set locally optimal values for individual knobs under the constraints imposed by the upper-level controllers and under the assumption that the lower-level controllers operate at the highest performance configuration. This allows the controllers to operate at varying time scales—from minutes for node adjustments down to seconds for frequency tuning. The different scales are needed as each operational change involves distinct overheads and energy-efficiency impacts.

Scale-out/in At every epoch (e.g., 30 minutes), the cluster manager computes the minimal number of nodes per pool that can support the load of a given request type. The manager uses a *load predictor* to forecast the incoming load, PL , for each request type based on its historic data (e.g., via lightweight load templates [62]). Moreover, the manager assumes that all instances will run at the highest-performance configuration (i.e., TP8 at 1980 MHz). Then, if the predicted peak of a given request type is PL , and the maximum load that a node can support for this request type is ML , the manager assigns $\lceil \frac{PL}{ML} \rceil$ nodes to that pool. By consolidating the work into a small number of nodes, the system tries to minimize the cost for the user and the idle energy of lightly-loaded GPUs.

Handling fragmentation: Allocating resources to handle the peak loads can cause resource underutilization. If overprovisioning accumulates across pools, the energy efficiency drops. To prevent such cases, DynamoLLM assigns one instance less to a given instance pool and moves the leftover load to the instance pool of the next larger request type. The cluster manager uses this information to forward a fraction of the load for a given request type to the appropriate larger instance pool during the next scheduling epoch (e.g., 30 minutes). In this way, only the instance pool for the largest requests can be overprovisioned minimizing the cluster-level fragmentation.

Shard-up/down At every epoch (e.g., every 5 minutes), the pool manager decides how to split the assigned N GPUs from the cluster manager into correct model parallelism (how many instances to create in the pool) and tensor parallelism (how many GPUs to use for each instance). The pool manager uses a simplified version of Equation (1) assuming that all instances run at the highest GPU frequency (i.e., 1980 MHz). Thus, the goal is to minimize the energy, while operating with the fixed number of GPUs running at the highest frequency, and controlling only the parallelism knob.

Accounting for the overheads: DynamoLLM stores the transitioning overheads (scale-out/in, shard-up/down) in an *Overhead Table*. This table is integrated with the controllers, so that when they calculate the energy benefits of new configurations, they can take into account the costs of reconfiguration. The controllers evaluate whether the energy savings gained from re-configuring justify the associated overheads and downtime.

Reducing downtime: The reconfiguration cannot occur simultaneously on all instances due to the risk of long downtime. Instead, DynamoLLM employs a staggered reconfiguration approach, where a subset of instances is reconfigured (e.g., re-sharded) at a time. This ensures that while some instances are undergoing reconfiguration, others remain operational to handle ongoing workloads. The system first reconfigures the instances that have higher potential energy savings.

Scale-up/down Finally, at every epoch (e.g., 5s), the instance manager fine tunes the GPU frequency for further energy savings given the assigned model parallelism. The instance manager uses the performance profile to first filter-out frequencies that violate the SLO under the current load. Then, it uses the energy profile to pick an acceptable frequency that minimizes the energy consumption.

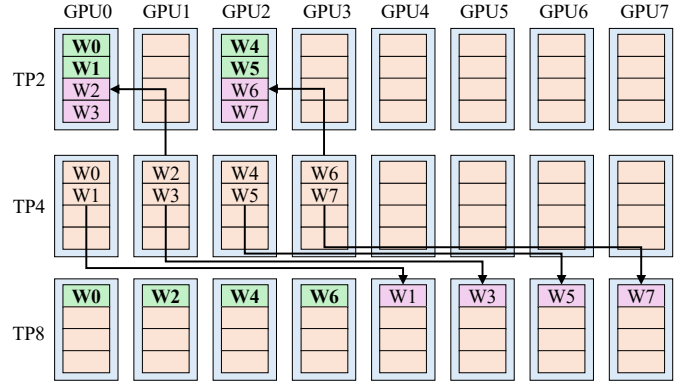


Fig. 5: Example of re-sharding a TP4 model to TP2/TP8.

C. Reduced Overheads for Smooth Reconfiguration

To enable frequent reconfiguration, DynamoLLM proposes a set of techniques to minimize the overheads of (1) *scaling-in/out* the number of LLM inference servers, (2) *sharding-up/down* the parallelism of a given instance, and (3) *scaling-up/down* the GPU frequency of a given instance.

Scaling in/out inference servers DynamoLLM reduces the overheads of creating a new server instance by implementing several strategies. First, it keeps the model weights cached locally within the cluster (shown in Figure 4) avoiding the need to fetch them from a global repository. Second, it starts VMs from a snapshot with the entire state of the inference engine already initialized, reducing the boot-up time. This snapshot includes pre-loaded libraries, GPU drivers and inference engine configurations. Third, it proactively creates new VMs in the background, outside of the critical path of active workload handling. Specifically, DynamoLLM predicts the peak load for the next scheduling epoch and starts the extra VMs before the epoch starts. By having these VMs ready to go, DynamoLLM can seamlessly offload a fraction of the load to new instances without any noticeable latency impact.

Sharding up/down an instance To reduce the re-sharding overheads, DynamoLLM optimizes the distribution of weights across GPUs. We propose two techniques to minimize the data transfers and latency of individual transfers. First, the system develops a graph matching algorithm that maximizes the amount of weights that remain stationary in their current GPUs. The algorithm takes current weight distribution and desired tensor parallelism as inputs, and outputs the source and destination GPUs and fraction of weights to be transferred between each source-destination pair. Specifically, the algorithm constructs a bipartite graph where nodes represent GPUs in the current and next configurations. Edges between nodes represent potential transfers, weighted by the amount of data to be transferred. Then, it applies a maximum weight matching algorithm to find the optimal transfer plan that minimizes the total weight of the edges (i.e., minimizes the amount of data transferred). Second, to reduce the transfer latency, the system uses inter-GPU direct transfers via NVLink, allowing them to send fractions of their weights in parallel to other GPUs without any host intervention.

Figure 5 shows an example of re-sharding from TP4 to TP2 and TP8. Consider first the case when going from a lower to a higher-level parallelism (TP4→TP8). In the initial state (TP4), GPUs 0-3 hold a quarter of the model weights each. In the final state (TP8), all GPUs need to hold an eighth of the model weight. Thus, the first four GPUs already have their required state, and they only need to send half of their weights to the remaining four GPUs. The four transfers (e.g., GPU0→GPU4, GPU1→GPU5,...) happen in parallel. Therefore, the re-sharding requires the time to send 1/8 of the model weights via NVLink (around 50ms in our setup).

Consider now the case when going from a higher to a lower parallelism (TP4→TP2). In the final state (TP2), two GPUs need to hold a half of the weights each. As each GPU initially holds a quarter of the weights, we merge the weights from two GPUs to a single one: GPU1 sends weights W2/W3 to GPU0, and GPU3 sends weights W6/W7 to GPU2. As these two transfers happen in parallel, the total re-sharding time is the time to send 1/4 of the model weights (around 100ms in our setup). Table VI shows the re-sharding overheads with different source/destination pairs with our optimized approach. Some configurations quickly switch to other configurations (transition time 0), other changes incur larger overheads (transition time 4T, where T is the time to send 1/8 of the weights).

Moreover, on some transitions, the instance continues serving the requests with the same throughput. This is the case when scaling from a smaller to a larger tensor parallelism (e.g., TP4→TP8). The old instance sends a fraction of the weights to the new instance, without increasing its memory footprint. During other transitions, the current instance needs to operate under lower throughput. This is the case when the instance scales from a larger to a smaller tensor parallelism (e.g., TP8→TP4). Some GPUs used by the old instance accept extra weights, reducing their memory capacity to serve new requests. In general, whenever the GPU memory required to hold model weights increases, the throughput decreases due to the lower memory capacity for the incoming requests.

Finally, after the weights are sent to the correct memories, the inference engine needs to synchronize the GPUs that run the new instance. State-of-the-art engines, such as vLLM [26], perform this operation in a few 100s of milliseconds to a few seconds. During this period the instance cannot receive any load, causing noticeable downtime. To reduce the downtime, DynamoLLM allows the old instance to process the requests while the new instance is going through the synchronization process. Only when the new instance comes online, the old instance is removed. This is possible only when the sum of the memory from the old and new instances is below the GPU’s memory capacity. When the sum exceeds the memory capacity (e.g., TP2 and TP4 with 70B parameters), the old instance needs to be shutdown before the new instance is created. Overall, different transitions incur different overheads and instance downtime requiring a fraction of the load to be shifted to another instance. DynamoLLM minimizes the overheads, and considers their impact on the overall efficiency when deciding whether to re-configure an instance.

Src/Dst	TP2	4TP2	TP4	TP2+TP4	2TP4	TP8
TP2	0	4T	2T	2T	2T	T
4TP2	0	0	0	0	0	0
TP4	2T	2T	0	2T	2T	T
TP2+TP4	0	2T	0	0	T	T
2TP4	T	T	0	T	0	0
TP8	T	T	T	T	T	0

TABLE VI: Overhead of transferring model weights on a re-sharding. T is the time unit to move 1/8 of the model (e.g., with 300GB/s NVLink bandwidth on NVIDIA DGX H100 [44] and Llama2-70B model [67], $T = 50ms$).

Scaling up/down the frequency The overheads of changing the GPU frequency are minimized by keeping the NVIDIA System Management Interface (`nvidia-smi`) monitor program directly loaded into memory. This eliminates the need to reload the program every time a frequency adjustment is required. Moreover, by running the controller in privileged mode, DynamoLLM avoids the overhead associated with OS-user interactions, allowing for rapid frequency adjustments.

D. Predictive Scheduling for Request Heterogeneity

To map the heterogeneity of requests to the heterogeneous instance pools, the cluster controller in DynamoLLM uses an *output-length predictor* to anticipate the request type and steer requests to the correct instance pool. The predictor acts as a proxy model that takes input prompt and classifies the output as short, medium or long. Based on the predicted output length and known input length, the cluster manager forwards the request to the pool manager being in charge for a given request type. If the instance pool is currently overloaded, the cluster manager forwards the request to the next available pool for a larger request type. Once the request arrives to the correct pool, the pool manager needs to pick an instance from the pool. Specifically, the manager uses the generated models from the profiling step to predict energy and response times of each instance after potentially adding a new request to that instance. Then, it chooses the instance that minimizes total energy while staying within per-instance throughput determined by the SLO.

Handling mis-predictions If the system over-estimates a request length, the request gets routed to a higher-performance pool. Hence, it runs with sub-optimal energy, but its latency remains unaffected. Conversely, if a request length is under-estimated, the request is placed to a lower-performance pool, potentially missing its SLOs. Similarly, load mis-predictions can result in insufficient resources for a given pool during request bursts. In both cases, due to some mis-predictions, the system needs to react to the created *emergency event*.

When an instance manager detects that its queue is building up, indicating that the rate of request processing is lower than the rate of request arrival, it triggers an emergency event. First, the instance manager tries to re-order the requests in its queue and prioritizes those requests that are about to miss their deadline. Second, if some requests will miss their deadlines even after the reordering, the instance manager ramps up the frequency of its GPUs to the maximum value. Third, if the

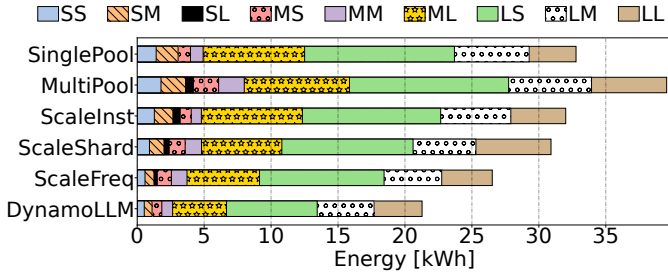


Fig. 6: Energy consumption with the six evaluated systems running open-source Llama2-70B model [67] with 1-hour open-source production traces [50].

backlog persists or worsens, the instance manager re-steers some requests that have not started their execution. A subset of requests is moved to another instance within the pool via the pool manager. Finally, if all the attempts are insufficient, the instance manager resorts to more drastic measures. One such measure involves squashing requests that have been waiting for processing beyond a specified threshold. This action signals users to retry their requests, allowing the frontend system to redirect these requests to alternative instance pools or retry them later when system load has stabilized.

E. DynamoLLM Implementation

We build DynamoLLM on top of vLLM [26], a state-of-the-art LLM inference platform. However, DynamoLLM’s modularity allows it to be integrated with other platforms, *e.g.*, TensorRT-LLM [47], without modifications. We implement controllers as lightweight gRPC servers with low memory and compute requirements. Cluster and pool managers are hosted in a dedicated VM for robust management. Instance managers are co-located with the LLM inference engine instances for low communication overheads. For output length prediction, we leverage a BERT-based proxy model [55], which provides accurate and efficient classification of incoming requests. For load prediction, we use a template-based approach that uses historical data to model load patterns over a week [62]. The pool manager employs Python’s PuLP library [53] for solving MILP. DynamoLLM models energy and performance using the *interpId* function from the SciPy [54] Python library.

V. EVALUATION

A. Evaluation Setup

We run our experiments on servers with 8 H100 GPUs [44]. We show the results for Llama2-70B [67], but other models (*i.e.*, Mixtral [38], Falcon [66], BLOOM [59]) follow the same trends. We set the load using production-level traces: 1 hour open-source traces [50] and 1-day and 1-week traces for *Coding* and *Conversation* from our fleet. We compare DynamoLLM to five systems. *SinglePool* (a state-of-the-practice baseline) schedules all the requests to the common pool of instances running with TP8 at the highest GPU frequency. *MultiPool* separates LLM instances in multiple per-request-type pools. *ScaleInst*, *ScaleShard*, and *ScaleFreq* additionally

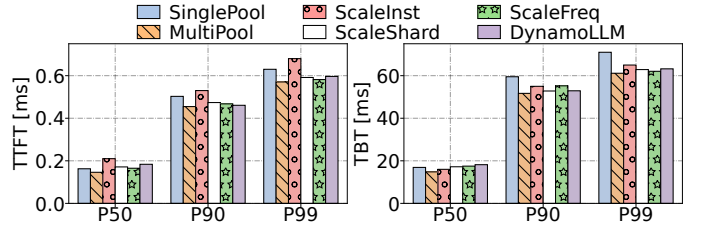


Fig. 7: Summary of the latencies for each of the systems running open-source Llama2-70B model [67] with 1-hour open-source production traces [50].

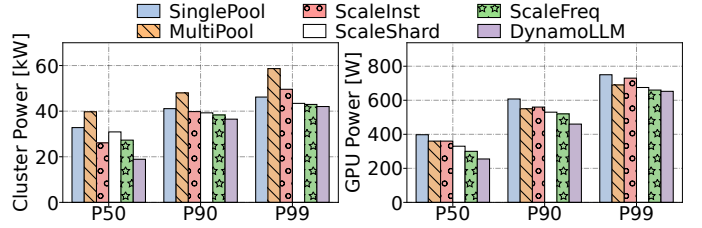


Fig. 8: Summary of the power for each of the systems running open-source Llama2-70B model [67] with 1-hour open-source production traces [50].

scale the number of instances in the pool, model parallelism, or GPU frequency according to the current load, respectively.

B. Cluster-Level Experiments

We first evaluate the system on a cluster of GPU servers using the 1h open source production traces for the *Conversation* service [50]. We provision the baselines with 12 H100 servers to handle the peak load, while DynamoLLM scales the number of servers according to the current load.

Energy Figure 6 shows the energy consumption of the cluster for the experiment. MultiPool increases the energy consumption by 20% over SinglePool, because it allocates a larger number of resources while always operating at the highest-performance configuration. Meanwhile, ScaleInst, ScaleShard, ScaleFreq and DynamoLLM reduce the energy consumption by 4.1%, 7%, 19% and 35%, respectively. ScaleInst/Shard/Freq reduce the energy by configuring one knob but leave substantial space for further savings. Finally, DynamoLLM synchronously scales multiple knobs to achieve the lowest energy consumption. We further breakdown the total energy per request type. Figure 6 shows that longer requests (*e.g.*, LL) and highly-popular requests (*e.g.*, ML) consume disproportionately more energy than the other types.

Latency Figure 7 shows the TTFT/TBT latencies for each system. By separating request types into different resource pools, MultiPool removes the head-of-line blocking effect and reduces the latencies over SinglePool. Similarly, ScaleShard and ScaleFreq, and DynamoLLM reduce the tail latency. However, these systems slightly increase the P50 latency by operating in lower-performance modes when there is available SLO slack. On the other hand, ScaleInst increases the tail latency due to the large overheads of creating a new inference server on the critical path of users’ load. Overall, DynamoLLM reduces

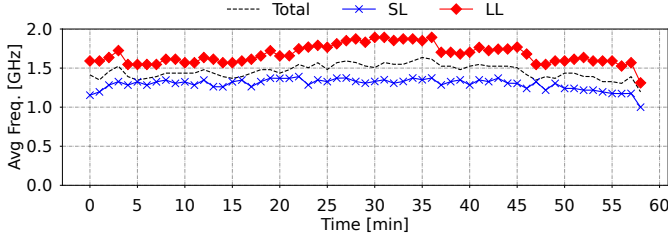


Fig. 9: GPU Frequency over one hour for DynamoLLM running open-source Llama2-70B model [67] with 1-hour open-source production traces [50].

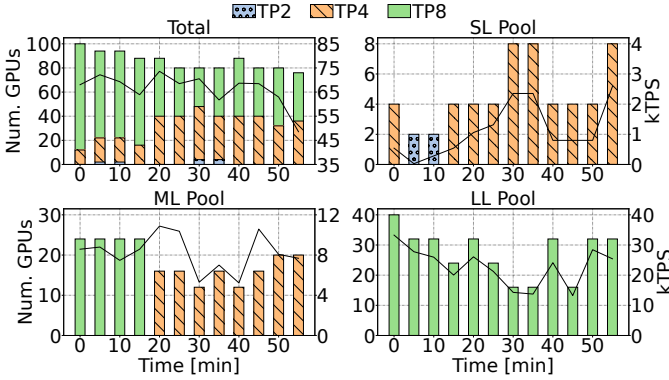


Fig. 10: Number of GPUs used for each sharding configuration (TP2, TP4 or TP8) over time running open-source Llama2-70B model [67] with 1-hour open-source production traces [50].

the P99 TTFT and TBT latencies by 5.3% and 11.1% over SinglePool, respectively, while it increases the P50 TTFT and TBT latencies by 11.4% and 7.6%, respectively.

Power Figure 8 shows the power consumption across the systems for the cluster (right figure) and on average per-GPU (left figure). Due to operating in energy-efficient modes, DynamoLLM effectively reduces both cluster and per-GPU power. DynamoLLM reduces the P50 and P99 power consumption over the baseline by 43% and 9%, respectively.

Frequency changes Figure 9 shows the average GPU frequency over time for (1) the whole cluster, (2) the pool serving short requests, and (3) the pool serving long requests. Average frequency is always significantly lower than the maximum allowed frequency (1980 MHz) that is used by the baseline. DynamoLLM effectively accommodates different request types by operating their pools at different frequencies.

Sharding changes Figure 10 shows the number of GPUs used for different model parallelisms (TP2, TP4 and TP8) for the whole cluster and for the individual pools (SL, ML and LL). The figure also shows the load over time that a given pool experiences. Different pools operate with different model parallelisms and DynamoLLM efficiently changes the parallelism as the load changes.

C. Sensitivity Studies

Sensitivity to predictor accuracy We analyze how the accuracy of the prediction models affects the overall system

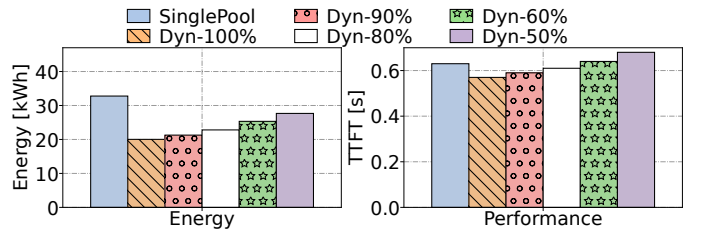


Fig. 11: Energy and performance with different accuracy running open-source Llama2-70B model [67] with 1-hour open-source production traces [50].

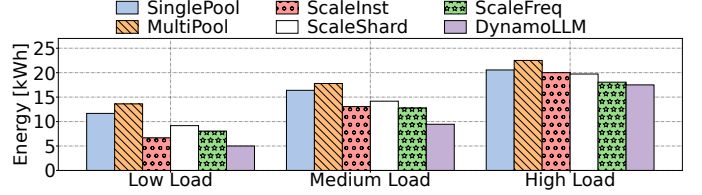


Fig. 12: Energy with different levels of load running open-source Llama2-70B model [67].

efficiency. We introduce bounded errors for the output length misclassification and measure the energy consumption with medium load. Figure 11 shows that the impact of the predictor accuracy is modest for both energy and performance. Compared to an environment with no error, an environment with an 40% error increases the energy consumption by 13% and TTFT by 7.3%. The reason for robustness to prediction errors is that DynamoLLM can promptly detect mis-predictions and re-configures the knobs accordingly.

Sensitivity to load We evaluate DynamoLLM with different system loads. We generate Low, Medium, and High loads with a Poisson distribution for request inter-arrival times. Figure 12 shows the energy consumption of the five evaluated systems with different load levels. With Low, Medium, and High load, DynamoLLM reduces the energy of SinglePool baseline by 51%, 40%, and 23.4%, respectively. As the load increases, the energy savings of DynamoLLM reduce, because the system more frequently needs to operate at higher frequencies with higher levels of model parallelism.

Sensitivity to number of pools Figure 13 shows the energy consumption and performance (TTFT) of DynamoLLM with different number of request pools. Recall that our chosen design has 9 pools. By adding too many pools (12 or 16), the system gets fragmented, and the idle energy of GPUs results in the overall energy increase. Reducing the number of pools (2 or 4) prevents the system from fine tuning the frequency and the model parallelism for specific request types. The performance improves by adding moderately more pools because it helps remove head of the line blocking and introduces more resources for execution.

D. Long Cluster-Level Experiments

We run longer experiments by running the 1-day traces for the *Conversation* service. The trace covers all invocations for

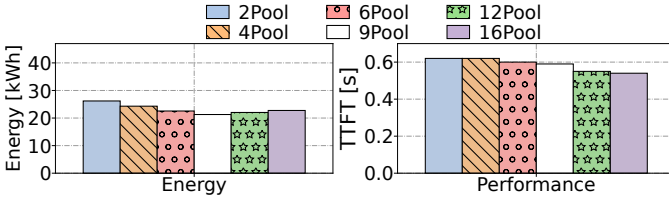


Fig. 13: Energy and performance with different number of pools (or request types) running open-source Llama2-70B model [67] with 1-hour open-source production traces [50].

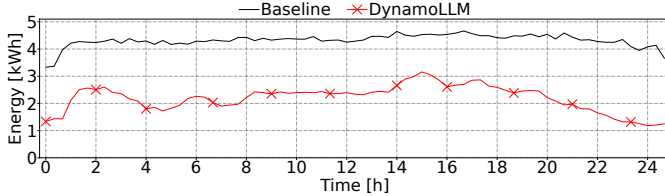


Fig. 15: Energy consumption of SinglePool and DynamoLLM with 1-day production traces running open-source Llama2-70B model [67].

a subset of the service’s instances during a typical work day. We run the experiment for 24-hours on 11 H100 servers with SinglePool and scale the number of instances based on the load in DynamoLLM. Figure 15 shows the energy consumption over 5-minute intervals for the two systems. DynamoLLM reduces the energy consumption over the baseline during peak hours (when dynamic power dominates), and during the low utilization times (when idle power dominates). Over the whole day, DynamoLLM reduces the energy consumption by 42%.

E. Large-Scale Simulations

To generalize our insights to large-scale, we develop a discrete-time simulator that simulates the energy consumption of different systems using production traces. Figure 14 shows the normalized energy consumption for the five evaluated systems using 1-week traces for *Conversation* and *Coding* services. DynamoLLM significantly reduces the energy consumption for both types of services. DynamoLLM operates in higher energy-efficient modes for the *Conversation* service due to its typically shorter input lengths (ML dominant request type). On the other hand, the *Coding* service has deep valleys during the night and weekends. Thus, DynamoLLM exploits the periods of low load to save energy. DynamoLLM reduces the energy consumption over the baseline by 47% and 56% for the *Conversation* and *Coding* services, respectively.

F. Cost and Carbon Emission

User cost DynamoLLM reduces the operational cost for users by minimizing the number of GPUs and optimizing their energy efficiency. The number of GPU servers for the week-long experiments reduces from 40 to 24.6 on average (38.5% cost reduction). Using the current GPU VM pricing [8], this saves \$1362.7/hour. By reducing the energy consumption, DynamoLLM reduces the associated energy costs by up to

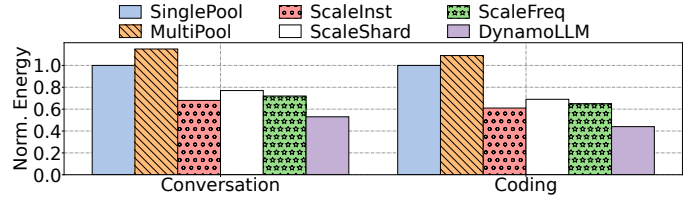


Fig. 14: Normalized energy consumption for the six evaluated systems with the week-long production traces running open-source Llama2-70B model [67].

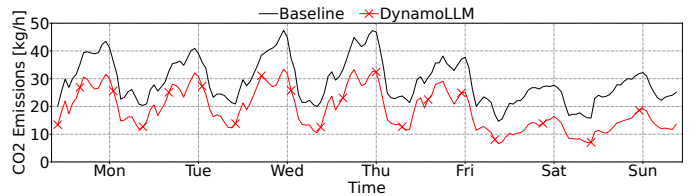


Fig. 16: Carbon emissions over time for the week-long conversation traces with DynamoLLM and SinglePool baseline running open-source Llama2-70B model [67].

56%. As energy costs [28] are currently substantially lower than GPU costs, this translates to only \$4.4/hour savings.

Carbon emissions The energy consumption translates into the amount of operational CO_2 emissions. We use the traces of carbon intensity [2] for a week-long period from multiple grids and map the carbon intensity to the energy consumption over time for the SinglePool baseline and DynamoLLM. Figure 16 shows the operational carbon emissions over time for the two systems for CAISO [1]. SinglePool and DynamoLLM consume 5t and 3.1t/week of CO_2 . These substantial savings (38%) make a step towards sustainable LLM environments.

VI. RELATED WORK

Cluster resource and power management A rich body of work seeks to improve resource efficiency under the SLO constraints through resource management for a wide range of latency sensitive workloads, such as microservices [76] and DL workloads, through effective resource sharing [6], [43], [51], dynamic allocation [71], and hardware reconfiguration [24]. Others focus on approaches that enable safe power management and oversubscription [16], [29], [49] leveraging workload characteristics [25], [75] and system state [62].

Energy-efficient workloads Prior works focused on energy-efficiency for CPU workloads [15], [31], [43], [61], and researchers started exploring unique energy properties of GPU workloads [58], [65], [74]. Recent schemes build on top and manage energy and power consumption for DNN inference and training [69], [70], [72] through frequency scaling [20], [23], [40], [41], [64], [77], [82], autoscaling [22], and resource partitioning and mapping [18], [64]. We show that improving energy efficiency for LLM inference necessitates a comprehensive view of all available knobs. DynamoLLM is holistic framework that dynamically reconfigures all the knobs considering the diversity and dynamism of requests and loads.

Efficient LLM inference serving Recent works propose approaches to improve LLM inference efficiency through heterogeneous resources [4] and platforms [35], memory and key-value cache management [9], [26], and node- and cluster-level scheduling [3], [30], [42], [48], [50], [73], [81]. While these studies focus on improving throughput or latency, we show that optimizing energy efficiency for LLM inference exhibits distinct trade-offs between performance, energy consumption, and overheads and thus requires a comprehensive framework.

VII. CONCLUSION

We present DynamoLLM, the first energy-management framework for LLM inference clusters. DynamoLLM exploits heterogeneity in inference compute properties and fluctuations in inference workloads to save energy. The system automatically and dynamically configures the energy-optimal organization of the cluster (number of instances, model parallelism and GPU frequency) while performing under performance guarantees. DynamoLLM reduces energy, carbon emissions and cost to the customer by 53%, 38% and 61%, respectively.

REFERENCES

- [1] “CAISO carbon emission,” <https://www.caiso.com/today-s-outlook/emissions>, 2024.
- [2] “WattTime,” <https://watttime.org>, 2024.
- [3] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, “SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills,” 2023.
- [4] K. Alizadeh, I. Mirzadeh, D. Belenko, K. Khatamifard, M. Cho, C. C. D. Mundo, M. Rastegari, and M. Farajtabar, “LLM in a flash: Efficient Large Language Model Inference with Limited Memory,” 2024.
- [5] J. Bowne, “Using large language models in learning and teaching,” <https://biomedsciences.unimelb.edu.au/study/dlh/assets/documents/large-language-models-in-education/llms-in-education>, 2024.
- [6] S. Chen, C. Delimitrou, and J. F. Martínez, “PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’19)*, 2019.
- [7] S. Chen, A. Jin, C. Delimitrou, and J. F. Martínez, “ReTail: Opting for Learning Simplicity to Enable QoS-Aware Power Management in the Cloud,” in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA ’22)*, 2022.
- [8] Cloud Price, “Standard_ND96is_H100_v5,” https://cloudprice.net/vm/Standard_ND96is_H100_v5, 2024.
- [9] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness,” 2022.
- [10] C. Delimitrou and C. Kozyrakis, “Amdahl’s Law for Tail Latency,” *Commun. ACM*, vol. 61, no. 8, jul 2018.
- [11] D. Ding, A. Mallick, C. Wang, R. Sim, S. Mukherjee, V. Ruehle, L. V. S. Lakshmanan, and A. Awadallah, “Hybrid LLM: Cost-Efficient and Quality-Aware Query Routing,” in *Proceedings of the International Conference on Learning Representations (ICLR ’24)*, 2024.
- [12] A. Faiz, S. Kaneda, R. Wang, R. Osi, P. Sharma, F. Chen, and L. Jiang, “LLMCarbon: Modeling the end-to-end Carbon Footprint of Large Language Models,” *arXiv preprint arXiv:2309.14393*, 2024.
- [13] GitHub, “The world’s most widely adopted ai developer tool,” <https://github.com/features/copilot>, 2024.
- [14] M. Halpern, B. Boroujerdian, T. Mummert, E. Duesterwald, and V. Janapa Reddi, “One Size Does Not Fit All: Quantifying and Exposing the Accuracy-Latency Trade-Off in Machine Learning Cloud Service APIs via Tolerance Tiers,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS ’19)*, 2019.
- [15] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, “Exploiting Heterogeneity for Tail Latency and Energy Efficiency,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’17)*, 2017.
- [16] C.-H. Hsu, Q. Deng, J. Mars, and L. Tang, “SmoothOperator: Reducing Power Fragmentation and Improving Power Utilization in Large-Scale Datacenters,” in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’18)*, 2018.
- [17] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, “Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting,” in *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA ’15)*, 2015.
- [18] A. Jahanshahi, M. Rezvani, and D. Wong, “WattWiser: Power and Resource-Efficient Scheduling for Multi-Model Multi-GPU Inference Servers,” in *Proceedings of the 14th International Green and Sustainable Computing Conference (IGSC ’23)*, 2023.
- [19] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei, “ S^3 : Increasing GPU utilization during generative inference for higher throughput,” in *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS ’23)*, 2023.
- [20] A. K. Kakolyris, D. Masouros, S. Xydis, and D. Soudris, “SLO-aware GPU DVFS for Energy-efficient LLM Inference Serving,” *IEEE Computer Architecture Letters*, 2024.
- [21] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, “Rubik: Fast analytical power management for latency-critical systems,” in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’15)*, 2015.
- [22] Y. G. Kim and C.-J. Wu, “AutoScale: Energy Efficiency Optimization for Stochastic Edge Inference Using Reinforcement Learning,” in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’20)*, 2020.
- [23] T. Komoda, S. Hayashi, T. Nakada, S. Miwa, and H. Nakamura, “Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping,” in *Proceedings of the IEEE 31st International Conference on Computer Design (ICCD ’13)*, 2013.
- [24] N. Kulkarni, G. Gonzalez-Pumariaga, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi, “CuttleSys: Data-Driven Resource Management for Interactive Services on Reconfigurable Multicores,” in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’20)*, 2020.
- [25] A. G. Kumbhare, R. Azimi, I. Manousakis, A. Bonde, F. Frujeri, N. Mahalingam, P. A. Misra, S. A. Javadi, B. Schroeder, M. Fontoura, and R. Bianchini, “Prediction-Based Power Oversubscription in Cloud Platforms,” in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC ’21)*, 2021.
- [26] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, “Efficient Memory Management for Large Language Model Serving with PagedAttention,” in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP ’23)*, 2023.
- [27] M. Lammertyn, “60+ ChatGPT Statistics And Facts You Need to Know in 2024,” <https://blog.invgate.com/chatgpt-statistics>, 2024.
- [28] LCG Consulting, “Energy Online: ERCOT Real Time Price,” https://energyonline.com/Data/GenericData.aspx?DataId=4&ERCOT__Real-time_Price, 2024.
- [29] S. Li, X. Wang, X. Zhang, V. Kontorinis, S. Kodakara, D. Lo, and P. Ranganathan, “Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale,” in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’20)*, 2020.
- [30] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica, “AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving,” in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’23)*, 2023.
- [31] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, “Towards energy proportionality for large-scale latency-critical workloads,” in *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA ’14)*, 2014.
- [32] Meta, “Llama2-13B,” <https://huggingface.co/meta-llama/Llama-2-13b>, 2024.
- [33] Meta, “Llama2-70B,” <https://huggingface.co/meta-llama/Llama-2-70b>, 2024.
- [34] Meta, “Llama3-70B,” <https://huggingface.co/meta-llama/Meta-Llama-3-70B-Instruct>, 2024.

- [35] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "SpotServe: Serving Generative Large Language Models on Preemptible Instances," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*, 2024.
- [36] Microsoft Azure, "ND H100 v5-series," <https://learn.microsoft.com/en-us/azure/virtual-machines/nd-h100-v5-series>, 2024.
- [37] A. Mirhosseini and T. Wenisch, "μSteal: A Theory-Backed Framework for Preemptive Work and Resource Stealing in Mixed-Criticality Microservices," in *Proceedings of the ACM International Conference on Supercomputing (ICS '21)*, 2021.
- [38] Mistral AI, "The Mixtral-8x22B Large Language Model," <https://huggingface.co/mistralai/Mixtral-8x22B-Instruct-v0.1>, 2024.
- [39] Mistral AI, "The Mixtral-8x7B Large Language Model," <https://huggingface.co/mistralai/Mixtral-8x7B-Instruct-v0.1>, 2024.
- [40] S. M. Nabavinejad, S. Reda, and M. Ebrahimi, "BatchSizer: Power-Performance Trade-off for DNN Inference," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASP-DAC '21)*, 2021.
- [41] S. M. Nabavinejad, S. Reda, and M. Ebrahimi, "Coordinated batching and dvfs for dnn inference on gpu accelerators," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, 2022.
- [42] K. K. W. Ng, H. M. Demoulin, and V. Liu, "Paella: Low-latency Model Serving with Software-defined GPU Scheduling," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, 2023.
- [43] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '20)*, 2020.
- [44] NVIDIA, "DGX H100: AI for Enterprise," <https://www.nvidia.com/en-us/data-center/dgx-h100/>, 2024.
- [45] NVIDIA, "NVLink and NVLink Switch," <https://www.nvidia.com/en-us/data-center/nvlink/>, 2024.
- [46] NVIDIA, "System Management Interface SMI," <https://developer.nvidia.com/system-management-interface>, 2024.
- [47] NVIDIA, "TensorRT-LLM's Documentation," <https://nvidia.github.io/TensorRT-LLM/>, 2024.
- [48] H. Oh, K. Kim, J. Kim, S. Kim, J. Lee, D.-s. Chang, and J. Seo, "ExeGPT: Constraint-Aware Resource Scheduling for LLM Inference," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*, 2024.
- [49] P. Patel, E. Choukse, C. Zhang, I. Goiri, B. Warriar, N. Mahalingam, and R. Bianchini, "Characterizing Power Management Opportunities for LLMs in the Cloud," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*, 2024.
- [50] P. Patel, E. Choukse, C. Zhang, A. Shah, I. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative LLM inference using phase splitting," in *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*, 2024.
- [51] T. Patel and D. Tiwari, "CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '20)*, 2020.
- [52] C. Peng, X. Yang, A. Chen, K. Smith, N. PourNejatian, A. Costa, C. Martin, M. Flores, Y. Zhang, T. Magoc, G. Lipori, M. Duane, N. Ospina, M. Ahmed, W. Hogan, E. Shenkman, Y. Guo, J. Bian, and Y. Wu, "A study of generative large language model for medical research and healthcare," *npj Digital Medicine*, 2023.
- [53] Python PuLP, "Optimization with PuLP," <https://coin-or.github.io/pulp/>, 2024.
- [54] Python SciPy, "SciPy Library," <https://scipy.org/>, 2024.
- [55] H. Qiu, W. Mao, A. Patke, S. Cui, S. Jha, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "Efficient Interactive LLM Serving with Proxy Model-based Sequence Length Prediction," in *The 5th International Workshop on Cloud Intelligence / AIOps at ASPLOS 2024*, 2024.
- [56] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [57] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated Model-less Inference Serving," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.
- [58] S. Samsi, D. Zhao, J. McDonald, B. Li, A. Michaleas, M. Jones, W. Bergeron, J. Kepner, D. Tiwari, and V. Gadepally, "From words to watts: Benchmarking the energy costs of large language model inference," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2023, pp. 1–9.
- [59] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé *et al.*, "BLOOM: A 176b-parameter open-access multilingual language model," *arXiv preprint arXiv:2211.05100*, 2022.
- [60] J. Stojkovic, E. Choukse, C. Zhang, I. Goiri, and J. Torrellas, "Towards Greener LLMs: Bringing Energy-Efficiency to the Forefront of LLM Inference," *arXiv e-prints*, p. arXiv:2403.20306, Mar. 2024.
- [61] J. Stojkovic, N. Iliakopoulou, T. Xu, H. Franke, and J. Torrellas, "EcoFaaS: Rethinking the Design of Serverless Environments for Energy Efficiency," in *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*, 2024.
- [62] J. Stojkovic, P. Misra, I. Goiri, S. Whitlock, E. Choukse, M. Das, C. Bansal, J. Lee, Z. Sun, H. Qiu, R. Zimmermann, S. Samal, B. Warriar, A. Raniwala, and R. Bianchini, "SmartOClock: Workload- and Risk-Aware Overclocking in the Cloud," in *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*, 2024.
- [63] K. Talamadupula, "A Guide to LLM Inference Performance Monitoring," <https://syml.ai/developers/blog/a-guide-to-llm-inference-performance-monitoring/>, 2024.
- [64] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E.-Y. Yang, M. Donato, V. Sanh, P. Whatmough, A. M. Rush, D. Brooks, and G.-Y. Wei, "EdgeBERT: Sentence-Level Energy Optimizations for Latency-Aware Multi-Task NLP Inference," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*, 2021.
- [65] Z. Tang, Y. Wang, Q. Wang, and X. Chu, "The Impact of GPU DVFS on the Energy and Performance of Deep Learning: an Empirical Study," in *Proceedings of the Tenth ACM International Conference on Future Energy Systems (e-Energy '19)*, 2019.
- [66] Technology Innovation Institute (TII), "Falcon-180B," 2024.
- [67] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.
- [68] T. Varshney, "Build an llm-powered data agent for data analysis," <https://developer.nvidia.com/blog/build-an-llm-powered-data-agent-for-data-analysis/>, 2024.
- [69] C. Wan, M. Santriagi, E. Rogers, H. Hoffmann, M. Maire, and S. Lu, "ALERT: Accurate learning for energy and timeliness," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '20)*, 2020.
- [70] F. Wang, W. Zhang, S. Lai, M. Hao, and Z. Wang, "Dynamic GPU Energy Optimization for Machine Learning Training Workloads," *IEEE Transactions on Parallel and Distributed Systems*, 2022.
- [71] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "AntMan: Dynamic scaling on GPU clusters for deep learning," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.
- [72] J. You, J.-W. Chung, and M. Chowdhury, "Zeus: Understanding and optimizing GPU energy consumption of DNN training," in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*, 2023.
- [73] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, 2022.
- [74] J. Yu, J. Kim, and E. Seo, "Know Your Enemy To Save Cloud Energy: Energy-Performance Characterization of Machine Learning Serving," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.
- [75] C. Zhang, A. G. Kumbhare, I. Manousakis, D. Zhang, P. A. Misra, R. Assis, K. Woolcock, N. Mahalingam, B. Warriar, D. Gauthier, L. Kunnath, S. Solomon, O. Morales, M. Fontoura, and R. Bianchini, "Flex: High-Availability Datacenters with Zero Reserved Power," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA '21)*, 2021.
- [76] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.

- [77] Y. Zhang, Q. Wang, Z. Lin, P. Xu, and B. Wang, "Improving gpu energy efficiency through an application-transparent frequency scaling policy with performance assurance," in *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*, 2024.
- [78] Y. Z. Zhao, D. W. Wu, and J. Wang, "ALISA: Accelerating Large Language Model Inference via Sparsity-Aware KV Caching," in *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*, 2024.
- [79] Z. Zheng, X. Ren, F. Xue, Y. Luo, X. Jiang, and Y. You, "Response Length Perception and Sequence Scheduling: An LLM-Empowered LLM Inference Pipeline," in *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS '23)*, 2023.
- [80] L. Zhou, L. N. Bhuyan, and K. K. Ramakrishnan, "Gemini: Learning to Manage CPU Power for Latency-Critical Search Engines," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*, 2020.
- [81] Z. Zhou, X. Wei, J. Zhang, and G. Sun, "PetS: A unified framework for Parameter-Efficient transformers serving," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '22)*, 2022.
- [82] P. Zou, A. Li, K. Barker, and R. Ge, "Indicator-Directed Dynamic Power Management for Iterative Workloads on GPU-Accelerated Systems," in *Proceedings of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID '20)*, 2020.