LLM-Pilot: Characterize and Optimize Performance of your LLM Inference Services

Malgorzata Lazuka
IBM Research, ETH Zurich
Zurich, Switzerland
mal@zurich.ibm.com

Andreea Anghel
IBM Research
Zurich, Switzerland
aan@zurich.ibm.com

Thomas Parnell
IBM Research
Zurich, Switzerland
tpa@zurich.ibm.com

***Abstract*—As Large Language Models (LLMs) are rapidly growing in popularity, LLM inference services must be able to serve requests from thousands of users while satisfying performance requirements. The performance of an LLM inference service is largely determined by the hardware onto which it is deployed, but understanding of which hardware will deliver on performance requirements remains challenging. In this work we present LLM-Pilot – a first-of-its-kind system for characterizing and predicting performance of LLM inference services. LLM-Pilot performs benchmarking of LLM inference services, under a realistic workload, across a variety of GPUs, and optimizes the service configuration for each considered GPU to maximize performance. Finally, using this characterization data, LLM-Pilot learns a predictive model, which can be used to recommend the most cost-effective hardware for a previously unseen LLM. Compared to existing methods, LLM-Pilot can deliver on performance requirements 33% more frequently, whilst reducing costs by 60% on average.**

***Index Terms*—large language models, inference services, performance, benchmarking, prediction**

I. Introduction

Large Language Models (LLMs) have gained massive popularity in recent years, in both industry and the research community [58] for their remarkable capability of performing a wide variety of natural language processing tasks [27]. In particular, LLMs excel at generating text and programming code, conducting conversation as chatbots, extracting information from text, and language translation. The development of LLMs is only accelerating and considered to be a modern-day Moore's law [2], [40], with numbers of parameters of newly released LLMs growing exponentially [26], [35], [53]. This is largely due to the fact that, until now, training larger LLMs has been a sure path to improving the LLM's output quality [4], [17]. Unfortunately, growing LLMs in size leads to the necessity to scale the hardware used for their training and inference [2], [43]. As most research efforts in this area go into improving the performance of LLM training, optimizing the performance of LLM inference services remains a fairly unexplored area [39]. Therefore, we are starting to observe a new challenge emerging: *Once you have trained a state-of-the-art LLM with billions of parameters, how do you deploy it in a cost-effective way while ensuring sufficient performance?*

The choice of hardware to which the inference service is deployed can significantly impact the resulting performance [39]. As LLM inference services must store billions of model weights, the choice of hardware is limited to a range of powerful GPUs, which are currently scarce and in all-time high demand [31]. As a result, GPUs are both hard to get and hard to afford. While cloud providers offer on-demand GPU instances, in the long-term they are more expensive than buying GPUs and institutions often prefer to pool on-prem resources instead [43]. In this work we consider the perspective of such an institution, which owns and maintains a large cluster of heterogeneous GPUs. Each GPU has some cost per unit time associated with it, which may be related to the total cost of ownership, energy consumption or some combination thereof. We consider two roles with respect to this cluster: a cluster user and a cluster administrator. The user wishes to deploy an LLM inference service and has a set of performance requirements that should be met, while the administrator has an interest that each inference service incurs as little cost as possible, thus ensuring that the resources are efficiently utilized. In some private cloud-like scenarios, the burden of cost may also be passed onto the user, who may be billed internally for the resources consumed. Trying to satisfy both performance and cost requirements is challenging since the performance of a given LLM inference service on a given GPU in the cluster is a-priori unknown. Additionally, it is impractical for the user to run a large set of experiments to characterize performance across different GPUs, because the cluster is typically close to fully-utilized. This motivates the two problems considered in this work: (1) how the administrator can collect data offline to characterize the performance of different LLMs on different GPUs and (2) how this data can assist the user to make online decisions about the type and number of GPUs that will satisfy their performance requirements in the most cost-effective way.

In this work we present LLM-Pilot, a first-of-its-kind system for characterizing the performance of LLM inference services, and recommending the cheapest deployment for a new LLM such that its performance requirements are met. LLM-Pilot consists of two main components: the performance characterization tool and the GPU recommendation tool. The performance characterization tool can be used offline, by the cluster administrator, to benchmark the performance of a collection of LLM inference services across the GPUs of the cluster. While doing so, it ensures that the inference service is subject to a realistic load of inference requests and that the server-

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

side batching algorithm has been configured individually for each GPU. The output is a characterization dataset containing performance measurements for a variety of LLMs across the different GPUs of the cluster. The GPU recommendation tool provides the cluster user with online recommendations regarding how to deploy an unseen LLM in the most cost-effective way whilst satisfying their performance requirements. This is achieved by learning a predictive performance model, fitted to the offline characterization data, and tailored to the user's specific performance requirements. In summary, the contributions of this work are as follows:

1) An LLM inference benchmarking tool[1] which ensures that the services are optimized for the specific hardware, and internally uses our novel workload generator[1], which subjects the services to a realistic workload of inference requests based on a large collection of production traces.
2) A performance dataset comparing the inference performance of many LLMs running on a variety of GPUs, which we have open sourced in order to deepen the understanding of LLM inference performance and its dependence on the GPU choice[2].
3) A GPU recommendation tool[2] which can ensure satisfying the performance requirements 33% more frequently than state-of-the-art methods, and reduces cost of recommended type and number of GPUs by 60% on average, thanks to the use of a novel performance prediction model designed specifically for this use-case.

The structure of this work is as follows: in Sec. II we discuss the background on the performance and deployment of LLM inference services. Then, we present the performance characterization tool (Sec. III) and the GPU recommendation tool (Sec. IV) of LLM-Pilot. In Sec. V we evaluate various novel components of LLM-Pilot, and in Sec. VI we discuss prior works on related topics.

II. Background

A. LLM inference performance requirements

Unlike inference services based on classical machine learning or deep learning models, LLMs process requests in two phases. First, in the so-called *prompt processing* phase, they split the input text into smaller segments called tokens, process them and populate the key-value (KV) cache [19], and finally generate the first output token. Then, in the *decode* phase, they update the KV cache and sequentially generate the output tokens, which are later converted into an output text and sent to the user. The latency of these phases (and consequently the end-to-end latency) can vary significantly depending on the number of input and output tokens. Therefore, in case of LLMs one typically defines two latency metrics. Time to first token (TTFT) measures the overhead of generating the first output token resulting from processing the input tokens. The inter-token latency (ITL) measures the latency of generating each subsequent output token. Depending on the application, one of these metrics can have larger impact on the end user's experience than the other. For example, in case of LLM-powered chatbots, it is important for the user's experience that the output starts being generated quickly, while the speed of subsequent tokens does not need to exceed the human reading speed [60]. On the other hand, when LLMs are used for text summarization, longer initial overhead is acceptable as long as long output texts can be generated quickly [60]. Inference services are typically subject to a service-level agreement (SLA) which outlines the requirements regarding the service's performance [33]. In case of LLM inference services, however, one must ensure that both TTFT and ITL meet their respective SLA constraints.

B. LLM inference servers

In practice, in order to perform inference on a trained LLM, one uses a framework called an inference server (e.g., vLLM [19], TGIS [14], TGI [11], Orca [57], NVIDIA Triton Inference Server [30]), which acts as a bridge between the LLM and the users sending their inference requests. The inference server handles running all incoming requests through the LLM and sending back the generated responses. As the incoming inference requests can strongly vary in terms of number of input and output tokens, inference servers typically use *continuous batching* [57] to improve utilization of the hardware hosting the inference service. In continuous batching, the server maintains a single batch of requests from various users that are being processed at any given moment. When processing of certain requests in the batch finishes, new requests are introduced into the batch from the queue, while other requests in the batch continue processing. This way, requests with diverse numbers of input and output tokens can be processesed in parallel but small requests can be processed quickly, without waiting for larger requests in the batch to finish processing.

When an LLM is deployed on a GPU, a large amount of its capacity is used for storing the LLM itself. The remaining capacity is required for storing the KV cache associated with the batch of requests currently being processed, as well as any auxiliary data structures required to perform a forward pass with the LLM. The larger the storage space assigned to the batch, the more requests can be processed in parallel, which in turn improves the throughput of the inference service. Inference servers control the maximum size of the batch using equivalent parameters under different names, e.g., max batch weight in TGIS [14], max num batch tokens in vLLM [19], and max batch total tokens in TGI [11]. In this work, we refer to it as the *maximum batch weight*. The maximum batch weight is the maximum allowed 'volume' of the batch, defined as the total number input and output tokens of all requests processed in the batch at any given time. This indirectly determines the maximum storage space that can be occupied by the batch and significantly impacts the inference performance.

Tuning the maximum batch weight is critical to achieving optimal inference performance. For example, when the max-

[1]Available at: https://github.com/fmperf-project/fmperf.

[2]Available at: https://github.com/IBM/LLM-performance-prediction.

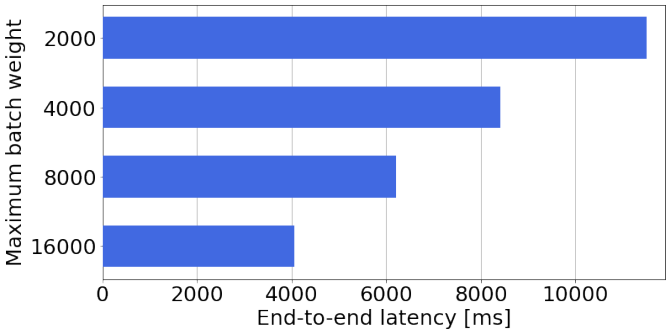


Fig. 1: Median end-to-end latency achieved by the inference service of a selected LLM (bigcode/starcoder [21]) deployed on one A100 GPU with varying maximum batch weight, for 128 concurrent users.

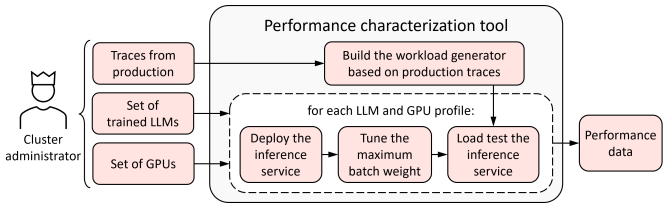


Fig. 2: Architecture of the performance characterization tool.

imum batch weight is doubled, we can expect the queuing time of requests to decrease because twice as many tokens are being processed in parallel. However, the end-to-end latency comprises both the queuing time and the processing time (i.e., time needed to generate all output tokens). If doubling the maximum batch weight increases the processing time of each request by more than 2×, the final end-to-end latency will actually be worse. Therefore, in Fig. 1 we analyze the end-to-end latency of a selected LLM inference service running on the same GPU with varying maximum batch weight. The chart shows that increasing the maximum batch weight improves the end-to-end latency: the largest maximum batch weight results in approx. 2.8× lower end-to-end latency than the smallest one. This confirms that (a) the maximum batch weight strongly impacts the inference performance, and (b) in order to optimize performance under the load of inference requests from our workload generator (see Sec. III-B), the maximum batch weight should be set as high as possible.

C. Deploying LLM inference services

When an inference service is to be used in production, it is deployed onto a Kubernetes (k8s) [7] or Openshift [36] cluster via an abstraction called a *Deployment*. Each deployment manages a number of replicas of the inference service, and each replica is deployed to the cluster via an abstraction called a *Pod*. Load balancing is performed across the pods of the deployment, which operate independently, and the number of pods can be scaled up or down based on demand.

Since inference is embarrassingly parallel at the level of requests, and LLM inference requests can easily take more than 100ms, we expect close-to-perfect scaling of the throughput with respect to the number of pods. We confirm this experimentally, as depicted in Table I. In the experiment, we have tested different numbers of pods of the same LLM inference service running on the same GPU, under different numbers of concurrent users whose requests are distributed across pods. Results confirm near-perfect scaling – across cases with the same ratio between the number of concurrent users and number of pods (e.g., 1 pod and 8 users, 2 pods and 16 users, etc.), indicated by the same cell background color, the relative standard deviation of throughputs per pod never exceeds 5% (2% on average).

TABLE I: Average throughput per pod achieved by a varying number of Llama-2-13b pods, each running on a A100 80GB GPU, for various numbers of concurrent users. Same color of diagonally adjacent cells marks cases with the same ratio between the number of pods and the number of concurrent users.

Number of pods	Number of concurrent users							
	1	2	4	8	16	32	64	128
1	47.1	78.1	118.2	174.2	237.8	288.6	314.7	292.6
2	23.5	44.4	74.7	114.3	171.1	229.8	282.7	286.3
4	11.6	23.3	43.0	73.5	113.0	169.5	231.4	283.4
8	5.7	11.7	22.8	42.3	72.6	112.9	169.2	231.1

When defining the deployment specification, one must declare what hardware resources should be allocated to each pod. These include the number and type of GPUs to be assigned to each pod (which we jointly refer to as the *GPU profile*), as well as the number of CPU cores and amount of memory. Note that in the case when the specified GPU profile comprises more than one GPU, the weights of the LLM and all computation will be sharded across the GPUs in a tensor-parallel manner. Tensor parallel deployments may be preferred when dealing with very large models that are too big to fit in a single GPU, and in some cases may also bring latency benefits [38]. Note that in this setup, each pod has exclusive access to the GPUs assigned to it, and thus there are no effects related to co-location that need to be considered.

III. Performance Characterization Tool

In this section we present the performance characterization tool, the structure of which is presented in Fig. 2. There are two important aspects of performance characterization that we have considered when developing LLM-Pilot. Firstly, we ensure that the LLM's performance will be characterized under a *realistic* workload. Therefore, we have acquired and analyzed a large collection of production traces of real inference requests to a variety of LLMs (Sec. III-A). Then, we have developed our own workload generator (Sec. III-B) based on the production traces to ensure that the generated workload resembles the real usage of LLM inference services. Secondly, the performance characterization tool ensures that the maximum batch weight is *optimized* for each GPU profile. As shown in Fig. 2, on each benchmarked GPU profile LLM-Pilot deploys the inference service, tunes the maximum batch

TABLE II: Characteristics of the production traces used to develop the workload generator.

Time period	5.5 months
Number of requests	17.3M
Number of users	approx. 2500
Number of LLMs	24 (with 3B–176B parameters)
Range of tokens	input: 1–4093, output: 1–1500
Batch sizes	1–5
Additional parameters describing the requests	33 (e.g., decoding method, top k, top p, repetition penalty, length penalty, temperature)

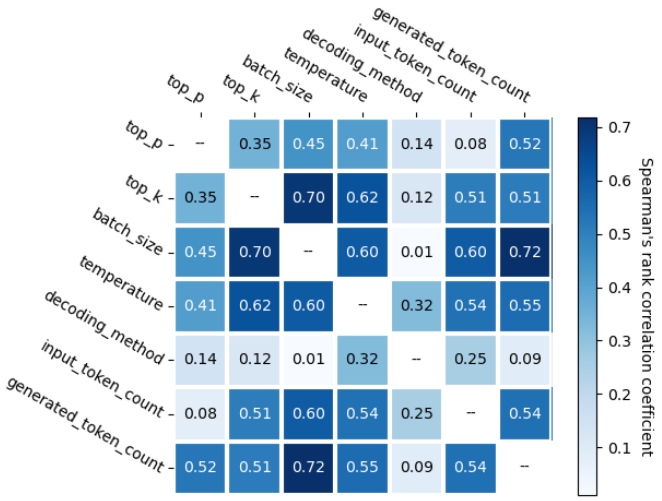


Fig. 3: Correlation between selected parameters of requests from the production traces.

weight, and subjects the service to a series of requests from the workload generator, collecting various performance metrics (Sec. III-C). Other considerations taken into account when developing the performance characterization tool of LLM-Pilot have been discussed in Sec. III-D.

A. Analysis of production traces

The production traces analyzed in this section come from an LLM inference platform used internally in our organization, which hosts a large number of LLMs deployed on an Openshift cluster running on A100 GPUs, which are made available for many users to send inference requests. Internally, the platform uses Text Generation Inference Server (TGIS) [14]. The traces are a record of every inference request (i.e., every request to perform inference on a certain input text using a specific LLM and return the output) sent to the inference platform by every user within a certain period of time. Each entry in the traces includes the user's id, timestamp and all details of the request (including various TGIS-specific request parameters set by the user), as well as output from the LLM and end-to-end latency of processing the request. As shown in the detailed characteristics in Table II, the traces were collected over a long period of time and include millions of diverse requests from thousands of users. To the best of our knowledge, the traces used in this work are larger than any publicly available LLM trace collection published to date [20], [23], [28], [59]. Furthermore, the traces were in no way generated or collected synthetically and therefore represent a fully realistic and diverse usage of LLM inference services.

Impact on latency: As shown in Table II, each request in the traces is described by a large number of parameters. We analyze the impact of these parameters on the latency through an importance study using a Random Forest (RF) regression model. First, we trained a RF regressor on all available trace data to predict the latency of all individual requests using all parameters included in the traces. The model achieved good accuracy, with the coefficient of determination $R^2 \approx 0.93$. Then, we evaluated their impact on the RF's predictions using the Mean Decrease in Impurity (MDI) [3]. According to our study and our expectation, the parameter with the largest influence on the performance is the number of output tokens, followed by the number of input tokens, the batch size and parameters related to token sampling (decoding method, temperature, top p, top k).

Correlation: Further, we have analyzed the correlation between the request parameters listed above using the Spearman's rank correlation coefficient [41]. The results presented in Fig. 3 indicate that many pairs of parameters are strongly correlated. Most notably, the parameters with the strongest influence on the performance – the batch size and the numbers of input and output tokens – are all strongly correlated with one another. Therefore, in order for a workload generator to produce realistic workloads, it should preserve the correlation between the parameters characterizing each request.

B. Workload generator

Based on the conclusions drawn in Sec. III-A, in each request the workload generator should specify the parameters which impact the latency, and it must consider the fact that in practice some parameters are strongly correlated. These considerations are an important contribution of our work, as to the best of our knowledge, no prior works on workload generation include any request parameters beyond the batch size and the numbers of input and output tokens [9], [12], [19], [34], [37], [46], and in some works the request parameters are treated as independent variables [12], [34].

1) Modelling the requests: Internally, the workload generator uses a non-parametric model of requests, which jointly models the distributions of all request parameters. For each parameter describing the requests, we divide the range of its values into a series of intervals called *bins*. This allows us to reduce the cardinality of each parameter, as the true parameter values will be replaced with the centers of their respective bin intervals. For each parameter, we define 64 bins (unless the cardinality of the parameter is lower, in which case we define as many bins as there are unique values). We aim to define the bins such that they all contain an approximately equal number of requests. Then, we proceed to create the joint, multi-dimensional model of requests. Each multi-dimensional

bin is defined by a distinct combination of bin assignments for values of all parameters.

2) Sampling requests: Whenever the workload generator needs to produce an inference request, it can draw a random sample from the model of requests by selecting one of its multi-dimensional bins. The probability of choosing each bin is defined by the histogram of the multi-dimensional bins, i.e. it is proportional to the number of requests from the traces that were assigned to it. This way, the distribution of drawn samples will be very similar to the empirical distribution observed in the traces. The final sample is a request with each parameter equal to the center of that parameter's interval in the selected bin. The input text for the request is generated based on some designated corpus of texts, truncated to match the number of input tokens indicated by the request's parameters.

C. Performance data collection

For each LLM and GPU profile to be benchmarked, LLM-Pilot performs a series of actions: (1) it deploys the inference service, (2) tunes the maximum batch weight parameter of the inference server to ensure maximum GPU utilization, and (3) runs a series of load testing experiments using the workload generator, collecting various performance metrics. These steps will now be explained in detail.

1) Deployment: The tool creates a TGIS deployment on the cluster, using a single pod, and with the number and type of GPUs set according to the given GPU profile. The amount of memory available to the pod is set to 250GB, and the number of CPU cores is set to be twice larger than the number of GPUs. LLM-Pilot then waits until the pod is created and the LLM has been loaded into GPU memory, before proceeding to the next step.

2) Tuning the batch weight: Based on the conclusions drawn in Sec. II, we must ensure that the maximum batch weight parameter is set as high as possible to achieve the best possible performance of the inference service. As GPU profiles vary in terms of memory capacity, so does the highest possible maximum batch weight that can be achieved. Thus, in order to be able to compare GPU profiles in a fair way, we must always ensure that the batch weight has been optimized individually for each one. This is achieved in LLM-Pilot by running a binary search to optimize the maximum batch weight, as an initialization step before starting the inference server. Namely, in each step of the search, we test a different maximum batch weight value and check if we encounter Out Of Memory (OOM) errors. This is achieved by passing a sequence of batches to the model that are designed to test all possible corner cases, with respect to the batch size, number of input and output tokens, that can be constructed according to the given maximum batch weight. If all corner cases succeed (i.e., none of them result in an OOM error), the batch weight is considered valid, and otherwise it is not. Once binary search is completed, we take the optimized value for the maximum batch weight and start the inference server. Once the server is ready to receive requests, we proceed to perform load testing.

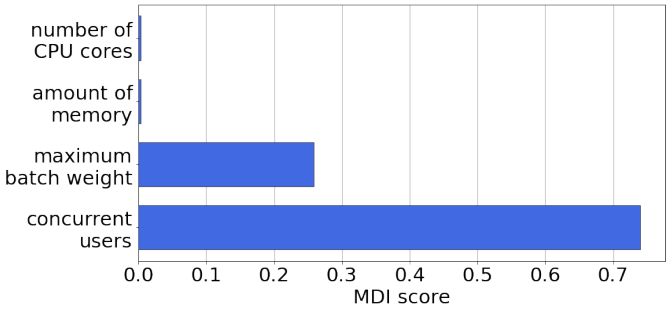


Fig. 4: The MDI importance scores of the number of CPU cores, amount of memory, maximum batch weight and number of concurrent users, determined by a RF predicting the TTFT and ITL latency for a selected LLM (bigcode/starcoder [21]).

3) Load testing: In each load testing experiment, LLM-Pilot simulates a different number of concurrent users simultaneously sending various requests generated by the workload generator (Sec. III-B), for a duration of 2 minutes. By default, subsequent experiments simulate 1, 2, 4, ..., 128 concurrent users, increasing exponentially. In each experiment, LLM-Pilot logs all generated tokens and the timestamps of their arrival to the client. From the timestamps, the following performance metrics are extracted:

- time to first token (TTFT) – the median latency of receiving the first output token. It includes the time spent on queueing and the prompt processing phase.
- normalized TTFT (nTTFT) – the median of TTFT latencies of requests divided by their number of input tokens. We create this new metric as its value does not change as significantly as TTFT with the number of input tokens.
- inter-token latency (ITL) – the median latency between all subsequent output tokens, excluding the first one.
- throughput – the total number of output tokens generated throughout the experiment, divided by its duration.

Once these three steps have been performed for all LLMs and GPUs, all of the collected performance data are aggregated into a characterization dataset, which is discussed in detail in Sec. V-B. While there exist other tools for benchmarking the performance of LLM inference services (discussed in Sec. VI-B), our work is the only one that benchmarks *optimized* inference services by finding the appropriate value of maximum batch weight to maximize the inference performance.

D. Other considerations

While we have shown that tuning the maximum batch weight is critical to evaluate performance across different GPUs, a natural question is whether other aspects of the deployment specification also need to be tuned (e.g., the amount of memory and number of CPU cores). Similarly to the study in Fig. 1, we have run an example LLM's inference service on a single A100 40GB GPU with varying number of CPU cores, amount of memory and maximum batch weight,

and measured the inference performance for different numbers of concurrent users as described in Sec. III-C. Then, we have trained a RF regressor to predict the TTFT and ITL latencies based on the varying parameters, and performed an importance analysis similarly to Sec. III-B. The MDI importance scores of all parameters are presented in Fig. 4. The number of CPU cores and memory achieved near-zero scores, over 300× lower than the MDI score of maximum batch weight. This suggests that they do not have considerable impact on performance and motivates why LLM-Pilot sets them according to trivial rules.

IV. GPU Recommendation Tool

In this section, we describe LLM-Pilot's GPU recommendation tool, depicted schematically in Fig. 5. In Sec IV-A we define the problem that the GPU recommendation tool aims to solve, and in Sec. IV-B we describe how LLM-Pilot solves it. LLM-Pilot's ability to recommend GPUs for unseen LLMs has been evaluated in Sec. V-C.

A. Problem statement

The input to the GPU recommendation tool consists of: an LLM model M with unknown performance, a set of GPU profiles $\mathbb{G}$ that the user considers for deployment (each defined as the number and type of GPUs assigned to each pod), the latency constraints on nTTFT and ITL denoted as $L_1, L_2 \in \mathbb{R}^+$ respectively (and jointly denoted as $L = (L_1, L_2)$), and the expected load on the service expressed as the total number of concurrent users $U \in \mathbb{Z}^+$ that will be simultaneously sending requests to the service, following the same request distribution as modeled by the workload generator. As we have argued in Sec. I, an important assumption of the recommendation tool is to make no performance evaluations of the unseen LLM M. Instead, LLM-Pilot uses the collection of historical performance data $\mathbb{D}_{\text{train}}$ collected using a set of training LLMs $\mathbb{M}_{\text{train}}$ on GPU profiles $\mathbb{G}$ to make predictions regarding nTTFT l_1 and ITL l_2 of the unseen LLM M. In all equations to follow, we omit the dependence on the total number of users U and the latency constraints L for brevity. The end goal of the GPU recommendation tool is to identify the most cost-effective GPU profile $G^\star \in \mathbb{G}$ and estimate the number n of pods running on GPU profile $G^\star$ that should be created in order for LLM M to serve U concurrent users under constraints L:

$$\text{find} \quad G^\star\big(M\big|\mathbb{D}_{\text{train}}\big) = \arg\min_{G\in\mathbb{G}} n\big(M,G\big|\mathbb{D}_{\text{train}}\big)\cdot c(G)\,, \quad (1)$$

where

$$n\big(M,G\big|\mathbb{D}_{\text{train}}\big) = \left\lceil \frac{U}{u_{\max}\big(M,G\big|\mathbb{D}_{\text{train}}\big)} \right\rceil, \text{ and} \quad (2)$$

$$u_{\max}\big(M,G\big|\mathbb{D}_{\text{train}}\big) = \\ = \max\left\{u\in\mathbb{U}\colon \begin{matrix} l_1\big(M,G,u'\big|\mathbb{D}_{\text{train}}\big)\le L_1 \\ l_2\big(M,G,u'\big|\mathbb{D}_{\text{train}}\big)\le L_2 \end{matrix} \forall_{u'\in\mathbb{U}\colon\, u'\le u}\right\}. \quad (3)$$

Value $c(G) \in \mathbb{R}^+$ denotes the cost of a single pod running on the GPU profile G, read from the GPU pricing tables. Value $n\big(M,G\big|\mathbb{D}_{\text{train}}\big)\in\mathbb{Z}^+$ denotes the number of pods with GPU profile G needed to serve U users under constraint L, predicted based on $\mathbb{D}_{\text{train}}$. $\mathbb{U}\subset\mathbb{Z}^+$ is the set of all considered numbers of concurrent users, while $u_{\max}\big(M,G\big|\mathbb{D}_{\text{train}}\big) \in \mathbb{U}$ denotes the maximum number of users that can be served by a single pod of M deployed on G without violating constraints L, predicted based on $\mathbb{D}_{\text{train}}$. Finally, $l_1\big(M,G,u\big|\mathbb{D}_{\text{train}}\big)$, $l_2\big(M,G,u\big|\mathbb{D}_{\text{train}}\big) \in \mathbb{R}^+$ respectively denote estimated nTTFT and ITL of M deployed on G and serving u concurrent users, predicted using a regressor trained on $\mathbb{D}_{\text{train}}$.

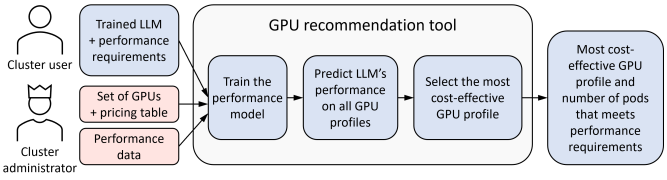


Fig. 5: Architecture of the GPU recommendation tool.

B. LLM-Pilot's solution

Before LLM-Pilot can decide on the most cost-effective GPU profile for an unseen LLM, it uses the performance model to make predictions regarding the unseen LLM's performance. The performance model's training data $\mathbb{D}_{\text{train}}$ was created using the performance characterization dataset described in Sec. V-B. The performance model takes an input the features describing the LLM M, features describing the GPU profile G and the number of concurrent users $u \in \mathbb{U}$. As output, it predicts the latencies l_1 and l_2 of that inference service. The recommendation tool uses these predictions to identify the most cost-effective GPU profile, following Eq. (1)–(3).

1) Feature engineering: The LLMs are characterized by the following set of features: LLM type (e.g., t5, codegen2), whether the LLM has an encoder-decoder or decoder-only architecture, the number of parameters, layers, positions and heads, whether flash attention was used, the vocabulary size, parameters for relative attention (maximum distance and number of buckets) and the data type used for training. The set of features characterizing the GPU profiles includes the number of GPUs, memory capacity and bandwidth, GPU architecture, number of Tensor/RT/CUDA cores, number of texture mapping units, raster operations pipelines, and streaming multiprocessors, TFLOPS for various data types, compute capability, interface generation, form factor (SXM vs. PCIe) and finally whether the GPU is connected using NVLink. The LLM features listed above were explained in detail in [48], and the GPU features in [16].

2) Regressor: Internally, the GPU recommendation tool of LLM-Pilot uses an XGBoost regressor [5]. As the end goal of LLM-Pilot's recommendation tool is to identify cost-effective GPU profiles rather than to make accurate latency predictions, we introduce two modifications to the regression task which will ultimately improve the GPU recommendations. Our first modification is to apply sample weights to our training data, in which the closer each data point's latency metrics are to

the latency constraints, the higher weight is assigned to it. Initially, we define weights based on nTTFT as follows:

$$w_1(M,G,u|\mathbb{D}_M) = 1 - \frac{\left|\hat{l}_1\left(M,G,u|\mathbb{D}_M\right) - L_1\right|}{\max\limits_{v\in\mathbb{U}}\left|\hat{l}_1\left(M,G,v|\mathbb{D}_M\right) - L_1\right|} \quad (4)$$

where $\hat{l}_1\left(M,G,u|\mathbb{D}_M\right)$ denotes the true nTTFT latency of M deployed on G with u concurrent users, extracted from the known performance $\mathbb{D}_M$ of LLM M, as $M \in \mathbb{D}_{\text{train}}$. The weights for the ITL latency $w_2(M,G,u|\mathbb{D}_M)$ are calculated in an analogous way, using the true ITL latency $\hat{l}_2\left(M,G,u|\mathbb{D}_M\right)$ and the constraint L_2. With this formula for the sample weights, training data points have weights inversely proportional to how far their latency is from the respective latency constraint. We combine both weights using arithmetic mean. The intuition behind our use of the sample weights is as follows: as we are solving the GPU recommendation problem (1), the regressor's purpose is to estimate the maximum number of concurrent users that can be served on a given GPU under the latency constraints. Therefore, it has to make accurate latency predictions mainly for those numbers of users for which the latency metrics are near the constraints.

However, the sample weights can also lead to drastically incorrect recommendations. For example, let us assume that for some GPU profile the true latency for 4 concurrent users is far from the latency constraint and therefore, that data point has a low sample weight. The regressor predicts an incorrect, very high latency value for that data point and LLM-Pilot determines that the latency constraint is already violated. At the same time, the regressor made accurate latency predictions for 16 and 32 concurrent users, as these data points are close to the latency constraint and have high sample weights. However, constraint violation for 4 concurrent users causes LLM-Pilot to determine that this GPU profile can only support 2 concurrent users and massively overestimate the number of pods needed. Therefore, we additionally enforce on the regressor a monotonicity constraint on the number of concurrent users, as based on our experiments, as the number of concurrent users increases, the nTTFT and ITL of the service increase or stay constant:

$$u' < u'' \implies l\big(M,G,u'\big|\mathbb{D}_{\text{train}}\big) \leq l\big(M,G,u''\big|\mathbb{D}_{\text{train}}\big) \quad \forall_{u',u''\in\mathbb{U},\ l\in\{l_1,l_2\}}$$

With the addition of the monotonicity constraint, if the regressor makes accurate predictions for data points close to the latency constraint, it will never incorrectly indicate violation or satisfaction of the constraint for the other data points. This way, the monotonicity constraint ensures that using the sample weights does not negatively impact the estimation of the maximum number of concurrent users.

3) Hyperparameter tuning: Before training an XGBoost regressor, one must first set a number of *hyperparameters* (HPs), which cannot be tuned as part of the training process but strongly impact the quality of the regressor's predictions. For XGBoost, these include the number of boosted trees, their maximum depth, learning rate, subsampling rates, the tree building method, and the number of bins for the histogram tree method. We tune XGBoost's HPs via a leave-one-LLM-out cross-validation procedure. We split the available performance data into the training dataset $\mathbb{D}_{\text{train}}$ used to train the regressor, and the validation dataset $\mathbb{D}_{\text{val}}$ used to evaluate the predictions. Specifically, all performance data from one LLM is used as $\mathbb{D}_{\text{val}}$ and all remaining LLMs act as $\mathbb{D}_{\text{train}}$. Finally, we select the configuration of HPs that achieved the lowest average validation error across all possible training/validation splits. As the error metric, we use the mean absolute percentage error (MAPE) weighted using the sample weights defined in Eq. (4) because it measures the error relative to the latency values, which vary significantly within our data.

4) Final GPU recommendation: Once the performance model has predicted the latencies for an unseen LLM M across all GPU profiles and numbers of concurrent users, LLM-Pilot recommends the most cost-effective GPU profile and the number of pods that will safisfy the performance requirements, following Eq. (1)–(3). Then, LLM-Pilot can be used to tune the maximum batch weight for that LLM on that GPU profile and to deploy the inference service, as described in Sec. III-C.

V. Analysis and Evaluation

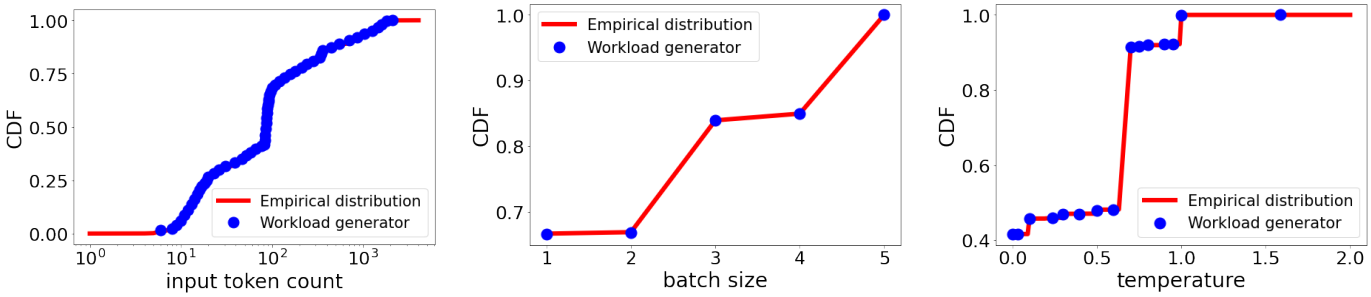
In this Section we analyze and evaluate LLM-Pilot's workload generator (Sec. V-A), the performance dataset collected using the performance characterization tool (Sec. V-B), and the GPU recommendation tool (Sec. V-C).

A. Workload generator

To evaluate the workload generator developed in this work (Sec. III-B), we analyze: (1) whether the generator's internal model of requests accurately models the distributions of all parameters describing the traces, (2) whether the preserved correlation between request parameters has any effect on the inference performance, and (3) whether using the workload generator has any benefits over generating requests by drawing random samples from the traces.

Accurate modelling: In Fig. 6a-6c we compare the empirical marginal cumulative distribution function (CDF) of selected request parameters to the marginal CDF obtained with the workload generator. Based on the plots we can conclude that the workload generator preserves the marginal distributions of parameters with both very high and low cardinality.

Parameter correlation: We have conducted an experiment for an example test case (Llama-2-13b running on one A100 80GB GPU) to prove that the correlation between request parameter affects the LLM inference performance. On average, across 1–128 concurrent users, generating parameter values from independent marginal distributions results in 13% lower throughput (up to 19%), 30% higher median TTFT (up to 98%) and 25% lower median ITL (up to 58%), as compared to generating them based on a joint distribution. We consider these differences significant enough to justify that modelling request parameters jointly is crucial for the measured performance to reflect what would be observed in production.



(a) Number of input tokens (b) Client-side batch size (c) Temperature

Fig. 6: Marginal CDFs of selected request parameters in the empirical distribution and in the workload generator.

TABLE III: LLMs and GPUs included in our performance characterization dataset: combinations for which data was collected (✓), combinations in which the GPU profile's memory capacity was too small to host the LLM while leaving sufficient space to process workload generator's requests (×), and combinations omitted due to software or hardware limitations (–).

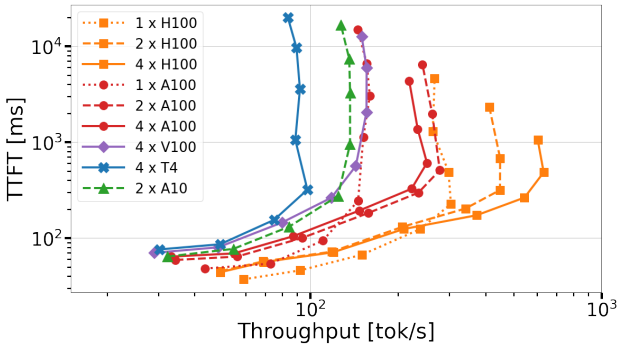
LLM		H100 (80GB)			A100 (40GB)			A10 (24GB)		T4 (16GB)			V100 (16GB)		
		1	2	4	1	2	4	1	2	1	2	4	1	2	4
google/flan-t5-xl [6]	3B	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
google/flan-t5-xxl [6]	11B	✓	✓	✓	✓	✓	✓	×	✓	×	×	✓	×	×	✓
google/flan-ul2 [45]	20B	✓	✓	✓	×	✓	✓	×	×	×	×	×	×	×	×
ibm/mpt-7b-instruct2 [13]	7B	✓	–	–	✓	–	–	×	–	×	–	–	×	–	–
bigscience/mt0-xxl [24]	13B	✓	–	–	✓	–	–	×	–	×	–	–	×	–	–
Salesforce/codegen2-16B [29]	16B	✓	–	–	×	–	–	×	–	×	–	–	×	–	–
Llama-2-7b [47]	7B	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	–	–	–
Llama-2-13b [47]	13B	✓	✓	✓	✓	✓	✓	×	✓	×	×	✓	–	–	–
EleutherAI/gpt-neox-20b [1]	20B	✓	✓	✓	×	✓	✓	×	✓	×	×	✓	–	–	–
bigcode/starcoder [21]	15B	✓	✓	✓	✓	✓	✓	×	✓	×	×	✓	–	–	–

Size and sampling speed: Because the parameters are strongly correlated, many combinations of parameters never occur, and their respective multi-dimensional bins are empty. Consequently, the collection of multi-dimensional bins in the workload generator is sparse, with 46.5 thousand non-empty bins, compared to 10.7 billion theoretically possible combinations of parameter bin assignments. Thanks to binning the parameter values and sparsity, the workload generator is much smaller in terms of storage space than the traces that it models (<1MB generator, compared to 1.6GB of traces), and will remain approximately the same size even if a much larger amount of traces is collected in the future. Furthermore, the workload generator produces requests much faster than directly sampling past requests from the traces. For an example of drawing 1000 samples, sampling requests from traces takes 770ms, while the workload generator produces the requests in 22ms, which is 35× faster. It is worth noting that the time needed to generate 1000 requests is lower than the typical ITL of generating a single output token.

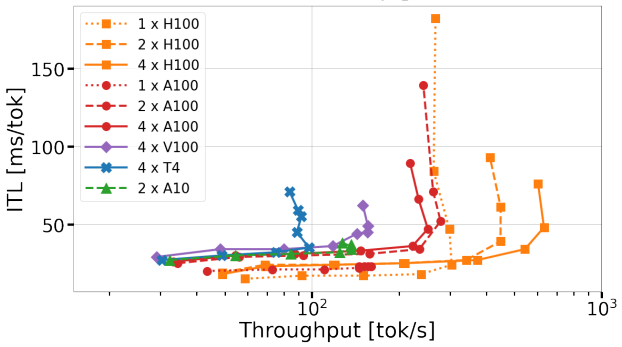
B. Performance characterization

Using LLM-Pilot's performance characterization tool, we performed a series of performance measurements using 10 LLMs deployed on 14 GPU profiles (single-GPU or tensor-parallel deployment across 2 or 4 GPUs, with one of 5 GPU types). Table III presents which combinations of LLMs and GPU profiles were feasible. For certain combinations it was impossible to collect performance data because the LLM would not fit into the GPU profile's aggregate memory or because the free space after loading the LLM into memory was insufficient to process the largest requests produced by the workload generator. Additionally, certain combinations were impossible due to software or hardware constraints. Specifically, at the time of writing this work TGIS didn't support tensor parallelism for certain LLMs. Furthermore, TGIS uses flash attention [8] for some LLMs and therefore these LLMs couldn't be deployed on the V100 GPUs because of insufficient CUDA capability.

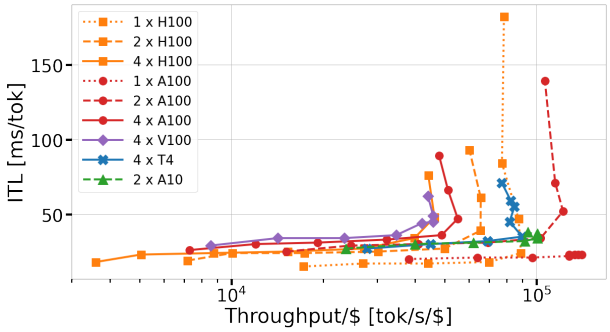
Performance data analysis: In Fig. 7a-7b we present the relationship between median latencies (TTFT and ITL) and throughput achieved by an example LLM deployed on various GPU profiles. As the prompt processing phase is compute bound [42], the typical behavior that we observe across all LLMs is that TTFT increases linearly with the increasing number of concurrent users because the LLM processes a larger batch of requests at the same time. For weak GPUs with many concurrent users, we can observe a sudden jump of TTFT due to increased queueing time. On the other hand, the decode phase is memory bandwidth bound [42]. As a result, ITL typically remains stable as the number of concurrent users and throughput increase, until the memory capacity is saturated. As the number of concurrent users increases further, the ITL increases rapidly, while throughput does not improve anymore. We can also observe that the larger the total memory capacity of the GPU profile, the larger number of concurrent users marks the point of saturation. Consequently, GPU profiles with



(a) TTFT vs. throughput



(b) ITL vs. throughput



(c) ITL vs. throughput per cost

Fig. 7: Relationship between TTFT, ITL, throughput, and throughput per dollar for google/flan-t5-xxl LLM [6] running on a variety of GPUs. Markers on each curve mark exponentially increasing numbers of users $\in \{1, 2, 4, ..., 128\}$.

larger memory capacity (and larger maximum batch weights) achieve higher throughput and lower ITL before saturation.

Additionally, in Fig. 7c we present the relationship between median ITL and throughput per dollar for the same LLM. Throughput per dollar is defined as throughput divided by the cost of the respective GPU profile, and can help quantify the trade-off between the inference performance achieved using various GPU profiles and the cost that they incurred. As the cost metric, we use hourly on-demand GPU instance prices collected from Amazon Web Services (AWS) pricing tables. However, the user of LLM-Pilot could also plug in their own pricing table or use any other metric to express their preference for certain GPUs. The plot demonstrates that the GPU profiles with the highest memory capacity are not necessarily the most cost-effective. For example, although profiles with H100 GPUs outperform others in terms of maximum throughput, profiles using A100 and T4 GPUs achieve higher throughput per dollar. However, if the service is subject to a very low latency constraint, many GPU profiles will not deliver on the SLA, even with just one concurrent user. In such cases, it is necessary to use GPU profiles with higher memory capacity despite their high cost.

To the best of our knowledge, the dataset that we have collected is the first work comparing the performance of many LLMs on a variety of GPUs (discussed in Sec. VI-B). It is also the first dataset in which the maximum batch weight was optimized individually for each GPU profile. We have made our performance characterization dataset public in the hope that it will benefit the community and extend the efforts towards maximizing the performance of LLM inference.

Characterization overhead: We expect the characterization tool to be used whenever users wish to add support for new LLMs or GPUs, which could happen frequently if support for many novel LLMs is desired. We estimate that collecting a performance dataset of similar size to ours would take approx. 8h: 5h to tune the maximum batch weights for all LLMs (30min/LLM, parallelized over GPUs), and 3h to run load testing experiments (20min/LLM, parallelized over GPUs). We note that the characterization tool was designed to be used offline by the cluster administrator, so although its overhead is nonnegligible, it does not disrupt the use of the GPU recommendation tool.

C. GPU recommendation

We evaluate LLM-Pilot's GPU recommendation tool, and multiple state-of-the-art performance prediction methods, using the performance characterization dataset described in Sec. V-B. We simulate a set of "unseen" LLMs, $\mathbb{M}_{\text{test}}$, via a nested cross-validation procedure, i.e., by iteratively excluding one LLM M from the performance characterization dataset and assuming its performance is unknown. We then tune the HPs of each method using the cross-validation procedure described in Sec. IV-B3 using only the performance data collected from the remaining models. We then use the regressors with tuned HPs to make performance predictions for M, and make recommendations. In the following experiments, we assume that the required total number of concurrent users $U = 200$, the latency constraints on nTTFT and ITL are equal $L_1 = 100ms$ and $L_2 = 50ms$, respectively, and the possible numbers of concurrent users per pod $u \in \mathbb{U} = \{1, 2, 4, 8, ...128\}$.

Evaluation metrics: In order to evaluate LLM-Pilot's ability to solve problem (1), we have defined three evaluation metrics. The first evaluation metric is **success rate** $\mathcal{S}$. The recommendation for LLM $M \in \mathbb{M}_{\text{test}}$ is considered a success $\mathcal{S}_M$ if, after the user has followed the tool's suggestion and deployed n pods with the GPU profile $G^\star\left(M \middle| \mathbb{D}_{\text{train}}\right)$ (denoted as $G^\star_M$ for brevity), the true performance of the inference service did in

fact meet their initial requirements regarding the total number of concurrent users U and latency constraints L:

$$\mathcal{S}_M = \begin{cases} 1 \text{ if } n\big(M, G^\star_M | \mathbb{D}_{\text{train}}\big) \cdot \hat{u}_{\max}\big(M, G^\star_M | \mathbb{D}_M\big) \geq U, \\ 0 \text{ otherwise.} \end{cases} \tag{5}$$

where $\hat{u}_{\max}\big(M, G^\star_M | \mathbb{D}_M\big)$ denotes the true maximum number of concurrent users that can be served within a single pod with the GPU profile $G^\star_M$ without violating the latency constraints L which could be determined if the real performance data $\mathbb{D}_M$ of LLM M was known. To balance the success rate, in successful cases $\mathbb{M}^{\mathcal{S}}_{\text{test}} = \{M \in \mathbb{M}_{\text{test}} : \mathcal{S}_M = 1\}$ we additionally calculate the **relative overspend** $\mathcal{O}_M \in \mathbb{R}^+$, which quantifies the relative difference between the expense that the user carried by following the tool's recommendation (n pods with GPU profile $G^\star_M$ determined by LLM-Pilot) and the expense of the truly most cost-effective deployment (denoted as $\hat{n}$ pods with the GPU profile $\hat{G}^\star_M$) that they could have chosen if they knew the real performance $\mathbb{D}_M$ of LLM M:

$$\mathcal{O}_M = \frac{n\big(M, G^\star_M\big) \cdot c\big(G^\star_M\big) - \hat{n}\big(M, \hat{G}^\star_M | \mathbb{D}_M\big) \cdot c\big(\hat{G}^\star_M\big)}{\hat{n}\big(M, \hat{G}^\star_M | \mathbb{D}_M\big) \cdot c\big(\hat{G}^\star_M\big)}. \tag{6}$$

We obtain the final success rate $\mathcal{S} \in [0, 1]$ by averaging the successes of all unseen LLMs $\mathbb{M}_{\text{test}}$, and the mean relative overspend $\mathcal{O} \in \mathbb{R}^+$ by averaging the relative overspends over all unseen LLMs for which the recommendation was successful $\mathbb{M}^{\mathcal{S}}_{\text{test}}$. The purpose of using both metrics above is for them to compliment each other: the success rate penalizes underpredicting the number of GPUs needed, while the overspend penalizes overpredicting them. Finally, we define the **S/O score** $\mathcal{SO} \in [0, 1]$, which combines the success rate $\mathcal{S}$ and the inverse of overspend $\mathcal{O}$ using the harmonic mean:

$$\mathcal{SO} = \frac{2 \cdot \mathcal{S} \cdot \max(0, 1 - \mathcal{O})}{\mathcal{S} + \max(0, 1 - \mathcal{O})}. \tag{7}$$

The S/O score serves as the most important metric in our study, as it directly evaluates how well we solve problem (1).

Baselines: While to the best of our knowledge there are no prior works that predict the inference performance of LLMs across various GPUs, we have implemented several methods developed in related fields. **PARIS** [55] predicts the performance of a previously unseen application across many virtual machine (VM) types in the cloud. First, PARIS measures the performance of the unseen application on two reference VM types: the weakest and the most powerful one. Then, it uses a RF regressor to predict the inference performance of that application running on other VM types, based on the features describing the application and the performance measurements collected on the reference VM types. In our implementation of PARIS, the performance measurements consist of nTTFT, ITL and throughput values for all numbers of concurrent users for two reference GPU profiles: 1×T4 and 4×H100, which, respectively, have the weakest and the strongest memory and computing parameters. To evaluate how the performance measurements of the reference VM types improve the quality of RF predictions in PARIS, we have also implemented a

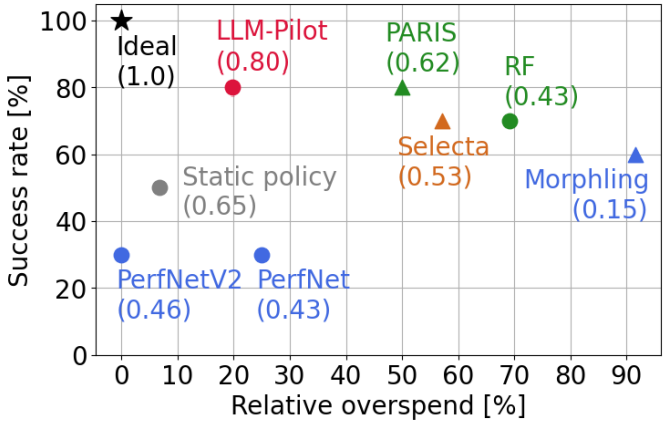


Fig. 8: Evaluation of the quality of recommendations made by LLM-Pilot and baselines. Numbers in parentheses present the S/O scores achieved by each method. We mark methods which make reference performance measurements using ▲, and methods which make no reference evaluations with ●. Additionally, ★ marks the theoretical ideal performance.

RF predictor that uses as input only features describing the LLM, and makes no performance measurements. **Selecta** [18] was developed for the same use-case as PARIS. Internally, it builds a sparse matrix containing the known runtimes of historical and reference (application, VM type)-combinations, and predicts the missing entries via collaborative filtering. We have implemented Selecta using the same library as the original work [10], and have chosen the same reference GPU profiles as we used for PARIS. **Morphling** [51], **PerfNet** [49], and **PerfNetV2** [50] predict the performance of inference services using neural network (NN) models. We have implemented all three NNs ourselves. As Morphling additionally fine-tunes the NN using two reference evaluations, we again use the same reference GPU profiles as for other baselines. Finally, **Static policy** is a simple, naive baseline, in which no performance predictions are made. The GPU recommendation is always the same: to create a fixed number of pods with a certain GPU profile. We have considered a broad range of static policies and present the one which achieved the highest S/O score: 4 pods running on 1×A100 GPU. Unknown HPs of baseline methods were determined through the same leave-one-LLM-out cross-validation procedure as for LLM-Pilot.

Recommendation results: Fig. 8 summarizes the recommendation scores achieved with LLM-Pilot and the baseline methods. Based on these results, we can draw a number of conclusions. Firstly, LLM-Pilot achieves good results – its recommendations are successful in 80% of cases and have average overspend of less than 20%, outperforming all other methods in terms of the S/O score. The static policy can be considered a high-risk, high-reward solution. It only succeeds in 50% of cases but it achieves an excellent overspend when it makes a successful recommendation, outperforming all other baselines in terms of the S/O score. PARIS and Selecta achieve the same or similar success rate as LLM-Pilot but have higher

average overspend, and require making additional performance measurements using the reference GPU profiles. RF, which depicts the performance of PARIS without the reference performance measurements, is significantly worse in terms of all three evaluation metrics. Finally, PerfNet models achieve good overspend scores but their success rate is the worst of all state-of-the-art solutions. On the other hand, Morphling achieves higher success rate than other NN-based methods thanks to performing reference performance measurements but is worse in terms of overspend.

Overall, the experimental results indicate that LLM-Pilot can successfully recommend the most cost-effective GPU profile for previously unseen LLMs, and outperforms all state-of-the-art methods. Its GPU recommendations satisfy the performance requirements 33% more frequently on average, thanks to ensuring that the performance is most accurately predicted in the neighborhood of the latency constraints. At the same time, it recommends GPU profiles which are on average 60% cheaper than state-of-the-art.

VI. Related Works

A. LLM traces

There are multiple publicly available collections of LLM traces, in some cases consisting of thousands or millions of LLM inference requests. While some datasets consist of real user inputs of inference requests and resulting outputs [23], [28], [59], they do not take into account any additional request parameters. There is also a range of inference request collections generated synthetically: [20] generated by a group of volunteers, [22], [25] with handcrafted system prompts, and [44] generated by GPT 3.5 using Self-Instruct [52]. None of the synthetic collections represents a real-life distribution of user requests, and therefore could not be used in this work for realistic workload generation.

B. Workload generators and benchmarking tools

There is a number of related works on benchmarking LLM inference services, which we have compared in Table IV. Many of them measure the LLM inference performance under a very simple workload of requests not based on real LLM usage [12], [34], [46]. As we have argued in Sec. III-B, a realistic and varied set of inference requests is necessary to perform meaningful performance measurements. Other benchmarking tools ensure that the inference requests are realistic by drawing random samples from existing trace collections [9], [19], [37]. However, none of the related benchmarking tools optimizes the maximum batch weight, which we have found to have big influence on the performance.

C. LLM performance datasets

While various benchmarking tools publish some of their benchmarking results, none of the existing datasets aggregates performance measurements of many LLM services deployed on a variety of GPUs (see Table IV). Optimum's [12] LLM-Perf leaderboard [32] includes benchmarking results of LLMs but only includes 2 GPUs, while MLPerf [37] collected data across many GPUs but only two LLMs are currently included. Other related benchmarking tools [9], [19], [34], [46] also published small collections of LLM performance data.

TABLE IV: Comparison of LLM-Pilot's performance characterization tool and related LLM benchmarking tools, including LLM performance datasets that they released publicly.

Comparison criterion	Workload based on real data	Maximum batch weight tuning	LLM performance data released publicly	
			Number of LLMs	Number of GPUs
Optimum [12]	×	×	34	2
LLMPerf [34]	×	×	3	1
Inference benchmark [46]	×	×	1	1
Fleece [9]	✓	×	5	5
vLLM [19]	✓	×	3	2
MLPerf [37]	✓	×	2	10
LLM-Pilot (ours)	✓	✓	10	14

D. LLM performance prediction

To the best of our knowledge, there are no prior works that predict the inference performance of LLMs across a variety of hardware platforms. The works that are most closely related to our work predict the runtimes of other types of machine learning applications. Many of these methods [18], [49]–[51], [55] have been used as baselines in this work and have been discussed in detail in Sec. V-C. Other works related to LLM-Pilot include [16] and [54] which predict the training runtime or inference latency of NNs across GPUs or runtime and resource configurations, while [56] and [15] predict the quality of outputs of machine learning models on downstream tasks.

VII. Conclusion and Next Steps

In this work we have presented LLM-Pilot, a system that can perform realistic and optimized benchmarking of LLM inference services across different GPUs. In addition, LLM-Pilot can recommend which GPU will meet performance requirements in the most cost-effective way for a previously unseen LLM, achieving on average 33% higher success rate and 60% lower cost compared to state-of-the-art methods. As a next step, we intend to extend LLM-Pilot to cover the multi-tenancy scenario, in which multiple users compete to deploy LLM inference services on the same hardware resources.

Acknowledgement

We would like to express our gratitude to our colleagues in IBM Research: Nick Hill, for sharing his technical insights into LLM serving using TGIS, and Burkhard Ringlein, for his valuable input during the experimental work and the process of writing this publication.

References

[1] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. Gpt-neox-20b: An open-source autoregressive language model, 2022.

[2] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models, 2022.

[3] L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. *Classification and Regression Trees*. Taylor & Francis, 1984.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[5] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models, 2022.

[7] The Kubernetes Community. Kubernetes. https://kubernetes.io/, accessed: 28.03.2024.

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[9] Zhongjing Wei et al. Fleece-benchmark. https://github.com/CoLearn-Dev/fleece-benchmark, accessed: 04.03.2024.

[10] Nicolas Hug. Surprise: A python library for recommender systems. *Journal of Open Source Software*, 5(52):2174, 2020.

[11] huggingface. Large Language Model Text Generation Inference. https://github.com/huggingface/text-generation-inference, accessed: 28.03.2024.

[12] huggingface. Optimum benchmark. https://github.com/huggingface/optimum-benchmark, accessed: 04.03.2024.

[13] IBM. Mpt-7b-instruct2. https://huggingface.co/ibm/mpt-7b-instruct2, accessed: 21.03.2024.

[14] IBM. Text Generation Inference Server. https://github.com/IBM/text-generation-inference, accessed: 28.03.2024.

[15] Ganesh Jawahar, Muhammad Abdul-Mageed, Laks V. S. Lakshmanan, and Dujian Ding. Llm performance predictors are good initializers for architecture search, 2023.

[16] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. Predicting the computational cost of deep learning models. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3873–3882, 2018.

[17] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[18] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 759–773, Boston, MA, July 2018. USENIX Association.

[19] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[20] Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi-Rui Tam, Keith Stevens, Abdullah Barhoum, Nguyen Minh Duc, Oliver Stanley, Richárd Nagyfi, Shahul ES, Sameer Suri, David Glushkov, Arnav Dantuluri, Andrew Maguire, Christoph Schuhmann, Huu Nguyen, and Alexander Mattick. Openassistant conversations – democratizing large language model alignment, 2023.

[21] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! 2023.

[22] Wing Lian, Bleys Goodson, Eugene Pentland, Austin Cook, Chanvichet Vong, and "Teknium". Openorca: An open dataset of gpt augmented flan reasoning traces. https://https://huggingface.co/Open-Orca/OpenOrca, 2023.

[23] Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. The flan collection: Designing data and methods for effective instruction tuning, 2023.

[24] Niklas Muennighoff, Thomas Wang, Lintang Sutawika, Adam Roberts, Stella Biderman, Teven Le Scao, M Saiful Bari, Sheng Shen, Zheng-Xin Yong, Hailey Schoelkopf, et al. Crosslingual generalization through multitask finetuning. *arXiv preprint arXiv:2211.01786*, 2022.

[25] Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. Orca: Progressive learning from complex explanation traces of gpt-4, 2023.

[26] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm, 2021.

[27] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models, 2024.

[28] Vercel NextJS and Upstash. Sharegpt. https://sharegpt.com/, accessed: 04.03.2024.

[29] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint*, 2023.

[30] NVIDIA Corporation. Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution. https://github.com/triton-inference-server/server, accessed: 28.03.2024.

[31] Emmanuel Ohiri. How ai is contributing to the gpu shortage. https://www.cudocompute.com/blog/gpu-supply-shortage-due-to-ai-needs/, accessed: 04.03.2024.

[32] Optimum. Llm-perf leaderboard. https://huggingface.co/spaces/optimum/llm-perf-leaderboard, accessed: 11.03.2024.

[33] Pankesh Patel, Ajith Ranabahu, and Amit Sheth. Service level agreement in cloud computing. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) Cloud Computing workshop*, 2009.

[34] Ray Project. Llmperf. https://github.com/ray-project/llmperf, accessed: 04.03.2024.

[35] Marco Ramponi. Why language models became large language models and the hurdles in developing llm-based applications. https://www.assemblyai.com/blog/why-language-models-became-large-language-models, accessed: 04.03.2024.

[36] Red Hat. Red Hat Openshift. https://www.redhat.com/en/technologies/cloud-computing/openshift, accessed: 28.03.2024.

[37] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark, 2019.

[38] Run:ai. How to achieve almost 2x inference throughput and reduce latency leveraging multi-gpu setup. https://www.run.ai/blog/achieve-2x-inference-throughput-reduce-latency-leveraging-multi-gpu, accessed: 28.03.2024.

[39] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. From words to watts: Benchmarking the energy costs of large language model inference, 2023.

[40] Julien Simon. Large language models: A new moore's law? https://huggingface.co/blog/large-language-models, accessed: 04.03.2024.

[41] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[42] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative llm serving, 2024.

[43] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp, 2019.

[44] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[45] Yi Tay. A new open source flan 20b with ul2. https://www.yitay.net/blog/flan-ul2-20b, accessed: 21.03.2024.

[46] TensorChord. inference-benchmark. https://github.com/tensorchord/inference-benchmark/, accessed: 04.03.2024.

[47] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[49] Chuan-Chi Wang, Ying-Chiao Liao, Ming-Chang Kao, Wen-Yew Liang, and Shih-Hao Hung. Perfnet: Platform-aware performance modeling for deep neural networks. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, RACS '20, page 90–95, New York, NY, USA, 2020. Association for Computing Machinery.

[50] Chuan-Chi Wang, Ying-Chiao Liao, Ming-Chang Kao, Wen-Yew Liang, and Shih-Hao Hung. Toward accurate platform-aware performance modeling for deep neural networks. *SIGAPP Appl. Comput. Rev.*, 21(1):50–61, jul 2021.

[51] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. Morphling: Fast, near-optimal auto-configuration for cloud-native model serving. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 639–653, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language model with self generated instructions, 2022.

[53] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models, 2022.

[54] Yuewen Wu, Heng Wu, Diaohan Luo, Yuanjia Xu, Yi Hu, Wenbo Zhang, and Hua Zhong. Serving unseen deep learning models with near-optimal configurations: a fast adaptive search approach. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 461–476, New York, NY, USA, 2022. Association for Computing Machinery.

[55] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 452–465, New York, NY, USA, 2017. Association for Computing Machinery.

[56] Qinyuan Ye, Harvey Yiyun Fu, Xiang Ren, and Robin Jia. How predictable are large language model capabilities? a case study on big-bench, 2023.

[57] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[58] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2023.

[59] 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. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

[60] Yinmin Zhong, Junda Chen, Shengyu Liu, Yibo Zhu, Xin Jin, and Hao Zhang. Throughput is not all you need: Maximizing goodput in llm serving using prefill-decode disaggregation. https://hao-ai-lab.github.io/blogs/distserve/, accessed: 25.03.2024.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

The focus of this work is the performance of Large Language Model (LLM) inference services and how it is impacted by the choice of the type and number of GPUs (jointly referred to as the *GPU profile*) onto which the service has been deployed. The product of this work is LLM-Pilot – a system for characterizing and predicting performance of LLM inference services on various GPU profiles. LLM-Pilot consists of two main components: the performance characterization tool and the GPU recommendation tool. The contributions of this work are as follows:

- C_1 A performance characterization tool which benchmarks LLM inference services while ensuring that the services are optimized for the specific hardware, and subjects the services to a realistic workload of requests based on a large collection of production traces.
- C_2 A performance dataset comparing the inference performance of many LLMs running on a variety of GPU profiles and subject to a varying load of requests, collected using the performance characterization tool of LLM-Pilot (contribution C_1).
- C_3 A GPU recommendation tool of LLM-Pilot, which can be used to predict the unknown performance of a previously unseen LLM on a variety of GPU profiles and recommend which one will satisfy the performance requirements in the most cost-effective way.

B. Computational Artifacts

To support the work's contributions, we have supplemented the work with two computational artifacts: A_1 and A_2. Both artifacts are stored together in a public repository:
https://github.com/IBM/LLM-performance-prediction
DOI: 10.5281/zenodo.12569208.

Artifact ID	Contributions Supported	Reproduced Paper Elements
A_1	C_2	Figure 7a–c
A_2	C_3	Figure 8

As presented in the table above, the artifacts allow reproduction of the results presented in the paper related to contributions C_2 and C_3. The computational artifact related to the performance characterization tool (contribution C_1) is planned to be released at a later date due to its dependency on other projects developed in our organization. However, artifact A_1 presents the product of contribution C_1 – the performance dataset that we have collected using the performance characterization tool of LLM-Pilot. Moreover, the main manuscript describes the performance characterization process in detail.

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

In artifact A_1, we publish the full collection of performance characterization data collected across many LLMs and GPU profiles, which is contribution C_2 of this work. Additionally, this artifact comprises code for preprocessing the detailed performance measurements into an aggregate dataset, and for producing plots which visualize the relationship between various performance metrics of LLM inference services running to a variety of GPUs under varying load. The artifact supports contribution C_2 by releasing the valuable performance data collected in this work, and supports the analysis of the performance dataset conducted in the main manuscript. Most notably, it reproduces Fig. 7a–c presented in the work for one selected LLM, and allows generating similar plots for other LLMs present in the performance characterization dataset C_2.

Expected Results

The expected outcomes of artifact A_1 are:

1) aggregate dataset with preprocessed performance measurements collected across many LLMs and GPU profiles,
2) example plots depicting the relation between latency metrics and throughput of the inference service for a selected LLM running on a variety of GPU profiles, reproducing Fig. 7a–c from the manuscript.

The output of the artifact therefore (1) provides an insight into how the raw performance data ought to be processed, (2) presents an easier to analyze, aggregate dataset with averaged performance measurements, and (3) provides a visual representation of the performance of LLM inference services running on a variety of GPUs under a varying load.

Expected Reproduction Time (in Minutes)

The expected reproduction time of Artifact A_1 is approximately 20min:

1) artifact setup: 15min (build the docker image, including installing all libraries),
2) artifact execution: 3min (load all performance data files, process and aggregate performance measurements),
3) artifact analysis: 2min (plot the performance measurements for a selected LLM).

Artifact Setup (incl. Inputs)

Hardware: The experiments presented in the artifact have been executed using a machine with one 14-core Intel® Core™i9-10940X CPU @ 3.30GHz, 125GB memory and two NVIDIA GeForce RTX 3070 GPUs. We note that access to GPUs is not necessary for artifact A_1.

Software: The artifact was run on a machine with Ubuntu 22.04.4 (LTS) operating system and uses the following software:

- docker version 24.0.5, URL: https://docs.docker.com/,
- python version 3.10.12, URL: https://www.python.org/,
- jupyter notebook version 6.5.2, URL: https://jupyter.org/,
- traitlets version 5.9.0, URL: https://github.com/ipython/traitlets/,
- numpy version 1.23.5, URL: https://numpy.org/,
- pandas version 1.5.3, URL: https://pandas.pydata.org/,
- matplotlib version 3.7.0, URL: https://matplotlib.org/,
- IPython version 8.10.0, URL: https://ipython.org/,

Datasets / Inputs: The dataset used as input is provided as part of the artifact, and is one of the contributions of this work.

Installation and Deployment: We have provided instructions to build a Docker image with all required software to ensure easy environment setup. Before executing the artifact, the user builds the docker image and runs Jupyter notebook in the container using the built image. The artifact is then executed by running all cells of the provided Jupyter notebook.

Artifact Execution

The artifact execution stage consists of two tasks performed sequentially $T_1 \rightarrow T_2$.

In the first task T_1 the provided code reads all 528 files containing raw performance measurements from individual experimental cases, and adds them as new entries to the aggregate dataset. Each experimental case is defined as a distinct combination of LLM, GPU profile, and number of concurrent users simultaneously sending inference requests. After aggregating all data, in task T_2 the script calculates averaged performance metrics for all experimental cases. The final product is a dataset with 528 rows, one for each combination of LLM, GPU profile and number of concurrent users for which performance data was collected. Each row includes the LLM name, GPU profile, number of concurrent users, median time to first token (TTFT) latency, median inter-token latency (ITL), total throughput, and total throughput divided by the cost of the respective GPU profile. The script displays the final dataset and saves it to a file.

Artifact Analysis (incl. Outputs)

In the artifact analysis stage, the aggregate dataset is used to generate plots visualising the relationship between various performance metrics for a selected LLM. The script generates three plots which visualize the following relations between performance metrics: TTFT vs. throughput, ITL vs. throughput, and ITL vs. throughput divided by cost. Each plot consists of 9 curves, each one associated with a different GPU profile, and spanning various numbers of concurrent users. To generate each point on each curve, the script identifies the row of the aggregate dataset which describes that particular experimental case, and reads the numbers in appropriate columns: median TTFT or median ITL, and throughput or throughput divided by cost. The plots are displayed and saved to files.

B. Computational Artifact A_2

Relation To Contributions

Artifact A_2 comprises the code of the GPU recommendation tool of LLM-Pilot (contribution C_3), as well as our implementation of all baselines considered in the work. Internally, the GPU recommendation tool and all other methods use the performance characterization dataset (contribution C_2) for training, validation and testing. Moreover, artifact A_2 includes the code which uses the performance predictions of LLM-Pilot and all baselines to recommend the most cost-effective GPU for a given LLM inference service under performance requirements, and extracts final evaluation scores of each analyzed method.

Expected Results

Artifact A_2 serves as a detailed demonstration of our evaluation of the GPU recommendation tool against the baselines. The expected outcomes of this artifact are:

1) all performance predictions made for all LLMs by LLM-Pilot and all baselines,
2) GPU recommendation scores achieved by LLM-Pilot and all baselines,
3) reproduction of Fig. 8 from the manuscript, visually presenting the GPU recommendation scores achieved by LLM-Pilot and the baselines.

Expected Reproduction Time (in Minutes)

The total expected reproduction time of Artifact A_2 is 75min, divided into steps as follows:

1) artifact setup: 15min (building the docker image, including installing all libraries - unnecessary if environment has been previously set up to execute artifact A_1),
2) artifact evaluation: 55min (for LLM-Pilot and all baselines: tune the hyperparameters, train the performance model, and make perfomance predictions),
3) artifact analysis: 5min (use LLM-Pilot and all baselines to recommend the most cost-effective GPU; calculate their GPU recommendation evaluation scores; generate a plot of all evaluation scores to reproduce Fig. 8).

Artifact Setup (incl. Inputs)

Hardware: The experiments presented in the artifact have been executed using one 14-core Intel® Core™ i9-10940X CPU @ 3.30GHz, 125GB memory and two NVIDIA GeForce RTX 3070 GPUs.

Software: The artifact was run on a machine with Ubuntu 22.04.4 (LTS) operating system and uses the following software:

- CUDA version 12.2, URL: https://developer.nvidia.com/cuda-toolkit,
- docker version 24.0.5, URL: https://docs.docker.com/,
- python version 3.10.12, URL: https://www.python.org/,
- jupyter notebook version 6.5.2, URL: https://jupyter.org/,
- traitlets version 5.9.0, URL: https://github.com/ipython/traitlets/,

- numpy version 1.23.5, URL: https://numpy.org/,
- pandas version 1.5.3, URL: https://pandas.pydata.org/,
- matplotlib version 3.7.0, URL: https://matplotlib.org/,
- scikit-learn version 1.4.1.post1, URL: https://scikit-learn.org/stable/,
- surprise version 1.1.3, URL: https://surpriselib.com/,
- xgboost version 2.0.2, URL: https://xgboost.readthedocs.io/en/stable/,
- IPython version 8.10.0, URL: https://ipython.org/,
- keras version 3.1.1, URL: https://keras.io/,
- pytorch version 2.16.1, URL: https://pytorch.org/,
- tersorflow version 2.2.1, URL: https://www.tensorflow.org/

Datasets / Inputs: The dataset used as input to the artifact is the preprocessed dataset produced as output by artifact A_1. In order to execute artifact A_2, one must first run artifact A_1 to produce the preprocessed data file.

Installation and Deployment: Together with the artifact, we have provided instructions to build a Docker image with all required software to ensure easy environment setup. Before executing the artifact, the user builds the docker image and runs Jupyter notebook in the container using the built image. The artifact is then executed by running all cells in the provided Jupyter notebook.

Artifact Execution

The artifact execution stage consists of the following tasks:

1) T_1: load the performance characterization dataset C_2, augment the dataset with features describing the LLMs and GPU profiles, and encode categorical features,
2) T_2: use LLM-Pilot's GPU recommendation tool to make performance predictions for each LLM in the performance characterization dataset:
   a) Tune the hyperparameters separately for each LLM,
   b) Train a prediction model for each LLM using the respective cross-validated hyperparameter configuration,
   c) Make performance predictions for each LLM and save them to a file,
3) T_3–T_8: perform the same steps as in T_2 for the baselines (PARIS, Random Forest, Selecta, PerfNet, PerfNetV2, and Morphling). In case of PerfNet and PerfNetV2, all hyperparameters were clearly stated in the original publications and therefore hyperparameter tuning was not necessary. Moreover, in order to reduce the total artifact runtime, we omit the hyperparameter tuning process of Morphling, as it was the most time-consuming task in the artifact. Instead, for each LLM we use the hyperparameter configuration that has been selected in our much longer experimentation.

Task T_1 must be executed first, while tasks T_2–T_8 are independent: $T_1 \rightarrow \{T_2, T_3, \ldots, T_8\}$. The provided scripts execute tasks T_2–T_8 sequentially.

In each task T_2–T_5 the hyperparameters of the respective method are tuned in a robust way by using a nested leave-one-LLM-out cross-validation procedure. For each test LLM, the hyperparameter configuration is selected based on the average validation score it achieved across all folds, i.e., across all training LLMs.

Artifact Analysis (incl. Outputs)

After LLM-Pilot and all baselines have been tuned and trained and their predictions have been saved, the script performs the analysis stage. Based on the performance predictions of each method, the script identifies which GPU should satisfy the performance requirements of each LLM in the most cost-effective way. After that, the script calculates the three evaluation scores that have been defined in this work to evaluate GPU recommendation capabilities of LLM-Pilot and the baselines: the success rate, the relative overspend and the S/O score. The scores for all methods are visualized as a scatter plot. Each method is represented by a single marker whose coordinates indicate the achieved success rate and relative overspend, while a label next to the marker indicates the achieved S/O score. The plot is displayed and saved to a file.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

We have tested the execution of the artifact using Chameleon Cloud resources with the following setup:

- Host: c11-08:
  - CPU: 2 x Intel® Xeon® E5-2670 v3 @ 2.30GHz,
  - GPU: 2 x NVIDIA P100,
  - RAM: 128GB,
  - threads: 48,
- Image: CC-Ubuntu22.04-CUDA.

As the artifact involves running a Jupyter notebook, we recommend forwarding port 8889 when connecting via ssh to Chameleon Cloud resources or any other remote machine:

```
ssh -L 8889:localhost:8889 [username]@[remote IP address]
```

To simplify the setup and verification process, in the artifact repository we provide a Dockerfile and a list of all requirements needed for environment setup. Therefore, it is first necessary to ensure that docker (version 24.0.5) and the nvidia-container-toolkit library are installed on the machine used for artifact evaluation. For convenience, our repository contains a script `docker/install_docker.sh` which executes all commands needed for these installations. After that, all other dependencies will be ensured by the Dockerfile. Therefore, to prepare the environment for artifact execution the user needs to perform the following steps:

1) clone the artifact repository and enter its directory,

```
git clone \
  https://github.com/IBM/LLM-performance-prediction.git
cd LLM-performance-prediction
```

or download the archive via DOI and enter its directory,

```
sudo apt install unzip
wget https://zenodo.org/records/12569208/files/\
IBM/LLM-performance-prediction-v1.0.0.zip
unzip LLM-performance-prediction-v1.0.0.zip
cd IBM-LLM-performance-prediction-3c66483
```

2) if necessary, run the script installing docker and nvidia-container-toolkit,

```
chmod u+x docker/install_docker.sh
./docker/install_docker.sh
```

3) build a docker image according to the Dockerfile provided in the repository,

```
sudo docker build -f docker/Dockerfile . -t llm-pilot
```

4) run a docker container using the built image (which will automatically start Jupyter notebook in the container).

```
sudo docker run -it \
  --gpus all \
  -v ./preprocess_data:/app/preprocess_data \
  -v ./predict_performance:/app/predict_performance \
  -p 8889:8889 \
  llm-pilot
```

The above steps are also listed and described in detail in the README of the artifact repository. At the end of the last step, Jupyter notebook will display the address needed to connect to it. Copy the provided address and paste it in your browser.

Artifact Execution

Executing the artifact involves running all cells in a Jupyter notebook `Preprocess_data.ipynb` provided as part of the artifact in the `preprocess_data` directory. Below, we list the tasks performed by each cell in the notebook.

1) Import necessary libraries.
2) Read all files containing raw performance measurements from individual experimental cases (defined as distinct combinations of LLM, GPU profile, and number of concurrent users simultaneously sending inference requests to that LLM hosted on that GPU profile), preprocess them and combine relevant metrics in an aggregate dataset `performance_characterization_data.csv`.
3) Define functions and style settings in preparation for plotting example performance results.
4) Generate plots presenting the relation between performance metrics: median time to first token (TTFT), median inter-token latency (ITL), throughput and throughput divided by cost, for a selected LLM.

Artifact Analysis (incl. Outputs)

The final outcome of the artifact is the series of plots presenting the relation between various performance metrics for a selected LLM, reproducing Fig. 7a–c from the main manuscript.. The plots are generated by reading the throughput and median latency metrics from the preprocessed aggregate dataset, and plotting one curve for each GPU profile present in the dataset for that LLM. For each GPU profile, the curve spans across a range of numbers of concurrent users included in the dataset.

To simplify the verification of the artifact outcome, in directory `preprocess_data/expected_results` we provide copies of figures that are expected as output. The expected plots are identical to Fig. 7a–c from the manuscript, and should also be identical to the plots generated by executing the artifact.

B. Computational Artifact A_2

Artifact Setup (incl. Inputs)

We have tested the execution of the artifact using Chameleon Cloud resources with the same setup as for artifact A_1, namely:

- Host: c11-08:
  - CPU: 2 x Intel® Xeon® E5-2670 v3 @ 2.30GHz,
  - GPU: 2 x NVIDIA P100,
  - RAM: 128GB,
  - threads: 48,
- Image: CC-Ubuntu22.04-CUDA.

Because the hardware of this host is weaker than the machine originally used for our experiments, the expected runtime of this artifact on Chameleon Cloud host c11-08 is 110min, approx. 50% slower than stated in Artifact Description.

If the artifact is executed on a remote machine (on Chameleon Cloud or any other remote host), port 8889 must be forwarded when connecting to the machine via ssh:

```
ssh -L 8889:localhost:8889 [username]@[remote IP address]
```

Before executing artifact A_2, it is necessary to execute artifact A_1 in order to preprocess raw performance data. Both artifacts require the same environment and can be run in the same docker container. Therefore, the setup for this artifact is the same as for artifact A_1:

1) clone the artifact repository and enter its directory,

```
git clone \
  https://github.com/IBM/LLM-performance-prediction.git
cd LLM-performance-prediction
```

or download the archive via DOI and enter its directory,

```
sudo apt install unzip
wget https://zenodo.org/records/12569208/files/\
IBM/LLM-performance-prediction-v1.0.0.zip
unzip LLM-performance-prediction-v1.0.0.zip
cd IBM-LLM-performance-prediction-3c66483
```

2) if necessary, run the script installing docker and nvidia-container-toolkit,

```
chmod u+x docker/install_docker.sh
./docker/install_docker.sh
```

3) build a docker image according to the Dockerfile provided in the repository,

```
sudo docker build -f docker/Dockerfile . -t llm-pilot
```

4) run a docker container using the built image (which will automatically start Jupyter notebook in the container).

```
sudo docker run -it \
  --gpus all \
  -v ./preprocess_data:/app/preprocess_data \
  -v ./predict_performance:/app/predict_performance \
  -p 8889:8889 \
  llm-pilot
```

When container starts, copy the address provided by Jupyter notebook and paste it in your browser. Then, execute artifact A_1 by running all cells in notebook `preprocess_data/Preprocess_data.ipynb`.

Artifact Execution

The artifact workflow is prepared as a notebook `Predict_LLM_performance.ipynb` available in the `predict_performance` directory of the artifact repository. All that is needed to execute the artifact is to run all cells of the provided notebook. Below we detail the tasks performed throughout the artifact workflow.

1) Import all necessary libraries, set various display options (cell 1).
2) Load and encode preprocessed performance data (cells 2–4).
3) Define common utilities, e.g., custom loss functions and cross-validation procedure (cell 5).
4) Train and make predictions using LLM-Pilot (cell 6).
5) Train and make predictions using baseline solutions (cells 7–14).
6) Analyze a range of naive static policy baselines and select the best one (cell 15).
7) Make GPU recommendations based on predictions made by LLM-Pilot and all baselines, calculate their evaluation metrics (cell 16).
8) Generate a figure visually comparing the evaluation metrics achieved by all analyzed methods (cell 17).

Artifact Analysis (incl. Outputs)

The outcome of the artifact is a figure similar to Fig. 8 in the main manuscript which reproduces the main conclusions and observations made in this work about LLM-Pilot's GPU recommendation tool. To simplify the verification process, in directory `predict_performance/expected_results` we provide files with expected predictions made by LLM-Pilot and each baseline, as well as the expected figure presenting their evaluation metrics.

We note that some baselines used in this work (PerfNet, PerfNetV2 and Morphling) were implemented using the *keras* library, they are sensitive to hardware changes. As a result, the behavior of these baselines may differ slightly from the results presented in the manuscript. Despite this, the general conclusions regarding LLM-Pilot's performance relative to the baselines remain valid and true.