Demystifying deep learning in predictive monitoring for cloud-native SLOs

Andrea Morichetta
Víctor Casamayor Pujol
Stefan Nastic
Distributed Systems Group, TU Wien
Vienna, Austria
lastname@dsg.tuwien.ac.at

Thomas Pusztai
Philipp Raith
Schahram Dustdar
Distributed Systems Group, TU Wien
Vienna, Austria
lastname@dsg.tuwien.ac.at

Deepak Vij
Ying Xiong
Zhaobo Zhang
Futurewei Technologies, Inc.
Santa Clara, CA, USA
firstname.lastname@futurewei.com

2023 IEEE 16th International Conference on Cloud Computing (CLOUD) | 979-8-3503-0481-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/CLOUD60044.2023.00013

***Abstract*—The complexity inherent in managing cloud computing systems calls for novel solutions that can effectively enforce high-level Service Level Objectives (SLOs) promptly. Unfortunately, most of the current SLO management solutions rely on reactive approaches, i.e., correcting SLO violations only after they have occurred. Further, the few methods that explore predictive techniques to prevent SLO violations focus solely on forecasting low-level system metrics, such as CPU and Memory utilization. Although valid in some cases, these metrics do not necessarily provide clear and actionable insights into application behavior. This paper presents a novel approach that directly predicts high-level SLOs using low-level system metrics. We target this goal by training and optimizing two state-of-the-art neural network models, a Short-Term Long Memory – LSTM-, and a Transformer-based model. Our models provide actionable insights into application behavior by establishing proper connections between the evolution of low-level workload-related metrics and the high-level SLOs. We demonstrate our approach to selecting and preparing the data. We show in practice how to optimize LSTM and Transformer by targeting efficiency as a high-level SLO metric and performing a comparative analysis. We show how these models behave when the input workloads come from different distributions. Consequently, we demonstrate their ability to generalize in heterogeneous systems. Finally, we operationalize our two models by integrating them into the Polaris framework we have been developing to enable a performance-driven SLO-native approach to Cloud computing.**

***Index Terms*—workload prediction; neural networks; cloud; LSTM; Transformers**

I. Introduction

বিস্তার

The proliferation of Cloud vendors and the increase in customers joining Cloud platforms calls for developing more effective and better-tailored management solutions for next-generation cloud applications. Notably, it is crucial to guarantee compliance with the Service-Level Objectives (SLOs), representing targets within contracts between the provider and its customers. Many current methodologies [1] are based on reactive approaches, i.e., promptly countering SLO violations only after they happen. Unfortunately, these approaches usually do not provide adequate guarantees in highly dynamic scenarios, typically found in the Cloud. Therefore, in recent years, we have witnessed the consolidation of works that predict the workload behavior to allow elastic management of Cloud systems and resources [2]. Still, most proposed solutions focus solely on analyzing and predicting low-level metrics, like CPU usage [3], achieving resource-level elasticity. However, resources represent just one of the three dimensions that govern an application elasticity, the others being cost (how much a customer is willing to pay for a service) and quality (how well the application performs its operations) [4]. In this regard, managing applications only through the prediction of low-level metrics do not provide a complete vision of its state, nor does it provide actionable insights, which can be used to set optimal elastic scaling strategies [5]. Hence, focusing the effort towards the usage of high-level SLO leads to (i) model more complex constraints in terms of resource usages, i.e., instead of talking about CPU usage, we can define the CPU usage efficiency. Furthermore, it helps to (ii) include cost and quality elasticity dimensions in the SLO-based Cloud management system. That means that we could consider cost-efficiency, as described in [5], as a constraint for the system. Finally, (iii) involve other system stakeholders in the deployment phase, i.e., machine learning engineers can set accuracy constraints to their prediction models and map them to a high-level SLO. All these characteristics create stronger bonds between the application and its underlying infrastructure, which are now weak and poorly tailored.

To bridge this gap and foster the usage of high-level SLO, we shift the focus moving the emphasis on *directly predicting high-level SLO metrics*. We introduce novel *neural network-based prediction methods* to predict high-level SLO metrics with available low-level metrics obtained through system monitoring. This way, we enable the models to develop the relationship between the low-level metrics and the target high-level SLO metric. We use two different models to gain insights about their suitability for the task and a clearer perspective on the needs for predicting high-level SLO metric. Nevertheless, to enable high-level SLO metric prediction with the models mentioned above, we need to *define the steps to perform in the data pipeline*, like data selection, feature engineering, model design, model optimization, and performance analysis. Furthermore, we provide *two levels of model analysis*. First, we estimate the ability of the two models to predict high-level SLO values by looking at *out-of-sample* (OOS) data. Out-of-sample data is typically a test set extracted from the same workload time series used for training. The second level considers previously unseen workload data, which stems from different jobs and can have noticeably different behavior. This type of data is called *out-of-distribution* (OOD). Usually, most available studies stop at the first level [6], whereas we dig deep. This analysis is particularly relevant as it allows us

2159-6190/23/$31.00 ©2023 IEEE
DOI 10.1109/CLOUD60044.2023.00013

to evaluate our models' performance over new workloads, investigating models' generalization, which is one of the main challenges in modern ML research [7] and key for cloud management at a scale. Finally, we *operationalize the developed prediction models* and put them at *"developers' fingertips"*. The ease of creating ML algorithms makes it appear that they can easily integrate with any complex solution. However, there are many aspects to consider [8]. Previously, we introduced a new framework, Polaris [9]–[11], which enables straightforward high-level SLO management and workload scheduling. We integrate our novel ML models into Polaris, ensuring the separation of concerns and leaving the application administrator with a view transparent to the prediction algorithms. We make our *open source* code publicly available.[1] In summary, our contributions are as follows:

- We present our methodology for designing, optimizing, and building models for ingesting low-level metrics and predicting high-level SLO metrics. We provide insights from each phase, letting the community leverage our experience.
- We evaluate and compare the proposed models on out-of-distribution data. We develop insights on models' robustness to new data and models' generalizations capabilities, and we show the specific limits and potentials for the LSTM and Transformer models.
- We integrate the final models in our Polaris framework for cloud management. We enhance the development of SLO policies for developers.

The rest of the paper is organized as follows. Section II describes the most relevant steps of the methodology.Section III offers an in-depth evaluation of our proposed models.Then, Section IV details the integration of models in the Polaris framework. Section V positions our work in the context of previous research. Finally, Section VI concludes the paper.

II. Methodology

Our work aims at building a system to forecast high-level SLO metrics, looking at low-level metrics, such as resources usage. The process involves multiple steps: (i) *Obtaining the data (Subsection II-A)*, i.e., analyzing the available resources and targets; (ii) *Features engineering (Subsection II-B)*, i.e., data preprocessing such as normalization, aggregation, and defining high-level metrics; (iii) *Prediction models design (Subsection II-C)*, that is, inspecting Long Short-Memory approach and Transformers for time series forecasting; lastly, (iv) *Optimizing the models (Subsection II-D)* performing hyperparameter optimization to select the best models' configuration. Figure 1 shows the process.

A. Obtaining the data

It is fundamental to set requirements for choosing the most appropriate dataset: it shall report from existing infrastructure, avoiding synthetic data; last more than one week; guarantee accurate visibility on the workload; provide information on the infrastructure; provide accurate documentation; avoid proprietary solutions, allowing experiment repeatability. Hence, we select and analyze the four databases that best fulfill these requirements from approximately twenty open-sourced collections.

The *Google Cluster Workload Datasets* contains four weeks-long measurements collected in 2011 from its Borg cluster. The dataset contains a range of Google applications sampled every five minutes [12]. The *Azure Dataset* comprises first-party (i.e., internal VMs, infrastructure) and third-party (i.e., communication, gaming, and more) workloads collected in 2019. The sampling rate is every five minutes [13]. The *TUDelft - Bitbrains* traces report the execution of business-critical workloads. They sample the metrics every 5 minutes. Finally, the *Alibaba* dataset consists of eight days of recording of 4 000 machines, 9 000 online services, and 4M batch jobs with static and runtime information, extracted every five minutes [14]. Table I summarizes their characteristics. We can deduce that, while reasonable for experimenting with forecasting, the Azure dataset has too coarse features and a not active community. Bitbrains offers various job types but is too small for our analysis. Alibaba, which may represent a notable collection for model benchmarking, has cluttered documentation. Finally, the Google 2019 version, despite being new and rich, is vast; plus, handling data retrieval and processing through BigQuery, which requires a subscription, limits its use. For these reasons, we choose Google Cluster Data in its 2011 version. Despite being 12 years old, the data still holds great significance. The current underlying infrastructure has been improved but is still built on top of the considered one, and the applications running at the Borg cluster, i.e., Google services, have evolved. However, we can consider them still similar for the sake of this paper.

Takeaways *Considering multiple requirements for dataset selection is fundamental to developing an in-depth analysis of high-level SLO metric prediction. Open-source, well-documented data guarantees reliability and research repeatability. Therefore, we select the 2011 Google Cluster Data as it satisfies the major data requirements.*

B. Features engineering

1) Selecting data pool: In the Google cluster data [15], the work arrives at a node (referred to as a "cell" in Borg) in the form of a job. A job has multiple tasks. Usually, all tasks within a job execute the same binary, sharing the same options and requests. Hence, different task categories run as separate jobs. Further, each task runs within its container.

Google uses the *scheduling class* category to describe how latency-sensitive a job (or a task) is. The value 0 denotes low sensitivity, whereas 3 is for latency-sensitive workloads. The latter includes jobs related to user-facing applications and internal infrastructure services [12]. Therefore, we focus on class 3, which is more sensitive to SLO violations, given the goal of forecasting high-level SLO metrics for proactive adaptation. Thus, we obtain a pool of more than 400 jobs. We select one for model optimization and testing, whereas

[1] https://github.com/polaris-slo-cloud/polaris-ai

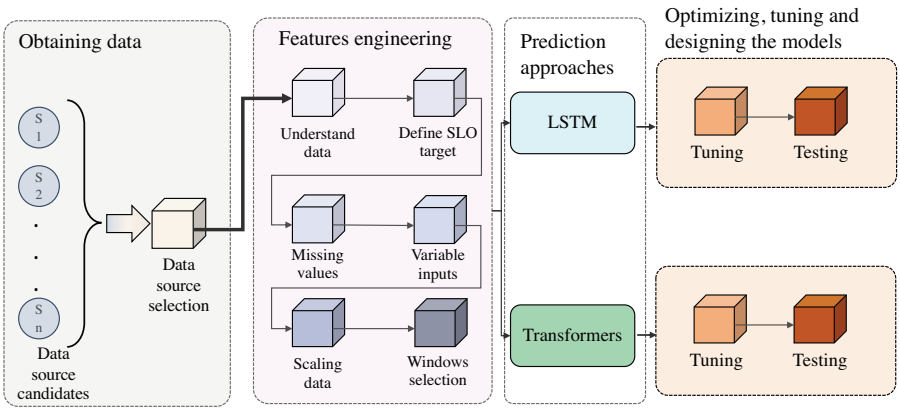


Fig. 1: Overview of the methodology and its main steps.

TABLE I: Datasets comparison.

Dataset	Size	Resources usage infos	Machine infos	Year	Documentation	Variety	Data collection
Google 2011	41GB (zip)	CPU, memory, and disk	Yes	2011	Github, community and papers	First-party workload	csv
Google 2019	2.4TB	CPU, memory, and disk	Yes	2019	Github, community and papers	First-party workload	BigQuery/ Json
Azure	235GB	CPU and memory	No	2019	Github, papers	First- and third-party workload	csv
Bitbrains	280MB (in total)	CPU, memory, network and disk	No	2013	Papers	Third-party workload	csv
Alibaba	280 GB	CPU, memory, network and disk	Yes	2018	Github, papers	First-party workload	csv

we use the rest for the out-of-distribution study. This way, we can check whether, with the bit of information provided by a single job, we can build robust and general models able to predict many use cases. As standard practice in machine learning, we split the job's data into three batches: the training set, the validation set, and the test set. [2] We use the first two for the model optimization and the test set to obtain results and evaluate the generalization error. We have 23 days of measurements for the model optimization (summing training plus validation set), which is a reasonable window to learn patterns and anomalies. The last five days constitute our test and serve for the final evaluation of our model.

Takeaways *Choosing the relevant pool of jobs is essential to provide meaningful insights. We only consider jobs from scheduling class 3, which are user-facing applications; these are more sensitive to SLO violations. Further, we select one job for training/validation/test and keep more than 400 jobs for the out-of-distribution analysis.*

2) Defining the high-level SLO metric: While the benefits of using high-level SLO s are promising, obtaining the information necessary to compute them from open-source data is often tricky. Since SLOs are tied to the application needs, they are typically left out in most of the datasets. Hence, given the available data, we have to find a compromise for our target. In our case, we perform high-level SLO metrics forecasting through *efficiency*, according to its definition provided by [16], [17]. The papers define *efficiency* as the mismatch between actual resource use and resource demand. A perfectly efficient system will match its resource usage with its demands and have a perfect *efficiency* of 1. With this notation, given a job, and a resource res measured for a task k, at time t, the *efficiency* for the resource res is represented by the Equation 1.

$$eff_t^k(res) = \frac{used_t^k(res)}{allocated_t^k(res)} \tag{1}$$

We target three resource metrics for each job to extract the *efficiency*, namely the CPU rate (CPU), the canonical memory usage (mem), and the disk usage ($disk$). We can define the total *efficiency* for a job j with m active tasks k at time t with the Equation 2; notice that we divide by three as we are considering the three resources equally. Nevertheless, in systems where CPU is more relevant we can foresee adding weights to the formula to account for this behavior.

$$Eff_t^j(CPU, mem, disk) = \frac{\sum_{k=1}^{m} eff_t^k(CPU) + eff_t^k(mem) + eff_t^k(disk)}{3m} \tag{2}$$

This high-level SLO is simply a linear combination of resource usage. Nevertheless, it is valid to show the capacity of our system to predict high-level SLO. Further, the distribution of a linear combination of random variables presents a different distribution than each one. Also, *efficiency* provides an aggregate description of the usage of the infrastructure resources that can ease their management. Imagine, for instance, that the sum is weighted with the cost of each resource; in such situations, we can obtain real insights and reduce costs.

The prediction of the *efficiency* high-level SLO metric takes for each job twelve[3] low-level metrics as features.

Takeaways *Defining a high-level SLO metric requires understanding the needs of the application to focus on the most*

[2]We split our workload data, which contains 8 352 data points (4 985 for training, one 644 for validation, and one 645 for the test).

[3]The fields are, namely: CPU rate, canonical memory usage, assigned memory usage, unmapped page cache, total page cache, maximum memory usage, disk I/O time, local disk space usage, maximum CPU rate, maximum disk IO time, cycles per instruction, memory accesses per instruction, efficiency.

important aspect. In this work, we are addressing several applications, hence, we are using efficiency which is a generic high-level SLO metric that provides a summary about any application state. Further, we consider CPU rate, canonical memory usage, and disk usage for its computation, as they are the more generic low-level metrics on the workload.

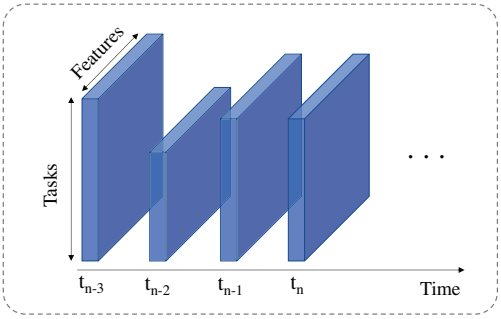


Fig. 2: The dimensions in job traces.

3) Managing variable input: Given a job, for every observation interval, Google provides an aggregated value of the measurements over the last five minutes for each task, at each step t_i. Following the representation in Figure 2, we can see the trace of a job as having a three-dimensional form. The horizontal *Time* axis represents the temporal evolution of usage information; the vertical *Tasks* axis shows the number of active tasks, and the depth *Features* axis corresponds to the number of features, i.e., the measured metrics. The variability over time of the number of active tasks, i.e., the vertical axis of Figure 2, poses challenges in data ingestion for neural network modeling. Indeed, the models we are considering must know the exact dimensions of the input data for each ingested batch in order properly to create their weights[4]. Thus, we need to neutralize the tasks' variability before developing a valid input for the neural network models. To this end, a common approach in the neural network community is to use *padding*. It offers the possibility of restraining input sequence length. A strategy involves truncating larger inputs by cutting everything exceeding the defined threshold. Otherwise, it is possible to add zeros to align short sequences to the largest value for the varying dimension.[5] However, padding severely limits the use of out-of-distribution workloads, where the task dimension can vary in an arbitrary range. In the dataset analysis, we could observe that the number of tasks in a job has wide variability, which is even more evident between different jobs.

Therefore, we rely on *aggregation* of the tasks metrics at each measurement. This is a widespread technique in statistics, and in workload characterization in particular [18]. It fixes the input shape, i.e., $(n\_features, 1)$ at each step, and keeps the richness of information by representing the features with statistics. In our case, for every step, to maintain a high-quality summary, we provide the *mean, median,* 25^{th} *quantile (*1^{st} *quartile),* 75^{th} *quantile (*3^{rd} *quartile)*, and we expand it with the *minimum, maximum, and the* 95^{th} *quantile*.

Takeaways *Padding is the typical approach for dealing with dynamic inputs. However, when considering the jobs' tasks, the padding would cap the number of tasks the input can consider at each interval. Since the tasks' feature is essential in predicting high-level SLO metrics, we lean on aggregation. This way, we extract a large set of tasks' summarization statistics, producing a correct input that fully represents the active tasks at each interval.*

4) Choosing the look-back and look-ahead window sizes: In time series forecasting, it is common to use a look-back sliding window. The look-back window represents the set of past time steps on which the models perform the estimation. Setting it too short could lead to missing relevant patterns; having it too long may mean having unrepresentative old data. In our scenario, where the job behavior varies quickly, it is essential to have fresh, relevant data. The look-ahead window choice depends on how far and how many steps in the future we want the models to predict. In our case, looking too far into the future could lead to making elasticity decisions that, given the rapid change in job behavior, can be inadequate.

We fix the size of look-back and look-ahead windows at design time. During our preliminary optimization phase for the look-back window, we consider values from 1 to 24 steps. The final choice is a window with the past 24 measurements, i.e., the last two hours of the job's activity. Regarding look-ahead window, we consider the *efficiency* values eff_{n+1}, eff_{n+2}, and eff_{n+3}, i.e., five, ten, and fifteen minutes in the future. This selection lets the system have up-to-date near-future predictions and an adequate interval to perform actions that can result in accurate SLOs violation prevention. Figure 3 depicts the approach. The blue horizontal bars represent our previous input steps, with aggregated tasks statistics. The green cubes represent the future *efficiency* values. The look-back and look-ahead windows are the same for LSTM and Transformer models.

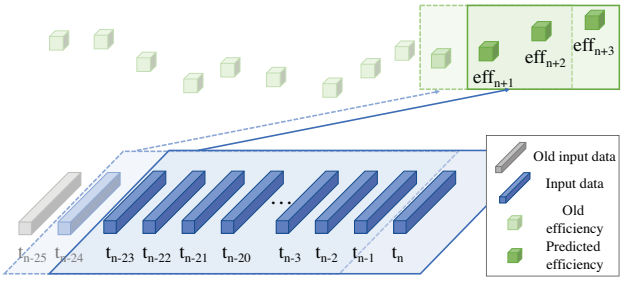


Fig. 3: Prediction steps depiction.

Takeaways *In the context of time series forecasts, the models estimate the future values on fixed-sized sliding look-back windows. In our scenario, we want to guarantee precise, up-to-date, near-future predictions to perform the correct actions promptly. Therefore, we select a 24-steps wide sliding look-back window, suitable to extract the main patterns. Given the*

[4]https://keras.io/guides/sequential_model/#specifying-the-input-shape-in-advance

[5]https://www.tensorflow.org/guide/keras/masking_and_padding

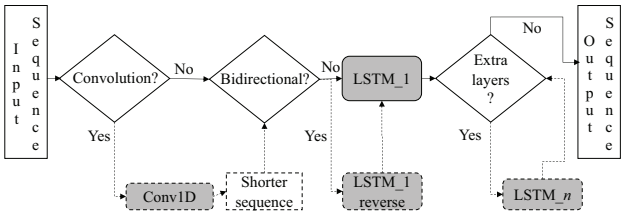


Fig. 4: Model schema for the LSTM approach.

near-future requirements, our look-ahead window targets the high-level SLO metric values for the next five, ten, and fifteen minutes.

C. Prediction models design

The rationale of this work is to predict the job *efficiency* values, estimating them based on time series of resource measurements. Therefore, it is essential to use methodologies that can deal with sequences. Since we aim at forecasting high-level SLO metrics based on low-level metrics, we need approaches that can seamlessly perform this mapping. Neural network methods natively offer automatic feature selection mechanisms [19]. In our work, we focus on two models. On the one hand, we inspect the LSTM model, a well-known recurrent neural network for time series forecasting; it guarantees fast and accurate results with lightweight implementations. On the other hand, we explore the Transformer architecture, a deep learning model based on self-attention. It achieved robust results in natural language processing (NLP) and is currently gaining momentum in other fields, like time series prediction. Differently from LSTM, it natively has many layers and neurons, making it a computationally expensive model. However, its structure offers excellent parallelization capabilities. We focus on neural network models to obtain high generalization capabilities. Statistical methods like ARIMA or ML models need more careful fine-tuning. While we did preliminary tests with them, we want to frame our work on neural models and keep them out of this evaluation.

1) LSTM-based model: This model is based on the Long Short-Term Memory layer, in which the key idea is that a cell can learn to recognize an important input, store it in the long-term state, and learn to extract it whenever it is needed. It is often appropriate to combine various layers to increase performance to forecast time series. In our work we test different setups, as depicted in Figure 4. The solid lines represent the established blocks. The dashed lines depict the optional components. The gray beveled boxes represent the logical layers, while the white rectangles show the data. In Section II-D we present the final model configuration, here we delineate the options:

Introduce a Convolutional layer before the recurrent one. This strategy is especially beneficial when dealing with long sequences and, in general, to obtain a higher-level representations of the input time series. Our idea is that this feature can provide an intermediate representation from the low-level input metrics to the high-level SLO metric.

Make the LSTM Bidirectional: The bidirectional LSTM layer looks at its input sequence both ways, obtaining richer representations and capturing patterns that the single pass might miss. In our case, we expect this technique to highlight recurring peaks at different temporal scales.

"Deepen" the network: Adding layers and "deepening" the network structure increases the number parameters allowing a more complex relation between inputs and outputs. In our study, this aspect is relevant as we have to face highly-variable data with sudden peaks and valleys. Hence, we expect to corroborate this need from the optimization process.

2) Transformer: Transformer implementations [20], have two main components: the encoder and the decoder. Each of these can be composed of several identical encoder or decoder layers. Where, each layer implements the attention mechanism, precisely a self-attention, and is followed by feedforward and normalization layers. Additionally, the decoder also implements an attention mechanism that relates the encoder's output with the decoder's current state.

As explained in [21], forecasting time series requires a complete transformer architecture with encoder and decoder layers. We have adopted the model from [22] and modified it to improve its performance and to fit our data[6]. The model presents three degrees of freedom (DoF) in terms of internal dimensions: (1) the inner dimension of the model, (2) the number of attention heads, and (3) the Query, Key, and Value (QKV) dimension. We adjust these specific dimensions of a transformer architecture as model hyper-parameters (see Section II-D). Additionally, the model has two more DoF corresponding to the number of encoder layers and the number of decoder layers. It is complex to make hypothesis regarding the best configuration of these DoFs, therefore, we entirely rely on the output obtained from the optimization process.

Further, the model has 2 inputs. The encoder inputs data as the LSTM model. However, the decoder inputs the *efficiency* values for the same time-steps. Finally, the decoder outputs the predicted *efficiency* values. The rationale is that the encoder processes low-level metrics and develops an encoded representation. Then, this is merged at the decoder with the corresponding past values of the high-level SLO metric. Finally, it provides the future values for the high-level SLO metric. This characteristic contrasts with the LSTM model, which does not explicitly require *efficiency* as input.
To leverage the transformer parallelization capabilities, the data for the training and validation process is fed into the model as three simultaneous rolling windows, which speeds up the training phase. Further, each rolling window has a different initial timestamp, providing a randomization characteristic to the training phase aiming at reducing any possible overfitting, similarly as proposed in [23], [24] or [25].

Takeaways *Despite the attempts to predict workloads behavior, the forecast of high-level SLO metric from low-level metrics has not yet found a preferred solution, making it a green field for developing tailored techniques. Our approach*

[6]https://github.com/LiamMaclean216/Pytorch-Transfomer

tests two very different solutions, i.e., LSTMs, lightweight and established in time series forecasting, and Transformers, more complex but rising in popularity, given their high accuracy. With our work, we aim to compare these two models, different in characteristics, to provide a cost/benefit analysis for both options.

D. Optimizing the models

Building a deep neural network means setting many hyper-parameters; thus, performing an exhaustive analysis and tuning is not always feasible. To build preparatory domain knowledge, we first study the prediction models' behavior with a single step ahead prediction. This step, which we do not report due to space limit, helps us fix a set of parameters representing our "ground truth."

Once we have established the ground truth, we focus on model optimization for the three-step-ahead prediction. We perform it using the hyperparameter tuning *ASHA scheduler* provided by RayTune[7]. This optimization method is appropriate for achieving parallelism during the search, making it fitting for exploring many combinations. In particular, ASHA [26] achieves good performance thanks to early stopping strategies, giving more resources and time to the most promising experiments while ceasing the ones with the least potential. For both models, we use the "`adam`" optimizer, and we compute the loss using "`mae`." In Tables II and III we report the hyperparameters that we consider in our analysis. The ones established with the single-step-ahead study are the non-bold ones. The bold parameters are the ones we let RayTune manipulate. The *Value* column shows the final decision for each hyperparameter.

TABLE II: Hyper-parameters of the LSTM model.

Hyper-parameters		Value
Model	Neurons	50
	Dropout	0.0
	Recurrent dropout	0.0
	Activation	tanh
	Activation output	sigmoid
	CNN filters	**None**
	Bidirectional	**No**
	# LSTM layers	**1**
Learning	Epochs	100
Data related	Batch size	72
	Input window length	24

Going more into detail, the results for the *LSTM* optimization process with RayTune did not improve the ground truth model. In this case, adding layers to the model does not provide a clear benefit; conversely, some combinations produce overfitting on the training and validation set. Hence, we decide to keep the LSTM model as lightweight as possible, which already provides good results without heavily overfitting. Table II summarizes the final selection of hyper-parameters. This way, we can better compare the difference between the shallow LSTM network and the rich Transformer model. The hyperparameter tuning of the *Transformer* model consists of running 1000 samples with RayTune using the configuration previously explained. Given our experience with the single step-ahead model, we fix learning and data-related parameters to shorten the hyperparameters' space to those defining the model. Table III summarizes the best performing configuration.

[7]https://docs.ray.io/en/latest/index.html

TABLE III: Hyper-parameters of the transformer model.

Hyper-parameters		Value
Model	**Encoder layers**	**8**
	Decoder layers	**1**
	Attention heads	**10**
	QKV internal dimension	**3**
	Model inner dimension	**8**
Learning	Learning rate	0.01
	Learning decay factor	0.99
	Epochs	30
Data related	Batch size	4
	Input window length	24

Takeaways *Designing and optimizing neural network models is a complex task. In this regard, approaching this prediction problem with two models allows us to take two different strategies to inspect trade-offs for each of them. Hence, after the optimization, we generate a simple and lightweight LSTM model and a complex and rich Transformer model. In our use case, the former approach should guarantee low resource consumption, while the latter could provide more accurate predictions.*

III. Comparative analysis

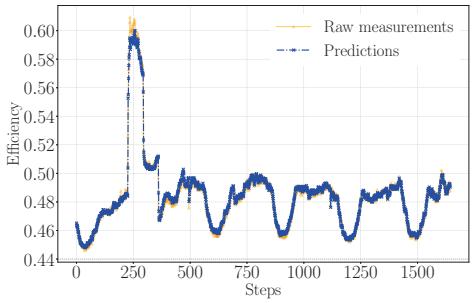
This section presents the comparative evaluation of the LSTM and Transformer models. We first juxtapose the models behavior on the test set. Then, we analyze their performance on out-of-distribution data. Finally, we provide a discussion where we examine strengths and weaknesses of the two approaches.

A. Evaluation on the test set

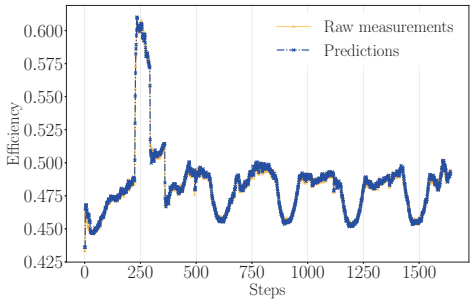
In the following, we show the results obtained from the test set for the LSTM and Transformer models. Figure 5 visually represents the target *efficiency* together with its predicted values for the entire test set, showing only the first step in the future (eff_{t+1}). From Figure 5, we can acknowledge the capability of both models to adhere to the target values.

Figure 6 provides a closer look at the results, considering the last 100 steps in the test set for LSTM (top row) and Transformer (bottom row). The LSTM and the Transformer models can accurately follow the *efficiency* evolution, showing a progression reproducing the true *efficiency* values. The Transformer approach is more sensitive to peaks, while the LSTM shows less accentuated ripples. Looking at the highlighted gap area, it can be observed an increasing shift from the original measurement as we focus on more distant predictions. So, for $t+2$, Figures 6b-6e, and $t+3$, Figures 6c-6f the gap is larger than for $t+1$, Figures 6a-6d.

Table IV shows the numeric results, precisely the Root Mean Square Error (RMSE) and in its normalized form



(a) **LSTM** - model performance .



(b) **Transformer** - model performance.

Fig. 5: Models performance over the whole test set. Example for $t+1$. The orange dashed line, with triangles, depicts the target *efficiency* values, while the blue dash-dotted line, shows the predicted values.

(RMSE %). The prediction errors are generally low, in percentage almost below 1%. Further, they confirm that results deteriorate towards far predictions. These results can depend on the model tending to rely on the latest information to forecast the following values. This intuition finds proof when analyzing the cross-correlation between actual and predicted values.

TABLE IV: LSTM and Transformer results on the test set.

		t+1	t+2	t+3
LSTM	RMSE	0.0029	0.0038	0.0049
	RMSE (%)	0.60	0.78	1.02
Transformer	RMSE	0.0029	0.0040	0.0050
	RMSE (%)	0.60	0.82	1.04

Figure 7 allows us to take a closer look at this behavior, showing the cross-correlation results for both models (LSTM (left) and Transformer (right)); the x-axis represents the offset from the curve median, while the y-axis shows the normalized correlation value; the markers in the scatter plot represent the cross-correlation at different steps. The green star portrays the behavior in the case of auto-correlation. We correlate the predicted series at $t+1$, $t+2$, and $t+3$ to the target values with the same delay. In these cases, the peaks are not centered at zero but shifted at -1, -2, and -3 for both LSTM and Transformer. This outcome confirms the intuition previously noted. In any case, the cross-correlation returns good values around the center, with a minimum value of 0.95, showing a satisfactory forecast accuracy.

B. Evaluation on out-of-distribution data

We perform inference over more than 400 selected workloads; we do not use batching, but we feed the models step by step with the look-back window to simulate a real-time prediction. This evaluation provides insights into how well the generated models can forecast different jobs. Further, we can discover the performance difference between LSTM and Transformer and highlight the strengths and limitations of each solution.

In Figure 8, we can see the average resource usage values for our training data (the red line) compared to the test set (the blue line). The values are normalized over the maximum value in the comparison, plus the standard deviation, e.g., for CPU usage: $\max\{\text{avg}_{\text{cpu\_usage}^{\text{train}}} + \text{std}_{\text{cpu\_usage}^{\text{train}}}, \text{avg}_{\text{cpu\_usage}^{\text{test}}} + \text{std}_{\text{cpu\_usage}^{\text{test}}}\}$. The blue and red areas represent the standard deviation. It is visible from Figure 8 how the test set behaves differently. Of course, the test set has a more significant deviation from the mean, as we considered 400 different jobs, but the contrast between the test and the training set we used to train the model is apparent.

Table V shows a summary of the results obtained. At first glance, we can immediately notice how the Transformer provides one order of magnitude less error in terms of RMSE and RMSE (%).

TABLE V: LSTM and Transformer results on the all workloads analyzed

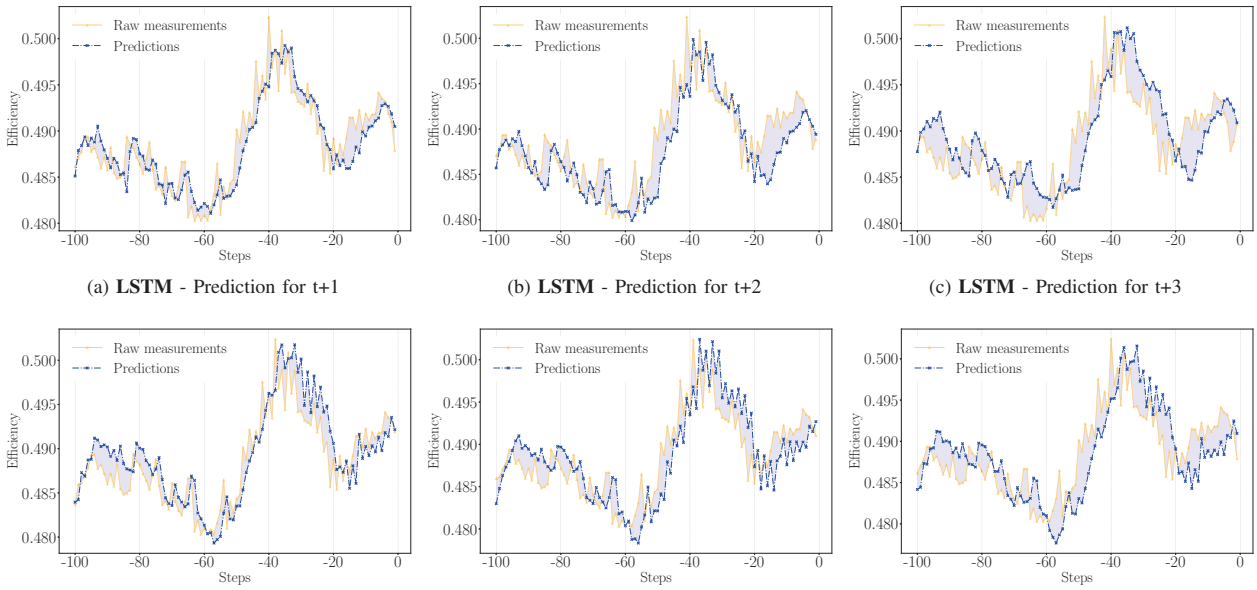
		t+1	t+2	t+3
LSTM	RMSE	0.2957	0.3156	0.2955
	RMSE (%)	58.84	63.43	57.46
Transformer	RMSE	0.0243	0.0262	0.0299
	RMSE (%)	5.527	6.126	6.687

Figure 9 shows all results as boxplots, each subplot shows the RMSE (%) for LSTM and Transformer respectively. [8] In general, the errors for Transformer are considerably lower. Nevertheless, they are one order of magnitude higher than the test set. The LSTM model is not able to provide good results, in general, when dealing with OOD data.

C. Discussion

We test the LSTM and Transformer models on the test set and out-of-distribution data. First, we could see how both models suffer from the *naive behavior*. This problem seems familiar for time-series analysis, and it is known as *naive behavior* [27] [28]. It can find its justification in the high time-series variability and the short-term forecast both models have to produce and calls for ad-hoc analyses to overcome this issue. Considering out-of-distribution evaluation, both models showed an increased divergence from the prediction and the actual value; this behavior is predictable as they performed inference on entirely new jobs. However, while the Transformer model showed one order of magnitude error increase

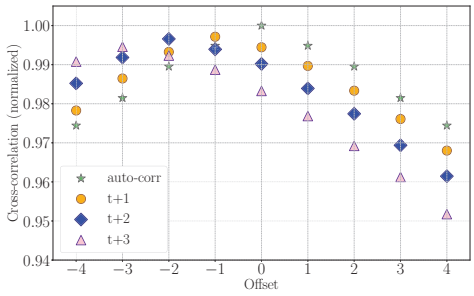
[8]The boxes represent the values between the 25^{th} quantile (1^{st} quartile), 75^{th} quantile (3^{rd} quartile). The magenta diamond represents the mean values, while the orange line is the median. The two whiskers denote the IQR, while the red fliers show the outliers.



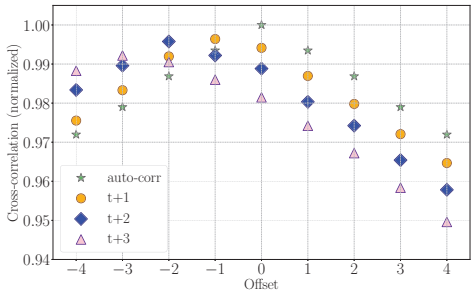
(a) **LSTM** - Prediction for t+1 (b) **LSTM** - Prediction for t+2 (c) **LSTM** - Prediction for t+3

(d) **Transformer** - Prediction for t+1 (e) **Transformer** - Prediction for t+2 (f) **Transformer** - Prediction for t+3

Fig. 6: Last 100 steps of the test series for LSTM and Transformer. The orange dashed line shows the target *efficiency*, while the blue dash-dotted line the predicted values. The light blue area highlights the gap between the target and predicted values.



(a) **LSTM** - Cross-correlation.



(b) **Transformer** - Cross-correlation.

Fig. 7: Correlation of the target series with the output series at different time steps for the last 100 values of the series.

compared to the test set results, for the LSTM, this difference is at two orders of magnitude. The Transformer's behavior falls in an acceptable error range, only around 6%, whereas for the LSTM, the average error is around 60%. While, by design, we kept the LSTM model as simple and lightweight as possible, the Transformer model results from a thorough optimization,

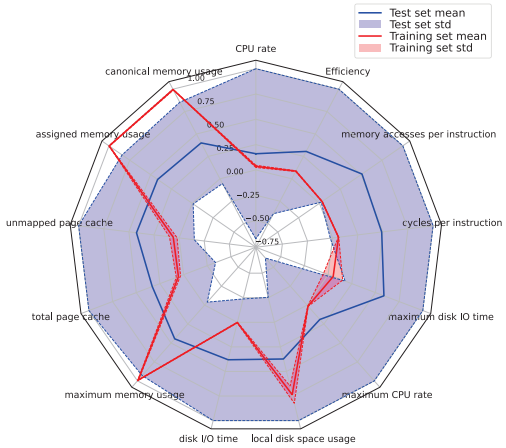


Fig. 8: Radar chart of the mean resource usage values for the training set and for the out-of-distribution test set.

plus, by structure, it uses the past values of *efficiency* at the decoder input. Therefore, these characteristics can justify the performance difference between the two models. Ultimately, analyzing the two approaches, the light LSTM model provides a faster training process and a lightweight model, making it helpful for a bootstrap monitoring phase. On the contrary, the Transformer model provides satisfactory results on out-of-distribution data but requires more extended training and tuning phases; thus, it can represent a solution in the long run.

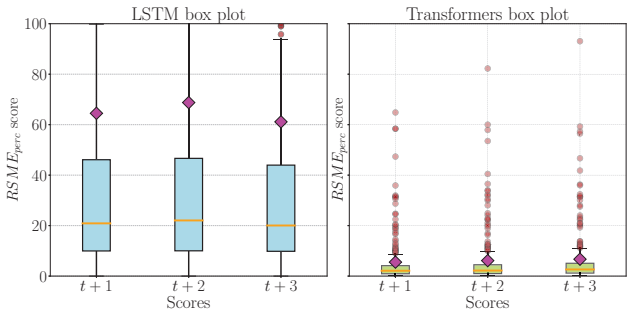


Fig. 9: $RMSE_{perc}$ results for LSTM and Transformer over *out-of-sample (OOS)* workload.

IV. Integration with the Framework

The Polaris SLO Cloud [9] is part of Centaurus project[10]. This project proposes a novel open-source platform for unified and highly scalable distributed cloud and edge systems. Polaris aims to make cloud computing SLOs first-class entities and facilitate their design and usage. Its users can leverage the CLI, based on TypeScript, to develop SLO mapping mechanisms, which allow the design and usage of high-level SLO metrics. In addition, Polaris presents predictive models for managing the SLOs. They allow proactive elasticity strategies for SLOs. Furthermore, they automatically link relevant low-level metrics to the target high-level SLO. In this way, the users do not have to care about performing complex low-to-high level metrics associations. Plus, they do not need to constantly monitor their system to develop articulated strategies for their SLOs. These processes happen under the hood, handled by a block called the *Predicted Metric Controller*. The developers only need to write the TypeScript code for the high-level SLO metric.

In the Predicted Metric Controller, one block gets the metrics from the deployed monitoring toolkit (in our case Prometheus), and the other works as a proxy to call ML services. This design allows the quick deployment of a unit that can get the relevant monitored low-level metrics and fetch, preprocess and invoke ML models to obtain predictions for the high-level SLO metric. Further, this modular design separates the ML components from the rest of the infrastructure, simplifying their interfaces. To build the ML proxy for operationalizing the models, we rely on base implementations for TensorFlow Serving [11] and PyTorch Serve [12]. The implemented services deliver the high-level SLO metrics forecasts and return them to the monitoring system, letting the users and the elasticity controllers work with predicted metrics transparently.

This approach, conversely to previous research and contributions limited in proposing data analytics sandboxing, proposes concrete solutions for operationalizing and deploying ML models in production, making them seamlessly work with the other services of the system. Most importantly, Polaris put the models and prediction capabilities at the "developers' fingertip," letting them only with the task of specifying the code for the high-level SLO metric.

Figure 10 shows a high-level representation of the main monitoring building blocks of the Polaris SLO Cloud. The predictive controller (1) obtains the low-level metrics from the monitoring tool, (2) it produces the high-level SLO metric forecasts, which (3) are sent back to the monitoring tool. In this way, the proposed layout transparently produces predictive metrics that (4) the system can use to *plan* and *execute* decisions to guarantee the application's elasticity. Further, this implementation provides seamless integration of reactive and pro-active management behaviors, which could be switched given the availability or quality of the predicted values. Moreover, this approach provides an excellent advantage for the *application developer*, who can solely focus on the SLOs definition and optimization, and not on estimating possible violations, since Polaris SLO Cloud automatically manages it.

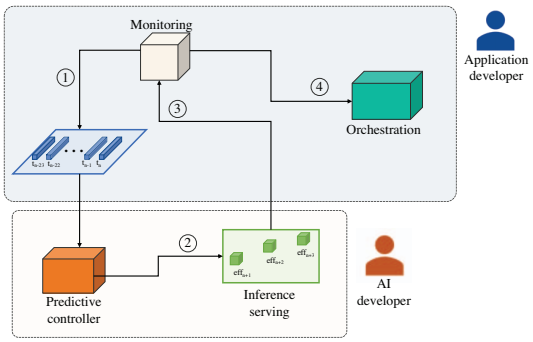


Fig. 10: Polaris architecture

V. Related Work

Traditionally, the background for solving time-series forecasting involves typical statistical methods. However, due to the complexity of cloud computing monitoring and the difficulty of promptly adapting to its changes, the last decade has witnessed several studies using machine learning techniques to forecast events and usage patterns automatically. Specifically, these studies focused on predicting metrics associated with the applications' use of the underlying infrastructure to prevent suboptimal use of the platforms.

a) Statistical methods: Traditionally, the background approach for solving time-series forecasting involves using the Autoregressive Integrated Moving Average (ARIMA) model. Thus, several approaches leveraged this method to predict cloud-related metrics over time. In [29] the authors used ARIMA to predict workload forecasting web requests from Wikimedia Foundation servers. This forecast then allows for dynamically provisioning VMs in a cloud SaaS environment. More recently, Kholidy [30] leveraged ARIMA, together with a Multiple Support Vector Regression (MSVR) model, for forecasting cloud users' resource requirements in terms of CPU, memory, and disk storage consumption. Still, in the context of statistic methods, Liu et al. [31] proposed a linear

[9]https://polaris-slo-cloud.github.io
[10]https://www.centauruscloud.io
[11]https://www.tensorflow.org/tfx/guide/serving
[12]https://pytorch.org/serve/

regression model (LiRCUP) to predict machine CPU usage over simulated traces, to obtain container consolidation.

b) Machine learning methods: Since the progress made by Recurrent Neural Networks in the last years, researchers and practitioners have started exploring the use of these techniques. In [32] they forecast CPU usage to predict the cloud host load using LSTM. Gao et al. [33] compared statistical and LSTM models for forecasting CPU and memory usage, m steps in the future, over the Google cluster dataset. They concluded that the best approach was a combination of clustering for similar workloads grouping and the Bayesian Ridge Regression model (BRR). In [34], for host load prediction, they proposed a system based on Bi-directional Long Short-Term Memory (BiLSTM). The proposed method used Google load trace data to perform the mean load prediction and actual load prediction tasks looking at the CPU. In [35] they introduced an approach that joins Autoencoders and RNN (ESN in this case) to predict CPU usage. The work of Nguyen et al. [36] suggested a system based on the LSTM Encoder-Decoder (LSTM-ED), testing it over the Google load traces for CPU usage. Similarly, Chen et al. [37] proposed a multi-resource forecast (CPU, memory, and disk) using Top-sparse Autoencoder (TSA) and Gated Recurrent Units (GRU), testing it on Google, Alibaba, DUX-based cluster traces. Autoencoders are also a key component in the work of Zhang et al. [38], where they combined them with Canonical Polyadic Decomposition (CPS) to forecast CPU usage in the PlanetLab simulation traces for workload prediction. On a different note, in [39] they used the graphical generative Deep belief networks to predict workload usage through CPU forecasting.

Although these works are interesting, they focus on low-level metrics, usually emphasizing CPU usage. Hence, these approaches aim to infer SLAs solely by the CPU usage. Despite being reliable in some cases, this perspective does not link the SLOs developer with higher-level objectives, which limits its visibility and capacity. Further, from the provider's perspective, having a unique target, like the CPU, for all SLOs means having less grip on the customers' needs, offering shallow and brittle management. From the point of view of the learning models, previous works mainly focus on optimizing one method, typically considering simplistic training data and using only a test set from the same distribution. Despite being a well-established approach, it presents limitations in a complex cloud scenario. Indeed, there is often little attention to how the models could work on unseen data at runtime, providing few elements to understand the proposed solutions' robustness. Finally, most proposed solutions focus solely on the predictive approach, skirting how to embed it in a complex system management framework. Even if this operation could be out of the scope of the presented works, it is crucial to demonstrate how the forecasting models could integrate into a cloud provider infrastructure, shedding light on which part of the system and which stakeholders could benefit from the work, and how.

VI. Conclusions

We leveraged low-level cloud workload data to forecast high-level SLO metrics in this work. This approach enables proactive Cloud management systems based on high-level SLO metrics. We have presented the methodology to obtain a high-level SLO metric from an open-source dataset and a detailed analysis of the development of two cutting-edge neural network models (Long Short-Term Memory & Transformer). We compared the performance of the models using out-of-distribution data, presenting deep insights into the strengths and limitations of the proposed models. Finally, we demonstrated how to incorporate the created model into the Polaris SLO Cloud, emphasizing the benefit of keeping the SLO metrics developer and the model predictions separate, going beyond performing some data analytics in a testing environment, and operationalizing the models.

In the future, it is interesting to look for mid- or long-term forecasts; this way, we can work on minimizing the so-called *naive behavior*. Further, we aim to compare their performance in a deployed scenario, leverage the Polaris SLO Cloud, and corroborate the need to combine simple and lightweight models with out-of-distribution resilient models to tackle all Cloud management phases. Finally, we aim to show how proactive Cloud management enhanced through predictive monitoring of high-level SLO improves the performance of the current reactive management systems.

References

[1] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *arXiv: Distributed, Parallel, and Cluster Computing*, 2016.

[2] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, "Machine learning-based orchestration of containers: A taxonomy and future directions," *ACM Computing Surveys (CSUR)*, 2021.

[3] D. Saxena and A. K. Singh, "Workload forecasting and resource management models based on machine learning for cloud computing environments," *ArXiv*, vol. abs/2106.15112, 2021.

[4] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong, "Principles of elastic processes," *IEEE Internet Computing*, vol. 15, no. 5, pp. 66–71, 2011.

[5] S. Nastic, A. Morichetta, T. Pusztai, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "Sloc: Service level objectives for next generation cloud computing," *IEEE Internet Computing*, vol. 24, no. 3, pp. 39–50, 2020.

[6] J.-C. Gagnon-Audet, K. Ahuja, M. J. D. Bayazi, G. Dumas, and I. Rish, "Woods: Benchmarks for out-of-distribution generalization in time series tasks," *ArXiv*, vol. abs/2203.09978, 2022.

[7] J. Wang, C. Lan, C. Liu, Y. Ouyang, W. Zeng, and T. Qin, "Generalizing to unseen domains: A survey on domain generalization," *arXiv preprint arXiv:2103.03097*, 2021.

[8] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in neural information processing systems*, vol. 28, 2015.

[9] T. Pusztai, S. Nastic, A. Morichetta, V. Casamayor Pujol, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "A novel middleware for efficiently implementing complex cloud-native slos," in *IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

[10] ——, "Slo script: A novel language for implementing complex cloud-native elasticity-driven slos," in *IEEE International Conference on Web Services (ICWS)*, 2021.

[11] S. Nastic, T. Pusztai, A. Morichetta, V. Casamayor Pujol, S. Dustdar, D. Vii, and Y. Xiong, "Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters," in *IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 206–216.

[12] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–17.

[13] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 153–167.

[14] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*. IEEE, 2019, pp. 1–10.

[15] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, pp. 1–14, 2011.

[16] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *10th international conference on autonomic computing ({ICAC} 13)*, 2013, pp. 23–27.

[17] M. Becker, S. Lehrig, and S. Becker, "Systematically deriving quality metrics for cloud computing systems," 2015.

[18] S. F. Piraghaj, R. N. Calheiros, J. Chan, A. V. Dastjerdi, and R. Buyya, "Virtual machine customization and task mapping architecture for efficient allocation of cloud data center resources," *The Computer Journal*, vol. 59, no. 2, pp. 208–224, 2016.

[19] F. Chollet, *Deep learning with Python*. Simon and Schuster, 2021.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[21] G. Zerveas, S. Jayaraman, D. Patel, A. Bhamidipaty, and C. Eickhoff, "A Transformer-based Framework for Multivariate Time Series Representation Learning," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, aug 2021, pp. 2114–2124.

[22] N. Wu, B. Green, X. Ben, and S. O'Banion, "Deep transformer models for time series forecasting: The influenza prevalence case," *arXiv preprint arXiv:2001.08317*, 2020.

[23] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," *IEEE International Conference on Intelligent Robots and Systems*, vol. 2017-September, pp. 23–30, dec 2017.

[24] R. Yang, M. Zhang, N. Hansen, H. Xu, and X. Wang, "Learning vision-guided quadrupedal locomotion end-to-end with cross-modal transformers," *arXiv preprint arXiv:2107.03996*, 2021.

[25] N. Hansen, H. Su, and X. Wang, "Stabilizing Deep Q-Learning with ConvNets and Vision Transformers under Data Augmentation," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. S. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 3680–3693. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/1e0f65eb20acbfb27ee05ddc000b50ec-Paper.pdf

[26] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-Tzur, M. Hardt, B. Recht, and A. Talwalkar, "A system for massively parallel hyperparameter tuning," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 230–246, 2020.

[27] M. Du, "Improving lstm neural networks for better short-term wind power predictions," *2019 IEEE 2nd International Conference on Renewable Energy and Power Engineering (REPE)*, pp. 105–109, 2019.

[28] M. E. Aydin and S. S. Kozat, "A hybrid framework for sequential data prediction with end-to-end optimization," *ArXiv*, vol. abs/2203.13787, 2022.

[29] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications' qos," *IEEE Transactions on Cloud Computing*, vol. 3, pp. 449–458, 2015.

[30] H. A. Kholidy, "An intelligent swarm based prediction approach for predicting cloud computing user resource needs," *Computer Communications*, vol. 151, pp. 133–144, 2020.

[31] J. Liu, S. Wang, A. Zhou, J. Xu, and F. Yang, "Sla-driven container consolidation with usage prediction for green cloud computing," *Frontiers of Computer Science*, vol. 14, no. 1, pp. 42–52, 2020.

[32] B. Song, Y. Yu, Y. Zhou, Z. Wang, and S. Du, "Host load prediction with long short-term memory in cloud computing," *The Journal of Supercomputing*, vol. 74, pp. 6554–6568, 2017.

[33] J. Gao, H. Wang, and H. Shen, "Machine learning based workload prediction in cloud computing," *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–9, 2020.

[34] H. Shen and X. Hong, "Host load prediction with bi-directional long short-term memory in cloud computing," *arXiv: Signal Processing*, 2020.

[35] Q. Yang, Y. Zhou, Y. Yu, J. Yuan, X. Xing, and S. Du, "Multi-step-ahead host load prediction using autoencoder and echo state networks in cloud computing," *The Journal of Supercomputing*, vol. 71, pp. 3037–3053, 2015.

[36] H. M. Nguyen, G. Kalra, and D. Kim, "Host load prediction in cloud computing using long short-term memory encoder–decoder," *The Journal of Supercomputing*, vol. 75, pp. 7592 – 7605, 2019.

[37] Z. Chen, J. Hu, G. Min, A. Y. Zomaya, and T. A. El-Ghazawi, "Towards accurate prediction for high-dimensional and highly-variable cloud workloads with deep learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, pp. 923–934, 2020.

[38] Q. Zhang, L. T. Yang, Z. Yan, Z. Chen, and P. Li, "An efficient deep learning model to predict cloud workload for industry informatics," *IEEE Transactions on Industrial Informatics*, vol. 14, pp. 3170–3178, 2018.

[39] F. Qiu, B. Zhang, and J. Guo, "A deep learning approach for vm workload prediction in the cloud," *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 319–324, 2016.