End-to-End Latency Prediction of Microservices Workflow on Kubernetes: A Comparative Evaluation of Machine Learning Models and Resource Metrics

Haytham Mohamed
Dakota State University
hmmohamed@pluto.dsu.edu

Omar El-Gayar
Dakota State University
omar.el-gayar@dsu.edu

Abstract

Application design has been revolutionized with the adoption of microservices architecture. The ability to estimate end-to-end response latency would help software practitioners to design and operate microservices applications reliably and with efficient resource capacity. The objective of this research is to examine and compare data-driven approaches and a variety of resource metrics to predict end-to-end response latency of a containerized microservices workflow running in a cloud Kubernetes platform. We implemented and evaluated the prediction using a deep neural network and various machine learning techniques while investigating the selection of resource utilization metrics. Observed characteristics and performance metrics from both microservices and platform levels were used as prediction indicators. To compare performance models, we experimented with a benchmarking open-source Sock Shop containerized application. A deep neural network technique exhibited the best prediction accuracy using all metrics, while other machine learning techniques demonstrated acceptable performance using a subset of the metrics.

1. Introduction

Recent progress made in container technology paves new ways to adopt containerized microservices in cloud infrastructure [1]. Containers orchestrating and scheduling platforms such as open-source Kubernetes [2] enables to automatically deploy, host and scale distributed containerized microservices [3]. A microservice could be built and bundled with all dependencies as one deployable image using a containerization technology such as Docker [4] and scheduled to run on a platform such as Kubernetes. The number of instances of each microservice in addition to the amount of required resources - such as memory and central processing unit (CPU) - of each instance could be specified to run applications reliably. Kubernetes cloud platform takes the responsibility to accommodate hosted microservice instances with adequate capacity. A microservice caters functionality with one or several deployed containerized instances, known as Kubernetes Pods. Typically, an application is composed of several microservices that are deployed and run in a cluster of infrastructure nodes (Kubernetes cluster).

Despite all the benefits of using Kubernetes, the distributed nature of cloud platforms comes with operation challenges to reliably manage running and to scale microservices [5]. The performance of microservices run as containers hosted on a Kubernetes cluster can degrade often in unpredictable ways. Microservices performance degradation could manifest with increased values in response time. Therefore, end-to-end response latency is considered to be one of the microservices reliability indicators. End-to-end response latency is commonly determined in terms of a percentile (i.e. 90^{th} Percentile or 95^{th} Percentile) within a certain time-window. Targeting a certain end-to-end response latency threshold assists in evaluating microservices reliability, dynamic microservices scaling, performance anomaly detection and alerting [6]. For instance, when a service exhibits an unacceptable latency threshold, software practitioners might need to rollback a newly deployed culprit release, adjust scaling a service with an adequate number of replicas, or fix a bug that causes the latency.

End-to-end response latency is sensitive to changes in either the application, operating system or hardware [7]. Hence, a variety of observed characteristics in microservices and the underlying cloud platform could be used as indicating factors to predict end-to-end response latency of requests flowing through microservices. For example, we can leverage performance metrics and build performance models to predict end-to-end response time of a purchasing order workflow in an online shopping application deployed

URI: https://hdl.handle.net/10125/70820
978-0-9981331-4-0
(CC BY-NC-ND 4.0)

as containerized microservices in a cloud platform such as Kubernetes. A flow of a purchasing order request may involve many microservices in its execution path, each performs a certain task. For instance, a typical request might need to invoke requests to a shopping cart, payment, shipping and probably user information microservices. Observed resources utilization and other performance metrics of microservices in workflow and underlying cluster nodes of the platform could be used to predict how long it took since a user requested until a response returned.

Many attempts were made recently to develop performance prediction modeling either for independent microservices using an analytical method [8] or for an interoperable ensemble of microservices that fall in a request execution path and their running platform using a data-driven method [6]. Analytical methods usually need experts' knowledge to change and instrument applications code for workload profiling. Such an approach is not practical in most cases with cloud applications composed of many microservices. Research efforts that used a data-driven modeling approach related observed resource metrics with the application performance [9]. A very recent data-driven effort attempted by [6] related CPU utilization of both the microservices and hosting virtual machine with application end-to-end tail latency. Latency in responses could be caused by other factors besides just the time spent by CPU utilization during computation time. Thus, we indented to expend on [6] attempt to consider additional factors in microservices and underneath platform, such as network utilization, disk I/O utilization, and the number of running instances of each microservice.

This research aims to investigate using machine learning techniques and compare their accuracies in predicting end-to-end response latency of requests that flow through a microservices application. The prediction depends on observed runtime performance characteristics of all microservices in the request execution path in addition to those of the underlying Kubernetes cluster nodes. We examined different machine learning techniques such as Linear Regression (LR), Support Vector Regression (SVR), Decision Trees (DT), Random Forest (RF), K-Nearest Neighbors (KNN), Extreme Gradient Boosting (XGBoost) in addition to Deep Neural Network (DNN). In addition to using and comparing between different machine learning models, we demonstrate that a variety of resource utilization metrics and performance characteristics of both microservices and underlying Kubernetes cluster platform could impact predicting end-to-end response latency. Our findings could guide practitioners to observe appropriate metrics per specific workflow in a Kubernetes platform. Furthermore, obtaining accurate predicting performance model would suggest scheduling microservices in Kubernetes with properly tuned resources.

The remainder of the paper is organized as follows. Section two outlines a recent advancement in microservices architecture running in cloud platforms and related works in existent research to predict end-to-end latency. Section three describes the experimental setup, the considered machine learning techniques, the collected metrics data and their use to train and evaluate the models. Experiment results were summarized in section four. Section five discusses the effort and findings. Section six concludes the paper.

2. Background and related work

2.1. Microservices on cloud platforms

Recent advancement in software development introduces the microservices architecture paradigm that is well suited to the elasticity feature of cloud platforms. Microservice architecture takes designing applications to another level with new principles of well-defined responsibility, scalability, rapid evolvement and easy adoption [10]. Instead of one monolithic architecture that encompasses all application components as one unit, the new architecture breaks application components into several microservices. This new architecture creates autonomous services in terms of development and operational management lifecycle where each service evolves independently [11]. A microservice should have a well-defined business domain context with a single responsibility. For example, an E-Commerce online application could be composed of different independent microservices, such as shipping, pricing, catalog information, checkout, and shopping cart. Each of these microservices evolves independently and is responsible to execute a specific function.

Cloud platforms have been widely adopted by several industries to deploy and run mission-critical workloads. Microservices architecture is well suited to leverage the basic functional offerings of cloud computing platforms. Container-based microservices could be deployed in a cloud-managed cluster of virtual machines such as open-source Kubernetes. The advantages of deploying microservices applications in a managed cloud Kubernetes platform include scheduling, scaling, and auto-recovery [12].

However, running microservices efficiently in a Kubernetes cloud platform requires a clear and real-time understanding of performance characteristics and

their effect on microservices reliability [5]. End-to-end response latency in a microservices application is considered as one of the application's quality indicators [13]. End-to-end response latency of a request flows through instances of different microservices could be affected by the performance characteristics of those instances, in addition to the characteristics of the underlying platform.

2.2. End-to-End latency prediction

Researchers such as [14] asserted to the fact that monitoring of inter-dependent microservices in cloud platforms is challenging. The difficulty was attributed to the distributed nature of the microservice application deployment. A typical request execution path may span multiple and concurrent microservices that complicate interaction and tracing across services. Furthermore, microservice instances run on the same cluster node are subjected to resource utilization interference by other processes run on the same node. Therefore, the performance of containerized microservices that run in a cloud platform such as Kubernetes my unexpectedly degrade [6]. These challenges suggest that a holistic monitoring approach at different cloud application layers (i.e., platform, services, containers) would be helpful to detect microservices degradation and anomalies.

The study by [6] applied machine learning techniques to predict the 95th percentile of end-to-end requests latency in a microservices application workflow. The authors evaluated the prediction of proposed performance models such as Linear Regression, Support Vector Regression, Decision Tree, Random Forest and Deep Neural Network. The neural network model performed best to predict end-to-end latency using the CPU utilization rate at the microservices and infrastructure virtual machines levels. As the authors accounted solely for CPU utilization rates to make the prediction, we extend their efforts to examine the prediction but leveraging utilization metrics of other resources besides the CPU utilization rate.

Other prior efforts include the work by [8] who developed a performance modeling, prediction method for independent microservices and formulated a microservice-based application workflow scheduling problem for minimum end-to-end delay. The authors of this study focused on microservices scheduling to minimize the end-to-end response delay under a pre-specified budget constraint. Another effort by [15] used response latency as a driving factor to trigger autoscaling cloud microservices. Moreover, [16] focused to investigate the impact of the heap size, garbage collection, concurrency and service demand on the tail latency of Java microservices.

Our work extends prior attempts to build performance prediction models of microservices by evaluating mostly the typical performance models examined by [6] while considering performance metrics such as services CPU utilization, services network utilization rates, number of replicas of each service, besides indicators such as CPU utilization, network utilization and Disk I/O (input/output) utilization rates on the managed Kubernetes nodes level. Further, we aimed to build performance models taking into account a combination of various resource metrics from a Kubernetes platform and microservices levels.

3. Methodology

3.1. Experimental setup

We collected various telemetry data from all microservices instances involved in making a purchasing order request. CPU and input/output (I/O) utilization of underlying cluster nodes were also collected. To be able to perform a comparison with the inspiring effort done by [6] we also collected the 95th percentile response latency metric. Data collected were fed into various machine learning models from which the prediction accuracy results were noted.

To set up the experiment, we leveraged the open-source "Sock Shop" [17] microservices application developed by Google. "Sock Shop" microservices application was commonly used in prior efforts as a performance characterization benchmarking and simulating application. It implements an online e-commerce experience where users can browse items, add items to a cart and make purchases. It is composed of many e-commerce microservices such as *front-end, user, catalog, payment, cart, and order*. Microservices in this application were implemented using different programming languages such as Node.js, Java and Go-Lang.

All microservices were deployed on a cloud-managed Kubernetes cluster maintained by "Linode" [18] cloud platform provider. The Kubernetes cluster consists of up to five nodes, each has 4 vCPU and 8 GB of RAM. It is worth mentioning that a microservice is not an individual object to track but complicated as its replicas are deployed across many nodes in the cluster. Therefore, to provide a uniform way to observe and collect metrics at the microservices level we enabled our Kubernetes cluster with the open-source "Istio" [19] service mesh. The application was subjected to various workloads of

concurrent requests using the benchmarking open source load test tool "Locust" [20]. The workloads ramped up to 1,500 concurrent users with a stepping rate of 10 users every 5 minutes. The metrics were pulled from the application using open source "Prometheus" [21] rules in 15-second intervals. Collected metrics were stored in a time-series enabled "Postgres" [22] database.

Individual performance time-series metrics of each microservice in the order workflow were extracted from the "Postgres" database and written out to comma-separated files. The same was done for metrics collected from the nodes in the cluster. The data was correlated using a date-time index, resampled with 15-second time-window and rearrange into a one dataset matrix of size 6,766 rows and 29 columns (including the latency metric). The collected data was then used to train and evaluate the end-to-end latency prediction accuracy of different machine learning models. Figure 1 below depicts the applied experiment method. We would like to note that, the experiment was not concerned with fixing any application or platform limitation issues to run as an error-free simulation. Rather, the experiment focused only on being able to subject the application to requests and gather various run-time metrics.

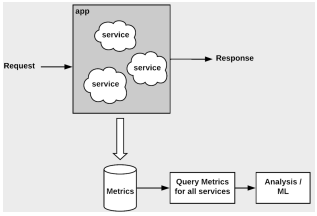

Figure 1. Metric collection simulation

3.2. Machine learning models

We considered different machine learning algorithms to perform the prediction. Overall, we built performance regression algorithms such as Multiple Linear Regression (LR), second-degree Polynomial Linear Regression (PLR), Support Vector Regression (SVR), k-Nearest Neighbors regressor (KNN) and Decision Tree (DT). LR is a technique that uses several explanatory variables to predict the outcome of a continuous response variable. The goal of LR is to model the linear relationship between the explanatory (independent) variables and a response (dependent) variable. The main aim of SVR is to decide a decision boundary at a distance from the original hyperplane such that data points close to the hyperplane or the support vectors are within that boundary line. The KNN algorithm uses "feature similarity" to "k" number of nearest data points to predict a value. This means that the new point is assigned a value based on how closely it resembles those number of "k" points in the training set. DT branches out using information gain as the splitting criterion leading to decision hierarchy and ultimately to a final predicted value.

Additionally, we evaluated the prediction from some of the ensemble algorithms such as Random Forest (RF) and Extreme Gradient Boosting (XGBoost). Ensemble algorithms are a powerful class of machine learning that combine the predictions from multiple models. Last but not least, we also implemented a Deep Neural Network (DNN) by composing a one-dimensional convolutional network and a fully multi-layer neural network. We supplied all metrics to the DNN model and only a selected subset to all other machine learning techniques.

3.3. Performance metrics

Users commonly interact with online e-commerce web applications using the HTTP protocol. They usually use an HTTP URI to perform actions through a graphical web interface. Typically, a request execution path in a microservices application flows across different services, each performs a certain task. All visited instances that part of a request execution flow forms a directed acyclic graph (DAG). As was noted by [6], this would cause end-to-end latency to be impacted by the performance characteristics of all visited microservice instances in the request execution path.

We focused in this paper on the purchasing order workflow. Hence, we observed and collected performance metrics that pertained only to all microservices that are part of the order requests graph. The list of these microservices was leveraged from the work done by [6] and verified by using the open-source virtualization tool "weavescope" [23]. Microservices that are part of the purchasing order requests execution path are illustrated in Figure 2 below and include *front-end, orders, user, shipping, payment, cart, users-db, orders-db and cart-db*.

There are two levels of key metrics we considered in our case: metrics from the managed Kubernetes cluster nodes level and metrics from the microservices workload level. From the microservices level, the following metrics were collected for every microservice in the orders requests execution flow:

- Microservices CPU utilization rate in seconds
- Microservices network utilization rate in bytes
- Microservices number of instances

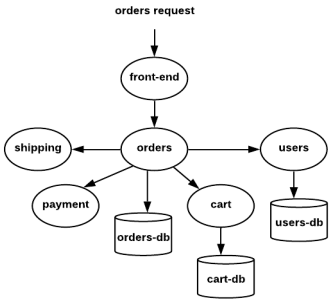

Figure 2. Microservices in the orders request execution path

On the cluster nodes level, and to further capture the inference impact of these microservices on each other, the below metrics from the managed Kubernetes nodes were also collected:

- Nodes CPU utilization rate in seconds
- Nodes network utilization rate in bytes
- Nodes disk I/O utilization rate in bytes

Additionally, we also captured the requests rate and observed the 95th percentile of the end-to-end response latency (in seconds) from requests flew through the *orders* microservice.

3.4. Training, evaluation and features selection

A behavior of one microservice is affected by the performance characteristics of all its instances in a cluster. These performance characteristics defined as Service Level Indicator (SLI) metrics were used as independent input features to predict the end-to-end response latency.

All machine learning algorithms were implemented in Jupyter notebooks [24] using Python scikit-learn [25] machine learning tools. All individual collected metrics were concatenated together to form a total of 6,766 records and saved in one file to read by the notebook. A record consists of 29 columns, each with 28 observed metric data elements, in addition to the last column that contains the 95th percentile of the response latency. To avoid over-fitting of the performance models we reduced the number of features using the stability features selection technique. As performed by [6], we leveraged Python scikit-learn Randomized Lasso to obtain a fewer number of relevant features to the latency values. Randomized Lasso penalizes the absolute value of the coefficients with a positive penalty term that less than 1. This technique reduced the features to only consider the request rate besides the CPU utilization of the *front-end, orders database, user database, shipping* and carts services. Additionally, the technique also considered the network utilization of *orders, shipping, payment, carts and carts database* microservices. From the cluster nodes metrics, the technique picked the CPU and disk I/O utilization.

The selected features and target orders latency fields were then used to train and validate the machine learning models. For the machine learning technique, we hold-off 20% of the data for testing, while used the rest to train and validate the models. The grid search cross-validation technique of 10 folds was used with a pipeline of tasks to scale and train the different regression models. The grid cross-validation performed picking the optimum hyper-parameters out of many configuration options that would achieve the highest coefficient of determination R^2 score. R^2 score is an indicator of how perfectly a regression prediction fits the data. Table 1 below lists the machine learning techniques and their various hyper-parameter options to search the optimal configuration for the highest performing models. Theoretically, if a model could explain 100% of the variance in the observed data, the predicted values would always equal the measured values and, therefore, all the data points would fall on the fitted regression line [6]. The more the R^2 score value is close to 1 the more the model fits well with the data.

Furthermore, a deep neural network model was also implemented using Python Keras tools [26]. The deep neural network model was formed using a 1-dimensional Convolutional Neural Network (CNN) followed by fully connected hidden layers that ended with a sigmoid output layer. The CNN part of the model was used to extract features out of all 28 input metric indicators before feeding a fully connected multi-layer neural network to perform the prediction. The CNN model consists of two layers each with 32 filters and a kernel size of 2, followed by a 0.1 dropout rate layer, a 1-dimension maximum pooling layer with a pool size of 2 and a flattening layer. The neural network consists of four layers of size 200, 100, 50 and 10 nodes in sequence before ending in a last linearly activated output layer. The deep neural network model used the mean squared error as a loss function to minimize while learning the complex pattern between the input metrics features and the latency values. The data was sliced to 10% for testing, 10% for validation and the rest to train the model.

Table 1 Machine learning models hyper-parameter options

Model	Hyper-parameters options
LR	Normalize parameter: false, true
PLR	Normalize parameter: false, true
SVR	Regularization parameter: 1, 0.5, 0.8 Kernel: linear, RBF
KNN	Number of neighbors: 3, 5,7
DT	Minimum samples in leaf: 1, 2, 3
RF	Maximum features: 14, 13, 11, 9 Number of estimators: 15, 10, 8, 6
XGBoost	Learning rate: 01, 0.05 Alpha: 1, 5 Subsample ration of columns: 0.5, 0.8 Maximum depth: 25, 50 Number of estimators: 100,150

4. Results

We used Parallel Coordinates Plots with a portion of the applied workload to examine the effect of different metrics indicators on the response latency. The graphs displayed in Figure 3 and Figure 4 contrast between various levels of response latency based on the CPU and network utilization of all microservices involved in the order request execution path (*front-end, orders, users, carts, shipping and payment*). Both *shipping* and *payment* microservices did not have noticeable CPU and network utilization variances. It is also shown that latency increased despite low CPU utilization rate values, indicating other time-consuming processes such as input/output (I/O) operations to cause delays in responses. Furthermore, Figure 5 illustrates the effect of the cluster nodes' CPU, disk I/O and network utilization rate on the response latency. It is noticeable that the nodes CPU and network utilization rate had more impact on increasing responses latency, while the disk I/O utilization rate did not show a big influence.

To take a look into the workload effect on latency, Figure 6 illustrates the relationship between the applied request rate and the corresponding observed response latency. It is shown that the latency spiked when the request rate increased, and latency decreased when the application was subjected to a lower rate of requests.

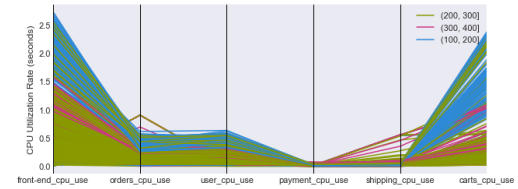

Figure 3. Parallel coordinates plot illustrates the effect of CPU utilization rate on latency

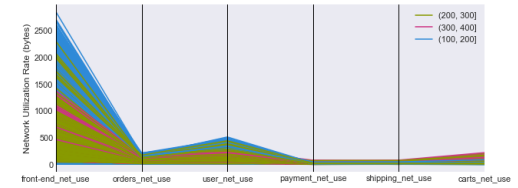

Figure 4. Parallel coordinates plot illustrates the effect of network utilization rate on latency

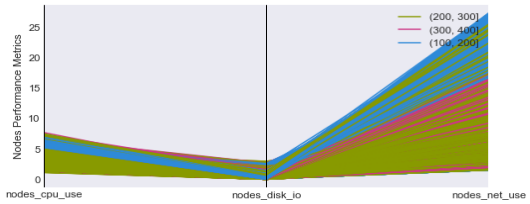

Figure 5. Parallel coordinates plot illustrates the effect of nodes CPU, disk I/O and network utilization rate on latency

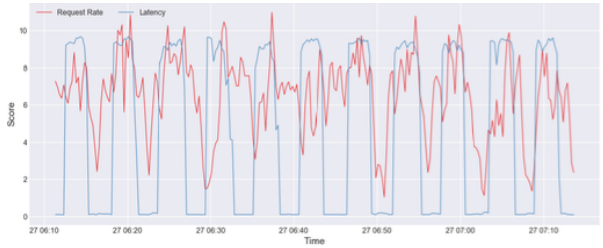

Figure 6. Request rate vs latency

Table 2 summarizes the achieved prediction accuracy of each model in terms of R^2 and the same is plotted in Figure 7. The prediction accuracy obtained was varied based on the regression algorithm used. Linear regression (LR) only resulted in R^2 score of 44.4%. However, by using a second-degree Polynomial Linear Regression (PLR) we were able to increase the accuracy score to 69.2%. K-Nearest Neighbors (KNN), Decision Tree (DT) and Support Vector Regression (SVR) models achieved accuracy scores of 72.3%, 73.6% and 76.2% respectively. The prediction accuracy was further enhanced using ensemble techniques such as Random Forest (RF) and XGBoost models. RF excelled with R^2 score of 79.3% while XGBoost enhanced the accuracy slightly and

achieved R^2 score of 79.6%. As anticipated for a deep neural network to learn the complex relationship between the metrics indicators and response latency, our hybrid model of convolutional neural network and multi-layer neural network architecture was able to learn the pattern from all of the 28 metric features and resulted in a high R^2 prediction accuracy score of 80.0%.

Table 2. Models predication accuracy (R^2)

Model	R^2	hyper parameters / configuration
DNN	0.8002	CNN: 2 1-D conv Layers (32 filters / 2 kernel size) + dropout + maxPooling + flatten layers NN: (200, 100, 50, 10, 1) nodes layers Epochs: 1500, lr: 0.01, batch: 300
XGBoost	0.7957	alpha: 1, colsample_bytree: 0.8, learning_rate: 0.05, max_depth: 50, n_estimators:150
RF	0.793	max_features: 14, n_estimators; 15
SVR	0.7618	C: 1, kernel: rbf
DT	0.736	min_sample_leaf: 3
KNN	0.7231	n_neighbors: 3
PLR	0.6916	
LR	0.4439	

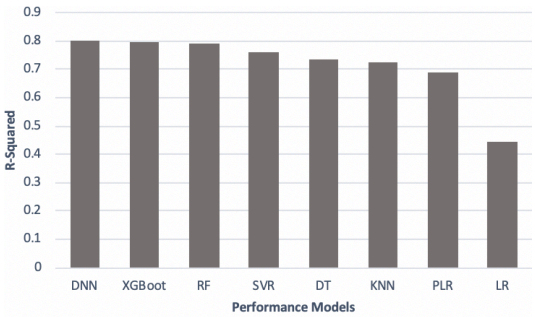

Figure 7. Performance models prediction accuracy

5. Discussion

In contrast with the work done by [6], we can note that LR did not do good in its R^2 prediction accuracy scores. This result asserts to the same previous research finding, linear regression models were not able to capture the non-linearity of response latency. However, running the regression with second-degree polynomial transformed features and enhanced the linear prediction accuracy. Resembling with the nearest three neighbors, KNN was able to predict the latency with slightly better accuracy. DT achieved a prediction accuracy that almost close to that of KNN model by limiting its leaf nodes to a minimum of three samples. The prediction accuracy score increased with SVR model based on RBF kernel. Additionally, the ensemble RF algorithm with 15 estimators was able to score a better prediction accuracy result and even higher using a boosted ensemble algorithm such as XGBoost with 150 estimators. While all these models learned using a subset of selected features, a DNN model that was composed of a CNN followed by a fully connected NN was able to learn from all the 28 metric features and achieved relatively the highest prediction accuracy score. The first CNN sub-model purpose was to capture the more relevant features out of all 28 metrics input and then used that to perform the prediction with the subsequent NN model. The DNN model was able to capture the non-linearity relationship between the observed metrics and response latency.

Furthermore, we consulted the Shapley Additive exPlanations (SHAP) [27] values of the performance models to find out the observed metrics with high impact to predict the response latency per our experiment. SHAP is based on the game theoretically optimal "Shapley Values". It is a method to explain individual instance prediction from a sample or to globally explain a model's prediction in general. SHAP values perform the explanation by computing the contribution of each feature to the prediction. We examined the global interpretation of the RF and DNN models using their SHAP values. The horizontal lines in a SHAP summary plot illustrate all sample instances values in a training dataset for every feature. Values with red color illustrate higher values than the ones with blue colors. Values on the right indicate a positive impact on prediction while values on the left side indicate a negative impact.

As suggested by the right bar on the SHAP summary plot of the RF model in Figure 8 the high values of CPU and network utilization rates in the *carts* service, network utilization rate in the *carts* database service, CPU utilization rate in the *user* database service, network utilization rate on the *payment* service, high values of CPU utilization rate on the cluster nodes and the number of instances of the *front-end* service are all positively impact the SHAP values, hence contributing to increase a latency predicted value.

On the contrary, this figure shows that a higher disk I/O utilization rate of the cluster nodes and network utilization rate of the *orders* service pull a

predicted latency value down. The network utilization rate measures both the incoming and outgoing traffic between a service and a client. Therefore, we could justify that as the network traffic from the *orders* service increases, responses would be faster, thus decrease the latency. And we could justify the negative impact caused by high values in disk I/O utilization rates of the cluster nodes with a reason that as data are read/written faster to disk the response would be formed faster, thus caused the latency to decrease. Last but not least, the plot summary diagram suggests a low *shipping* service impact to the predicted latency values, as the *orders* service performed making shipment requests asynchronously. This observation agrees to what illustrated by the Parallel coordinates plots in Figure 3 and Figure 4

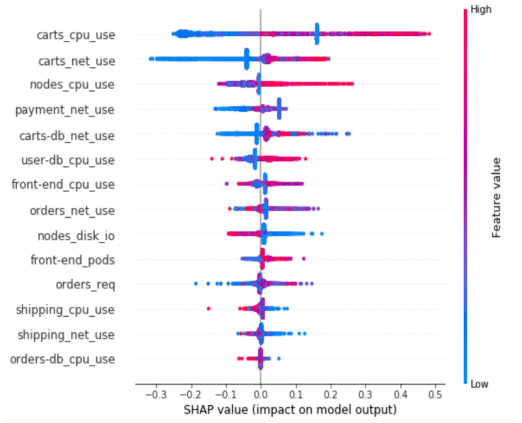

Figure 8. RF features important based on SHAP values

Also examining the SHAP summary plot of the DNN model in Figure 9, we could see a list of the most 20 impactful metrics features. We could also notice that the CPU, network of the *carts* service and the network utilization rates of the *carts database* service were still among the highest influencing predictors in the model. The CPU utilization rate of the cluster nodes was also still having a high impact. It seems that the neural network model was sensitive to the number of replicas in the *front-end* and *user* services as the number of pods of each was high in their feature values.

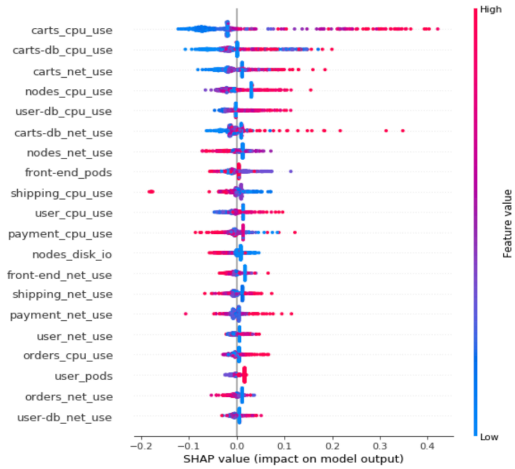

Figure 9. Neural network features importance based on SHAP values

To illustrate the impact of the features in a further simple presentation, the summary plot in Figure 10 depicts in numeric measures those features matter to the DNN model. Features were displayed in decreasing importance and the length of the bars in the figure helped us to virtually make comparisons between all the metric features. Hence, while the effort made by [6] used CPU resource utilization, we were able to illustrate those resource characteristics such as the services number of replicas and network utilization rate of the services and platform nodes were contributing besides CPU utilization rate to predict the response latency.

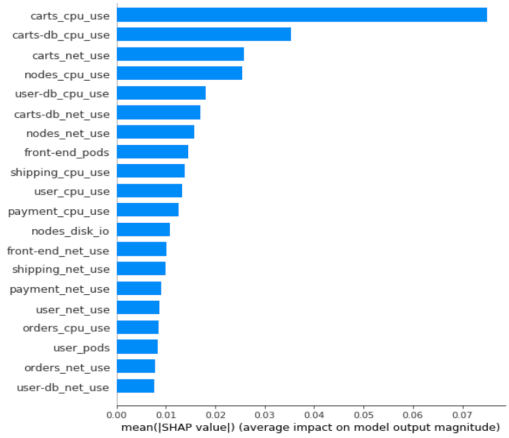

Figure 10. Features impact to neural model prediction output magnitude

The prior effort made by [6] studied also the feasibility of utilizing the proposed performance models in making efficient resource scaling decisions by formulating a constraint nonlinear optimization

problem. The suggested problem sought the maximum CPU utilization of all services involved in a workflow that meets a desirable Service Level Objective (SLO) goal. Theoretically, our exploration to obtain a comparable latency prediction accuracy could contribute to extending the suggested optimization problem by using our DNN predicting model. The solution could consider additional performance characteristics in addition to CPU utilization.

Practically, being able to predict response latency with a neural network leveraging a comprehensive set of metrics or with other machine learning techniques based on selected relevant features would suggest to Service Reliability Engineers (SREs) to observe appropriate metrics to meet their services SLO target. Furthermore, the method used in this research to monitor and collect metrics from applications run in a Kubernetes cluster is practical for practitioners as it used available open-source tools and techniques.

6. Conclusions

In this research, we used a variety of observed performance metrics to evaluate the accuracy of different machine learning techniques to predict end-to-end response latency in microservice application flow. Inspired by the work performed by [6], we observed and collected different performance metrics from the underlying platform cluster nodes and all microservices involved in executing a purchasing order request in a microservice application called Sock Shop deployed on a cloud Kubernetes cluster. We then examined different machine learning performance models to evaluate predicting end-to-end response latencies based on the collected metrics data. We were able to obtain a high prediction accuracy using all collected features by using a Deep Neural Network model.

To the extent of our knowledge existing work used only CPU utilization on the microservices and infrastructure VMs levels to predict tail end-to-end response latencies. We added more performance indicators to perform the prediction. In addition to the CPU utilization of the microservices and cluster nodes, we examined the effect of network traffic utilization in the microservices and the cluster nodes. The time consumed by microservices to perform reading and writing data through the network may cause a response delay. Further, we considered also disks input/output (I/O) operations on the cluster nodes level. Disk I/O could occur from operations performed by database queries in the microservices, reading or writing files. Delays on performing reading/writing from/to a disk could affect a response latency as well. Last but not least, we also put in our consideration the factor of the number of instances of each microservices. A load balancer may route requests to congested microservice instances that would cause inverse effects in response latency rates. Moreover, we considered using the request rate in the experiment workload instead of the number of concurrent users as another impacting factor to predict the response latency. It is also worth noting that, we observed the metrics of the platform nodes on a cloud Kubernetes cluster level instead of on the infrastructure VMs lower level.

Our results conformed to the fact that the relationship between the metrics features and response latency is nonlinear. This observation stemmed from the poor prediction accuracy of the Linear Regression algorithm. Cutting down the number of metrics by selecting relevant ones to the response latency enabled us to obtain good results with second-degree Polynomial Linear Regression, K-Nearest Neighbors, Decision Tree and Support Vector Regression machine learning techniques. The prediction was enhanced more by using ensemble machine learning techniques such as Random Forrest and eXtream Gradient Boosting (XGBoost). We were able to use all the performance features with a Deep Neural Network model and obtained the highest prediction accuracy.

The findings could help guide reliability engineers to consider observing relevant features to predict response latency per a specific use case. Software operation practitioners might have automation in place to automatically rollback a newly deployed version if observed metrics resulted in increased latency. Automation could also help to automatically scale down or up a service to achieve an acceptable latency threshold. Also, software engineers might detect bug issues alerted by such observed latency. Furthermore, as we were able to manage the CPU resource amount in a Kubernetes Pod, we envision to also control having adequate network bandwidth to achieve obtaining acceptable latency threshold. Software practitioners can leverage the experimental bandwidth plugin feature [28] in Kubernetes Pod to constrain a deployed microservice instance with a certain amount of network bandwidth. Hence, the platform would be able to scale based on the network resource utilization as well to meet a certain latency SLO target.

Our effort could also suggest to researchers a modified efficient resource scaling problem to solve. We envision a further research opportunity to expand on investigating this problem to be based on multiple variables such as CPU and network utilization, with probably other potential indicators, such that to avoid violating a target SLO of response latency. We would like to mention that our attempt was limited to one

flow in an online shopping application. We look forward to expanding it to other application domains while running the experiment with a load applied to a different workflow. Last but not least, we considered predicting current latency, we are planning to try forecasting future response latency based on prior history of microservices performance metrics.

7. References

[1] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud Container Technologies : a State-of-the-Art Review," vol. 7161, no. c, pp. 1–14, 2017.

[2] Kubernetes, "an open-source system for automating deployment, scaling, and management of containerized applications.".

[3] L. P. Dewi, A. Noertjahyana, H. N. Palit, and K. Yedutun, "Server Scalability Using Kubernetes," *TIMES-iCON 2019 - 2019 4th Technol. Innov. Manag. Eng. Sci. Int. Conf.*, pp. 1–4, 2019.

[4] Docker, "Containerization technology helps development teams build and ship applications.".

[5] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open Issues in Scheduling Microservices in the Cloud," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 81–88, 2016.

[6] J. Rahman and P. Lama, "Predicting the end-to-end tail latency of containerized microservices in the cloud," *Proc. - 2019 IEEE Int. Conf. Cloud Eng. IC2E 2019*, pp. 200–210, 2019.

[7] Jialin Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Hardware, OS, and Application-level Sources of Tail Latency," *Chron. High. Educ.*, vol. 52, no. 20, 2006.

[8] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 9, pp. 2101–2116, 2019.

[9] L. Wang, J. Xu, H. A. Duran-Limon, and M. Zhao, "QoS-driven cloud resource management through fuzzy model predictive control," *Proc. - IEEE Int. Conf. Auton. Comput. ICAC 2015*, pp. 81–90, 2015.

[10] A. Sill, "The Design and Architecture of Microservices," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 76–80, 2016.

[11] H. Silveira and M. Sundaram, "A Microservice Based Reference Architecture Model in the Context of Enterprise Architecture," *2016 IEEE Adv. Inf. Manag. Commun. Electron. Autom. Control Conf.*, pp. 1856–1860, 2016.

[12] C. Esposito, A. Castiglione, and K. K. R. Choo, "Challenges in Delivering Software in the Cloud as Microservices," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 10–14, 2016.

[13] X. Liu, S. Jiang, X. Zhao, and Y. Jin, "A shortest-response-time assured microservices selection framework," *Proc. - 15th IEEE Int. Symp. Parallel Distrib. Process. with Appl. 16th IEEE Int. Conf. Ubiquitous Comput. Commun. ISPA/IUCC 2017*, pp. 1266–1268, 2018.

[14] D. Géhberger, P. Mátray, and G. Németh, "Data-driven monitoring for cloud compute systems," *Proc. - 9th IEEE/ACM Int. Conf. Util. Cloud Comput. UCC 2016*, pp. 128–137, 2016.

[15] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," *Proc. - 2019 IEEE Int. Conf. Web Serv. ICWS 2019 - Part 2019 IEEE World Congr. Serv.*, pp. 68–75, 2019.

[16] P. Tennage, S. Perera, M. Jayasinghe, and S. Jayasena, "An analysis of holistic tail latency behaviors of java microservices," *Proc. - 21st IEEE Int. Conf. High Perform. Comput. Commun. 17th IEEE Int. Conf. Smart City 5th IEEE Int. Conf. Data Sci. Syst. HPCC/SmartCity/DSS 2019*, pp. 697–705, 2019.

[17] Sock Shop, "open-source demonstration online shopping microservices application." [Online]. Available: https://github.com/microservices-demo/microservices-demo.

[18] Linode, "managed Kubernetes service."

[19] Istio, "open source service mesh that layers transparently onto existing distributed applications."

[20] Locust, "open source load testing tool." [Online]. Available: https://locust.io/.

[21] Prometheus, "open-source metrics and alerting monitoring tool.".

[22] Postgres, "open source relational database." [Online]. Available: https://www.postgresql.org/.

[23] Weavescope, "a visualization and monitoring tool for Docker and Kubernetes." [Online]. Available: https://www.weave.works/docs/scope/latest/introducing/.

[24] Jupyter, "web-based interactive development and computing environment." [Online]. Available: https://jupyter.org/.

[25] Scikit-learn, "Python open-source machine learning tools." [Online]. Available: https://scikit-learn.org/.

[26] Keras, "Python Deep learning API." [Online]. Available: https://keras.org/.

[27] S. M. Lundberg and S.-I. Lee, "A Unified Approach to Interpreting Model Predictions Scott," *Nips*, vol. 16, no. 3, pp. 426–430, 2012.

[28] N. Plugin, "Kubernetes Network Plugin." [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.