BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services

Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, Gabor Karsai
Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, Tennessee, USA
Email: {anirban.bhattacharjee; ajay.d.chhokra; zhuangwei.kang; hongyang.sun; a.gokhale; gabor.karsai}@vanderbilt.edu

Abstract—Pre-trained deep learning models are increasingly being used to offer a variety of compute-intensive predictive analytics services such as fitness tracking, speech and image recognition. The stateless and highly parallelizable nature of deep learning models makes them well-suited for serverless computing paradigm. However, making effective resource management decisions for these services is a hard problem due to the dynamic workloads and diverse set of available resource configurations that have different deployment and management costs. To address these challenges, we present a distributed and scalable deep-learning prediction serving system called Barista and make the following contributions. First, we present a fast and effective methodology for forecasting workloads by identifying various trends. Second, we formulate an optimization problem to minimize the total cost incurred while ensuring bounded prediction latency with reasonable accuracy. Third, we propose an efficient heuristic to identify suitable compute resource configurations. Fourth, we propose an intelligent agent to allocate and manage the compute resources by horizontal and vertical scaling to maintain the required prediction latency. Finally, using representative real-world workloads for an urban transportation service, we demonstrate and validate the capabilities of Barista.

Keywords—Resource Management, Machine Learning Models, Predictive Analytics, Serverless Computing, Containers

I. INTRODUCTION

A. Emerging Trends

Cloud-hosted, predictive analytics services based on *pre-trained deep learning models* [1] have given rise to a diverse set of applications, such as speech recognition, natural language processing, fitness tracking, online recommendation systems, fraudulent behavior detection, genomics, and computer vision. End-users of these services query these pre-trained models using an interactive web or mobile interface through RESTful APIs. Based on the supplied input, these pre-trained models infer the target values and return the prediction results to the end-users. As an example, a speech recognition system transcribes spoken words into text [2].

These prediction services are usually containerized and encapsulated with all the required software packages [3]. Thus, deployer of these services can preferably use the *function-as-a-service* (FaaS) approach to hosting these services in an event-driven manner, wherein the functions are executed on the occurrence of some trigger or event (e.g., incoming request). This overall approach can be handled using *serverless computing* [4], since the service creator needs only to provide the function logic, the trigger conditions, and the *service level objectives* (SLOs), such as latency bounds, which are on order of few seconds. It is then the responsibility of the serverless platform provider to provide the hosting environment for these services and to ensure that the SLOs are met.

B. Challenges and State-of-the-Art

The execution environments for deep learning-based prediction services typically comprise containers running on a cluster of virtual machines (VMs). These prediction services are usually stateless, parallelizable (multi-threaded), and compute-intensive. The model sizes of these prediction services are large (hundreds of megabytes to gigabytes), which takes a significant time to load them into the containers and provision the infrastructure. Moreover, due to the parallelizable nature of these models, their running times can be substantially reduced by assigning more CPU cores (see Figure 1). However, allocating more memory only marginally improves the running times. Thus, a naïve approach to assuring SLOs is to over-provision the service infrastructure; however, doing so imposes unnecessary costs on the serverless provider. Efficient management of computing resources dynamically is required to minimize the cost of hosting these services in the cloud [4]–[6].

Although a substantial amount of literature exists on finding the sweet spot between resource over-provisioning (which wastes resources and increases cost) and under-provisioning (which violates SLOs) [7]–[9], these works focus primarily on long-running analytics jobs for which the goal is to find optimal configurations to meet the SLOs and scale the resources dynamically to handle their variable workloads.

In contrast, the prediction services are latency sensitive and have short running times. Moreover, the incoming request patterns (workloads) can fluctuate significantly and follow a diurnal model, which requires rapid management of resources to meet the variable workload demands. A reactive approach is not suitable as the prediction latency may increase significantly due to infrastructure provisioning time (e.g., an order of minutes due to VM creation and model loading times). Hence, the desired solution is one that can forecast the workload patterns and can estimate the required resources for the application to maintain the SLOs under the forecasted workload. Determining the right cluster configurations and dynamically allocation of resources are hard due to the variability of cloud configurations (VM instance types), the number of VMs, and their deployment and management costs [7], [10].

Only recently have some solutions started to emerge to address these concerns [5], [11]. Nevertheless, there remain many unresolved problems. First, current horizontal elasticity

solutions for prediction services often do not account for the container-based service lifecycle states (e.g., whether the VM is already up or not, or whether the container is running, and if so whether the model is loaded or not). Each such state incurs a hosting cost and impacts the running time. Second, vertical elasticity solutions for containers do not yet fully exploit the parallelizable aspects of the pre-trained models. Third, proactive resource scaling decisions require effective workload forecasting and must be able to monitor the service lifecycle states, both of which are missing in prior efforts [12], [13]. Finally, existing strategies for container allocation tend to overlook performance interference issues from other co-located containerized services, which may cause unpredictable performance degradation [14].

C. Solution Approach

To address these unresolved problems, we present a serverless framework, called Barista, which hosts containerized, pre-trained deep learning models for predictive analytics in the cloud environment, and meets the SLOs of these services while minimizing hosting costs. Barista comprises an efficient, data-driven and scalable resource allocator, which estimates the resource requirements ahead of time by exploiting the variable patterns of the incoming requests, and a forecast-aware scheduling mechanism, which improves resource utilization while preventing physical resource exhaustion. We show how serverless computing concepts for dynamically scaling the resources vertically and horizontally can be utilized under different scenarios. Barista efficiently and cost-effectively provisions (scale-up and scale-down) resources for a prediction analytics service to meet its prediction latency bounds. Specifically, we make the following contributions:

1) **Workload Forecast:** We propose a hierarchical methodology to forecast the workload based on historical data.
2) **Resource Estimation:** Barista allows service providers to communicate the performance constraints of their service models regarding their SLOs. An analytical model is provided to estimation the resources based on latency bound, workload forecasting, and the profiled execution time model on different cloud configurations.
3) **Serverless Resource Allocation:** Barista provides a novel mechanism using the serverless paradigm to allocate resources proactively based on the difference between resource requirement estimation and current infrastructure state in an event-driven fashion.

D. Organization of the Paper

The rest of the paper is organized as follows: Section II presents a survey of existing solutions in the literature and compares them with Barista; Section III presents the problem formulation; Section IV presents the design of Barista; Section V evaluates the Barista resource allocator for a prototypical case study; and finally, Section VI presents concluding remarks alluding to future directions.

II. BACKGROUND AND RELATED WORK

This section provides an overview of the literature along the dimensions of deep learning-based prediction services, infrastructure elasticity, serverless computing, and workload forecasting, all of which are critical for the success of the presented work on Barista.

A. Deep Learning-based Prediction Services

Pre-trained models based on deep learning techniques are increasingly being used in prediction analytics services. In this approach, all the learned internal parameters are stored in the form of a vector of scores for each category along with their weights in a pre-trained deep learning model of desired accuracy [1]. Once the models are trained, these prediction models are seamlessly integrated into applications to predict outcomes based on new input data [15].

B. Serverless Computing

Serverless computing focuses on providing zero administration by automating deployment and management tasks. In this paradigm, the responsibility of deployment and management is delegated to a cloud infrastructure provider or a mediating entity. The execution platform leverages container technology to deploy and scale the latency- sensitive prediction service components, which helps to minimize idle resource capacity [16]. These features are beneficial to the design and deployment of parallelizable deep learning prediction services.

However, due to variation in workloads, the providers of prediction services are required to modify their resource requirements by monitoring the resources continuously [13], and that reactive approach can often violate the SLOs. The MxNet deep learning framework [17] shows the feasibility of using serverless computing AWS Lambda framework [18]. There are several efforts [19]–[23] to deploy and orchestrate VMs or containers dynamically with all software dependencies. Barista's focus is orthogonal to these efforts; Barista intelligently and efficiently manages the containerized allocation based on resource estimation by workload forecasting and profiling the execution time of prediction services.

C. Dynamic Infrastructure Elasticity

Most state-of-the-art technology and research strategies to *horizontally* or *vertically* scale the resources are heuristics-driven or rule-based and have custom triggers. Selecting optimal cloud configurations is an NP-hard problem, and various models are presented based on heuristics [14], [24]. In Barista, we lease required VMs from the cloud provider to meet the latency bounds based on time series forecasting of the incoming workloads. We propose an efficient heuristic to select the configuration of VM types to guarantee bounded prediction latency while minimizing the cost.

Swayam [5] presented a short-term predictive provisioning model to guarantee SLO while minimizing resource waste. However, they only consider horizontal scaling by allocating more backend containers from the resource pool. Model loading time for deep learning models is significant, especially when the container is in cold state [17]. In contrast, Barista proactively considers infrastructure provisioning time to scale the system and also allows vertical resource scaling.

Vertical elasticity adds flexibility as it eliminates the overhead in starting a new VM and loading the service model. Prior
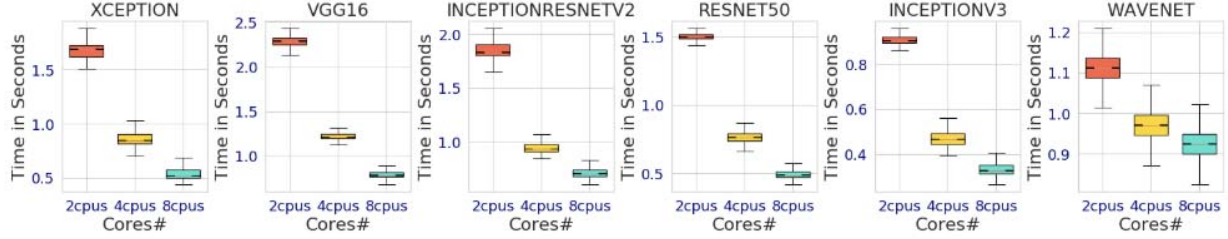
24

Fig. 1: Box plots of prediction times for different deep learning pre-trained models on different numbers of CPU cores (2, 4 and 8).

efforts to scale the CPU resources vertically appear in [25], [26] including an approach that uses the discrete-time feedback controller leveraging MAPE-K loop for containerized applications [27]. Barista uses an efficient, proactive method to trigger the scaling of resources horizontally while relying on vertical scaling reactively to allocate and de-allocate CPU cores for model correction when our estimation model cannot predict accurately. Our reactive approach can also handle sudden workload spikes within a threshold.

D. Workload Forecasting

Workload forecasting is indispensable for service providers to anticipate changes in resource demands and make proactive resource allocation decisions. Various forecasting methods based on time series analysis are described in [28]. Dejavu [29] and Bubble-Flux [30] proposed self-adaptive resource management algorithms, which leverage workload prediction and application performance characterization to predict resource requirements. These efforts employ a linear model for workload prediction, which often results in low-quality forecasts with high uncertainty.

Several non-linear methodologies based on Support Vector Machine [31], Error Correction Neural Network (ECNN) [32], Gaussian processes [3], [33] are proposed to predict workloads. However, these models fail to capture longer-term trends, which are key traits of cloud-hosted services [34]. In [34], a hybrid model called *Prophet* is proposed for forecasting workloads by combining linear/logistic trend models with a Fourier series-based seasonality model. According to the authors, Prophet is easier to use than the widely used ARIMA models [27], [35] as it handles missing values automatically. Moreover, the ARIMA model generally struggles to produce good quality forecasts as it lacks seasonality detection.

In general, workload forecasting methods tend to lack feedback to update predictions based on recent performance. Therefore, Barista extends Prophet with a non-linear decision-based model that modifies the forecast according to previous prediction errors. Barista workload forecasting model estimates the resource requirement and proactively scales the infrastructure to guarantee application SLOs.

III. SYSTEM MODEL AND PROBLEM DESCRIPTION

This section first describes the infrastructure model and assumptions and then presents the problem formulation, which has two subproblems. First, given the SLO of the service,

the properties of the pre-trained model and the costs of VM configurations, we determine the cost-effective VM types by solving an optimization problem to meet the SLO (Section III-B). We then consider the dynamic management of VM and container resources through workload forecasting and infrastructure elasticity (Section III-C).

A. Infrastructure Model and Assumptions

To explore the *serverless capabilities*, we assume that the deep learning pre-trained model for a predictive analytics service is encapsulated inside containers which are executed in a cluster of VMs. All the requests to the service are assumed to be *homogeneous*, i.e., they execute the same prediction model, and the service is stateless. Since deep learning models are generally compute-intensive, they benefit from executing on multiple cores. We validate this claim in Figure 1, which shows the range of prediction time latencies for several pre-trained deep-learning models on a VM hosted on AMD Opteron 2300 (Gen 3 Class Opteron) physical machine with different numbers of assigned CPU cores, demonstrating good speedup behaviors. The results are obtained by running 10,000 trial executions in isolation for each model.

The VM lifecycle in cloud infrastructure for service deployment and management is considered as follows.

① VM Cold: VM has not been deployed.
② VM Warm: VM is deployed, but the container inside the VM has not been downloaded.
③ Container Cold: the container is downloaded, but the pre-trained deep-learning model has not been loaded into the container's memory.
④ Container Warm: the deep-learning model is loaded, and the container is ready to serve the prediction requests.

Finally, we assume that once a VM is deployed, it is leased from the cloud provider for a minimum duration of τ_{vm} time. During this time, the deployment cost is paid for even if the VM is not used (because we do not scale down the system immediately). This could happen when the prediction model is unloaded from the container's memory during lightly-loaded periods so that the VM could serve other batch jobs in the background (e.g., deep analytics or map-reduce applications).

B. VM Flavor Selection and Initial Deployment

We first consider the static problem of serving a fixed set of requests that execute a deep-learning prediction model by finding the most cost-effective VM flavor type. Let λ denote

the constraint specified by the SLO of the model regarding its execution latency,[1] and let t_p denote the latency when the model is executed on p CPU cores. Further, let min_mem denotes the minimum amount of memory required to run a prediction model.

To serve the prediction requests, we need to deploy a set of VMs from the cloud provider that offers a collection of m possible configurations (flavors), denoted as $\{vm_1, vm_2, \ldots, vm_m\}$. We consider three parameters to specify each configuration vm_i: (1) p_i, the number of available cores; (2) mem_i, the memory capacity; and (3) $cost_i$, the cost of deployment. In particular, the cost includes both the running cost and the management cost of deploying the VM. For each configuration vm_i, suppose $\alpha_i \in \{0, 1, 2, \ldots, \}$ number of VMs are deployed. Then, the total deployment cost is given by $total\_cost = \sum_{i=1}^{m} \alpha_i \cdot cost_i$. While each deployed VM is assumed to serve only one request at a time (because each request benefits from consuming all the cores as the prediction service is highly-parallelizable), it could serve multiple requests one after another. In this case, the request that is served later in the pipeline needs to wait for the preceding requests to be first completed, which will delay its prediction time.

The optimization problem concerns deploying a set of VMs, i.e., to choose α_i's for all $i = 1, \ldots, m$, so that the total cost is minimized while the SLOs of all the requests can be satisfied (i.e., with latency no larger than λ). Section IV-D presents our VM deployment solution.

C. Dynamic Resource Provisioning via Workload Forecasting and Infrastructure Elasticity

The resources must be provisioned dynamically to meet the SLOs under workload variations. Since the reactive approach can be detrimental to response times, we use a proactive approach to handle the variation in the workload by forecasting the future service demands based on historical workload patterns of a deep-learning prediction model.

With the forecasted future workload (i.e., number of service requests), the VM deployment decision (as described in Section III-B) must also be adapted accordingly. Horizontal scaling [36] is a promising approach to provision the resources dynamically. To that end, we exploit the four cloud infrastructure states described earlier. Figure 2 illustrates the actions needed to transition between the states. Each action incurs a state transition time. Specifically, we denote the VM deployment time by t_{vm}, the container service download time by t_{cd}, and the pre-trained model loading time by t_{ml}.[2]

Figure 3 shows concrete timings for the different prediction services we tested on our experimental infrastructure. Further, we refer to the total time to set up the service as $t_{setup} = t_{vm} + t_{cd} + t_{ml}$. This motivates the need to forecast

---

[1] Depending on the SLO, the execution latency can be flexibly defined based on, e.g., worst-case latency or x-percentile latency. In this paper, we consider the 95th percentile latency.

[2] The time required to unload the model from memory, denoted by t_{mu}, is negligible and thus not considered. The time taken to move from any state to VM Cold is denoted by t_{exp}. This duration does not impact the resource manager logic and hence is also ignored.
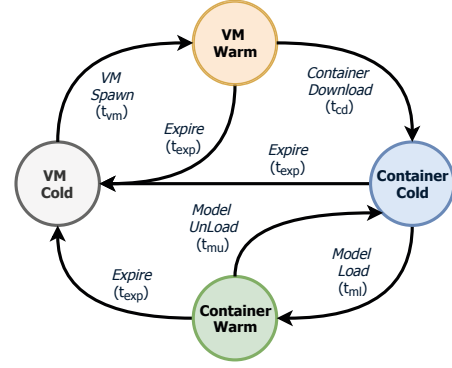

Fig. 2: An abstract state machine showing different states and transitions associated with a life cycle of a VM in cloud infrastructure. Edges are labeled with actions and time duration to complete the state transition.

the future workload $t'_{setup} = t_{setup} + t_{forecast}$ time ahead to guarantee the SLO with certain accuracy, where $t_{forecast}$ is the time taken to obtain the forecast. The forecasting needs to be performed for t'_{setup} time into the future to account for the infrastructure setup time.
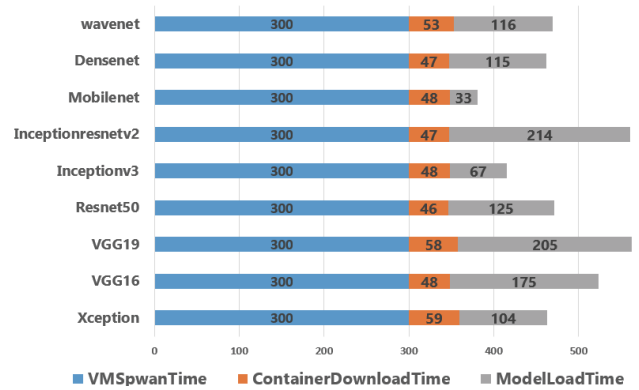

Fig. 3: The setup times (in seconds) for different deep-learning prediction models as per our experiment. The blue bar shows VM deployment time (t_{vm}), orange bars show the specific pre-trained model container download time (t_{cd}), and grey bar show prediction model loading time (t_{ml}).

When the VM is not serving any requests, the infrastructure state transitions from Container Warm to Container Cold. Later on, when the load increases and the VM needs to serve requests again, the model will be reloaded. As a result, we also need to check the infrastructure state ahead of time to make decisions for downloading the container and loading the model if it is not already in the Container Warm state. We account for all of these times in meeting the SLOs, while avoiding excess over-provisioning of the infrastructure resources. Section IV-E presents our combined solution to the dynamic resource management problem that incorporates infrastructure elasticity, workload forecasting, and VM deployment.

Moreover, if the workload forecaster over-estimates the workload, we allow the excess resources to be utilized by the low-priority batch jobs via vertical scaling. Co-locating

various jobs on a server can cause performance interferences. In our approach, we assume 10% performance degradation (for the worst-case scenario based on our experiment) if a latency-sensitive prediction service is co-hosted with batch jobs.

IV. DESIGN & IMPLEMENTATION OF BARISTA

In this section, we give the architectural insights of Barista by describing its various components. We also explain our solutions to the problems of static VM deployment and dynamic resource provisioning as mentioned in Section III.

A. Architecture of Barista

Barista architecture consists of a pool of frontend and backend servers, load balancers to distribute the requests, a platform manager to allocate and scale backends for different prediction services as shown in Figure 4. Frontend servers are the virtual machines, which host the user interface, whereas backend servers host the containerized pre-trained deep learning model. End users send their requests to the frontend load balancer, which redirects the requests to the frontend servers based on the *round robin* policy. Each frontend server forwards the request to the backend load balancer, which then redirects the request to one of the backend servers assigned to serve the prediction query based on the *least loaded connection* policy. Each backend server processes a single request at a time and gives the prediction result back. The platform manager is an integral part of Barista, which is responsible for dynamic provisioning of resources in cloud infrastructure by forecasting workload patterns and estimating the execution time of various prediction queries. The platform manager (as zoomed in Figure 4) consists of a prediction service profiler, a request monitor, a request forecaster, a prediction latency monitor, and a resource manager. They are described as follows:

① **Prediction Service Profiler:** It profiles the execution time of a prediction service on different numbers of CPU cores and finds the best distribution (as shown in Figure 5). This provides the 95th percentile latency estimate of the execution time for the prediction service based on the assigned number of cores.

② **Request Monitor:** It monitors and logs the number of aggregated incoming requests received every minute by the backend load balancer.

③ **Request Forecaster:** It predicts the number of requests infrastructure provisioning time steps into the future based on the historical data. The forecaster also updates its model every minute based on the previous prediction errors and the actual data from the request monitor to diminish uncertainty.

④ **Prediction Latency Monitor:** It monitors and logs the SLO violations for the incoming requests every five seconds. SLO is defined over the response time of the backend servers to a prediction query requested by the frontend servers.

⑤ **Resource Manager:** It allocates the required number of virtual machines for the forecasted workloads and

performs intelligent scaling based on resource estimation and provisioning strategies as discussed in Section IV-D.
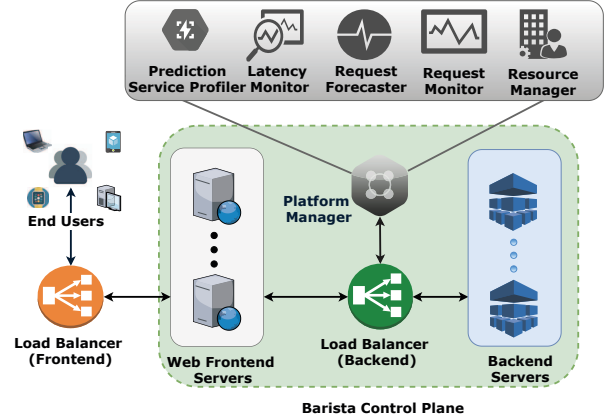

Fig. 4: Architecture of Barista serving system.

The operation of the platform manager can be categorized into two phases, *online* and *offline*, as shown in Figure 5. In the offline or design phase, the execution time of the deep learning model is profiled on different VM configurations followed by distribution estimation. The workload forecasting model is also trained in this phase. In the online or runtime phase, based on the output of workload forecaster and execution time estimator, resources are estimated and provisioned. The actual workload is also being monitored and stored, which is used to update the forecasting model based on the last five error predictions and rolling training window. The following subsections provide more details on these operations.
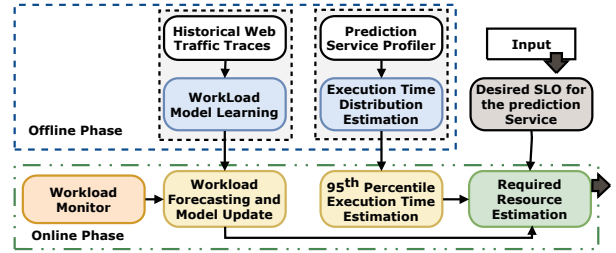

Fig. 5: Data flow model of Barista platform manager.

B. Execution Time Distribution Estimation

An inaccurate estimate of the execution time of a pre-trained deep learning model can result in erroneous output produced by the resource manager, which may lead to over- or under-provisioning of resources. Thus, in Barista, extensive offline profiling of different deep learning models is performed on various VM configurations. Figure 1 shows our experiments on configurations involving 2, 4 and 8 CPU cores with required memory size, where each experiment contains 10,000 trails. The execution times are random in nature and follows an unknown distribution. In Barista, the resource manager uses the 95th percentile statistic of the execution time to provision

27

resources. In order to accurately calculate the percentile values, we remove sample bias and estimate the unknown distribution.

Barista prediction service profiler estimates the distribution using non-parametric methods and compares the sample with different reference probability distributions iteratively. For quantifying the goodness of fit, we use one-sample Kolmogorov-Smirnov (K-S) test [37] to rank different hypothesized distributions. Given the cumulative distribution function F_0 of the hypothesized distribution and the empirical distribution function $F_{data}(x)$ of the observed dataset, the test statistic (D_n) can be calculated by Equation (1), where sup_x is the *supremum* of the set of distances and n is the size of the data. According to [38], if the sample comes from distribution $F_0(x)$, then D_n converges to 0 almost surely in the limit when n goes to infinity.

$$D_n = sup_x |F_0(x) - F_{data}(x)| \qquad (1)$$

C. Workload Forecasting

Barista uses an online rolling window based forecasting methodology to predict workload to allocate resources proactively. Request forecaster is composed of two main components: ① *Forecaster*, which is responsible for modeling both periodic and non-periodic elements associated with time-varying workloads, and ② *Compensator*, which modifies the forecast produced by the first component according to the last five forecast errors. These two components are briefly described as follows:

1) Forecaster: A time-varying workload can be composed of three main elements - *trend*, *seasonality* and *holidays* [34], [39], and they are combined as shown in Equation (2) below:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t \qquad (2)$$

where $g(t)$, $s(t)$, $h(t)$ model non-periodic changes, periodic changes (e.g., daily, weekly and yearly seasonality), and effects of holidays which occur on potentially irregular schedules over one or more days, and ϵ_t represents noise.

The trend function ($g(t)$) models how the workload has grown in the past and how it is expected to continue growing. Modeling web traffic is often similar to population growth in natural ecosystem, where there is a non-linear growth that saturates at carrying capacity [34]. This kind of growth is typically modeled using the logistic function shown in Equation (3), where C is the carrying capacity, k is the growth rate, and m is an offset parameter.

$$g(t) = \frac{C}{1 + \exp(-k(t - m))} \qquad (3)$$

The seasonality function ($s(t)$) models multi-period seasonality that repeats after a certain period. For instance, a five day work week can produce effects on a time series that repeat each week. A standard Fourier series, as shown in Equation (4), is used to provide a flexible model of periodic effects [40], where P is the expected time series period and N is the order.

$$s(t) = \frac{1}{2}a_0 + \sum_{n=1}^{N} \left[a_n \cos(\frac{2\pi nt}{P}) + b_n \sin(\frac{2\pi nt}{P}) \right] \qquad (4)$$

The holiday function ($h(t)$) models the predictable variations in workload caused due to holidays. However, these variations do not follow a periodic pattern, so their effects are not well modeled by a cycle. The holidays are added in the form of a list and are assumed independent of each other. An indicator function is added for each holiday that shows the effect of a given holiday on the forecast. Barista leverages Prophet [34] for implementing Forecaster.

2) Compensator: This component adjusts the output of the forecaster based on the past forecast errors. It can be modeled as a transformation function c, which changes the output of the forecaster y based on the errors from the last m forecasts $E = \{e_1, e_2, \ldots, e_m\}$, as shown in Equation (5) below:

$$y' = c(y, y_{upp}, y_{low}, E) \qquad (5)$$

where y_{upp} and y_{low} are the upper and lower estimation bounds of y. The transformation can be learned using data-driven methods. In Barista, we use H2O's AutoML framework [41] to find the best hyper-parameter tuned algorithm.

D. Resource Estimation

Resource estimation is one of the two main tasks performed by the resource manager. Depending upon the forecasted value, the SLO and the service type, the resource manager solves the static VM deployment problem described in Section III-B. Due to the NP-hardness of the problem, this subsection presents a greedy heuristic to perform static VM deployment.

For each VM configuration vm_i, we can compute the number of requests n_req_i it is able to serve for a deep learning prediction service while meeting the SLOs:

$$n\_req_i = \begin{cases} \lfloor \frac{\lambda}{t_{p_i}} \rfloor, & \text{if } mem_i \geq min\_mem \\ 0, & \text{otherwise} \end{cases}$$

Recall that λ is the model's SLO timing constraint, min_mem is the model's minimum memory requirement, t_{p_i} is the latency to serve each request of the model using configuration vm_i with p_i CPU cores, and mem_i is amount of memory available in vm_i. We can then define the *cost per request* for each configuration vm_i as follows:

$$cpr_i = \frac{cost_i}{n\_req_i}$$

Let i^* denote the index of the VM configuration with the minimum cost per request, i.e., $cpr_{i^*} = \min_{i=1 \ldots m}\{cpr_i\}$. Clearly, given an estimated workload y' from the output of Equation (5), an optimal rational solution will deploy $\alpha^* = \frac{y'}{n\_req_{i^*}}$ VMs of configuration vm_{i^*} and incurs a total cost:

$$total\_cost^* = \frac{y'}{n\_req_{i^*}} \cdot cost_{i^*} \qquad (6)$$

To find the optimal integral solution is unfortunately NP-hard (via a simple reduction from the knapsack or the subset sum problem). However, Equation (6) nevertheless serves as a lower bound on the optimal total cost.

To solve the integral problem, our greedy algorithm also chooses a single configuration vm_{i^*} that has the minimum

cost per request while breaking ties by selecting the configuration with a smaller deployment cost. Thus, it deploys $\alpha = \lceil \frac{y'}{n\_req_{i*}} \rceil$ VMs of configuration vm_{i*} for serving y' requests, and incurs a total cost that satisfies:

$$
\begin{aligned}
total\_cost &= \lceil \frac{y'}{n\_req_{i*}} \rceil \cdot cost_{i*} \\
&< \left(\frac{y'}{n\_req_{i*}} + 1 \right) \cdot cost_{i*} \\
&= total\_cost^* + cost_{i*} \qquad (7)
\end{aligned}
$$

Equation (7) shows that the total cost of the greedy algorithm is no more than the optimal cost plus an additive factor $cost_{i*}$. When serving a large number of requests, the incurred cost is expected to be close to the optimal. Furthermore, the algorithm always deploys VMs from the same configuration regardless of the number of requests to be served. This makes it an attractive solution for handling dynamic workload variations without switching between different VM configurations. The complete algorithm is illustrated in Algorithm 1.

Algorithm 1: Resource Estimation

1 Initialize: $i^* \leftarrow 0$, $cpr^* \leftarrow \infty$, $cost^* \leftarrow \infty$, $n\_req^* \leftarrow 0$
2 **for** $i = 1$ *to* m **do**
3    $t_{p_i} \leftarrow$ getExecutionTime(vm_i, $model$)
4    $mem_i \leftarrow$ getMemory(vm_i)
5    $cost_i \leftarrow$ getCost(vm_i)
6   **if** $mem_i \geq min\_mem$ **then**
7       $n\_req_i \leftarrow \lfloor \frac{\lambda}{t_{p_i}} \rfloor$
8       $cpr_i = \frac{cost_i}{n\_req_i}$ ▷ Cost per request
9   **if** $cpr_i < cpr^*$ **then**
10          $i^* \leftarrow i$
11          $cpr^* \leftarrow cpr_i$
12          $n\_req^* \leftarrow n\_req_i$
13          $cost^* \leftarrow cost_i$
14   **else if** $cpr_i = cpr^*$ & $cost_i < cost^*$ **then**
15          $i^* \leftarrow i$
16          $n\_req^* \leftarrow n\_req_i$
17          $cost^* \leftarrow cost_i$
18   **end**
19   **end**
20 **end**
21 Deploy $\alpha \leftarrow \lceil \frac{y'}{n\_req^*} \rceil$ VMs of configuration vm_{i*}

E. Resource Provisioner

Resource provisioning is a critical process running inside Barista resource manager. Its main objective is to scale the resources proactively to handle the dynamic workload. Barista resource provisioner intelligently frees acquired resources for latency-sensitive services when the predicted workload is low, i.e., horizontal scaling down. And when the workload increases, the resource provisioner acquires new resources while taking into account already freed resources. We use two sets, κ and ψ, to indicate VMs that are used for latency-sensitive prediction services. The VMs in κ denote resources that are actively used for serving latency-sensitive prediction services while the VMs in ψ are those that have been freed by resource provisioner.

Algorithm 2: Resource Provisioning

1 Initialize: $Flag \leftarrow True$, $\alpha \leftarrow 0$, $n\_req_{i*} \leftarrow 0$, $i^* \leftarrow 0$, $params \leftarrow \{model, \lambda, min\_mem, mem_i, p_i, cost_i, \forall i = 1 \ldots m\}$, $\kappa \leftarrow \emptyset$, $\psi \leftarrow \emptyset$, H_{cdl}, H_{mld}, H_{exp}
2 **while** $True$ **do**
3    $t \leftarrow$ GetCurrentTime()
4    $y' \leftarrow$ GetForecast(t, t'_{setup})
5   **if** $Flag$ **then**
6       $i^*, n\_req_{i*} \leftarrow$ ResourceEstimation($params$)
7       $Flag \leftarrow False$
8   **end**
9    $\alpha \leftarrow \lceil \frac{y'}{n\_req_{i*}} \rceil$
10    $C^\kappa_{exp}, C^\psi_{exp} \leftarrow$ GetExpireVMCount($t + t'_{setup}, \kappa, \psi$)
11    $\delta \leftarrow \alpha - (|\kappa| - C^\kappa_{exp})$
12   **if** $\delta > 0$ **then**
13       $\delta_{scaled} \leftarrow \min \left(|\psi| - C^\psi_{exp}, \delta \right)$
14       $\delta_{new} \leftarrow \max \left(0, \delta - \delta_{scaled} \right)$
15   **for** $i = 1$ *to* δ_{new} **do**
16          $IP \leftarrow$ DeployVM(i^*)
17   κ.add(IP)
18          $H_{cdl}[t + t_{vm}] = (IP, model)$
19          $H_{mld}[t + t_{vm} + t_{cd}] = (IP, model)$
20          $H_{exp}[t + \tau_{vm}] = (IP, model)$
21   **end**
22   HScaleUp($\delta_{scaled}, \kappa, \psi$)
23   **else**
24   HScaleDown(δ, ψ, κ)
25   **end**
26    $C \leftarrow H_{cdl}[t]$
27    $M \leftarrow H_{mld}[t]$
28    $E \leftarrow H_{exp}[t]$
29   **foreach** $c \in C$ **do**
30   DownloadContainer($c.IP, c.model$)
31   **end**
32   **foreach** $m \in M$ **do**
33   LoadModel($m.IP, m.model$)
34   **end**
35   **foreach** $e \in E$ **do**
36   ModelUnload($e.IP, e.model$)
37   TerminateVM($e.IP$)
38   **end**
39   LoadBalancerUpdate()
40   WakeUpAtNextTick()
41 **end**

Resource provisioner is implemented as a daemon process that is invoked at a fixed interval of time, as shown in Algorithm 2. On each invocation, the resource manager obtains a workload forecast t'_{setup} timesteps into the future [Line 4], where t'_{setup} is the accumulation of infrastructure provisioning time (t_{setup}) and forecasting time ($t_{forecast}$). Required number of VMs (i.e., α) can be calculated based on the forecasted workload as indicated in Algorithm 1. The heuristic presented in Section IV-D depends upon the SLO and cost per request; the best VM configuration will remain fixed as long as these two factors are unaltered. Thus, the best VM configuration index and the maximum number of requests served per VM are calculated once and stored in variables i^* and n_req^*, respectively [Lines 5-8]. After obtaining the required number of VMs α [Line 9], the difference between α and $|\kappa|$ is calculated to find the net number of VMs needed
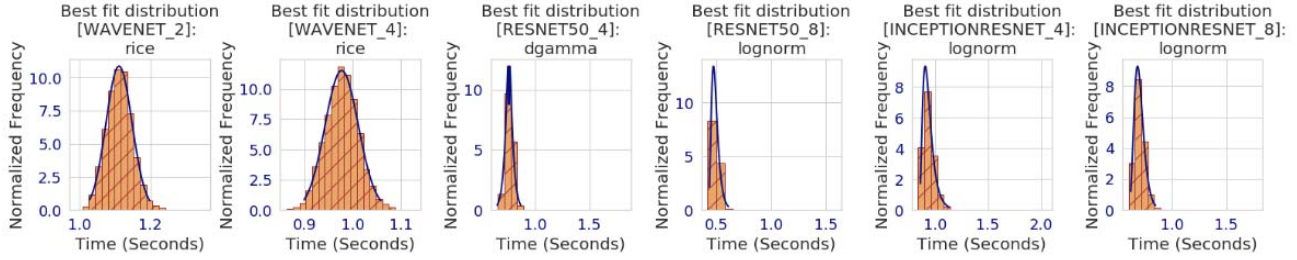
Fig. 6: Top ranked distribution that describes the variation in the sample data. The distribution (blue) is plotted on top of the histograms (orange) of observations.

at timestep $t + t'_{setup}$. For this difference, C^{κ}_{exp} (the number of VMs that will expire at time $t + t'_{setup}$) is subtracted from $|\kappa|$ to compensate for the VMs that will become unavailable due to lease expiration. The final difference value is referred to as δ [Lines 10-11].

A positive value of δ implies more VMs will be required at $t + t'_{setup}$. This requirement is fulfilled by: ① re-acquiring δ_{scaled} freed VMs, and ② spawning δ_{new} new VMs, such that $\delta = \delta_{scaled} + \delta_{new}$ [Lines 13-22]. A negative value of δ signifies the abundance of VMs at timestep $t + t'_{setup}$ which leads to freeing VMs [Lines 23-25]. Here, *HScaleUp* and *HScaleDown* functions scale the resources by re-acquiring δ_{scaled} VMs from κ to ψ and freeing δ VMs from κ and ψ, respectively. New VMs are spawned by scheduling container download, model loading and lease expiration events at timestamps $t + t_{vm}$, $t + t_{vm} + t_{cd}$ and $t + \tau_{vm}$, respectively [Lines 16-20]. Here, H_{cdl}, H_{mld} and H_{exp} are used to register events related to downloading containers, loading models and terminating VMs as key-value pairs, where the timestamps are the keys and the tuple $(IP, model)$ is the value. Apart from scheduling VMs for the future, resource provisioner also initiates the routines for downloading containers and loading models from the previous forecasts at the current timestamp t, followed by terminating VMs whose leases expire [Lines 29-38]. In the end, resource provisioner updates load balancer for newly deployed VMs, and sleeps till the next tick [Lines 39-40].

We vertically scale down the number of cores of a particular container if we meet the SLO with some threshold margin, and share the cores with the batch jobs. Vertical scaling is also helpful to allocate more resources for sudden workload surges. Moreover, if the resource estimator over-estimates the resources and whenever our prediction services can de-allocate cores while maintaining the SLO, Barista frees up cores, and if we miss any SLO, Barista will increase the number of cores immediately if more cores are available. We de-allocate one core at a time to minimize latency miss, and we double the number of cores (within maximum core limits of the VM) for the prediction service if there is any SLO miss.

V. EVALUATION

In this section, we present experimental validation for the different phases of our Barista framework.

A. Experiment Setup

Our testbed comprises an OpenStack Cloud Management system running on a cluster of AMD Opteron 2300 (Gen 3

Class Opteron) physical machines. We emulated the VM configurations as per the Amazon EC2 pricing model[3] [42]. We used the AWS pricing model to emulate our cost model[4]. We employed DockerSwarm [43] as our container management service on top of the VMs. HAProxy (http://www.haproxy.org) was used as our frontend and backend load balancers. We built a NodeJS-based frontend web application to relay the query to the backend predictive analytics service.

B. Predicting Execution Time of Predictive Analytics Services

In this experiment, we considered two different kinds of predictive analytics applications: image recognition and speech recognition, based on six different pre-trained deep learning models: Xception, VGG16, InceptionV3, Resnet50, InceptionResNetV2, and Wavenet (see Figure 1). All these models were profiled on OpenStack VMs of variable numbers of CPU cores. Based on the data generated from profiling these models, the best distribution was estimated from a list of available probability distributions. All the empirical distributions closely resemble the hypothesized distributions. For example, the best-fit distributions for the Wavenet service on two and four cores, for the Resnet50 service on four and eight cores, and for the InceptionResnetV2 service on four and eight cores are shown in Figure 6. Based on the best-fit distributions, we calculated the 95th percentile latency for each service.

C. Workload Forecasting

Two different time series datasets were used to emulate a realistic workload for predictive analytics services. The first dataset was collected and published by NYC Taxi and Limousine Commission [44]. We processed the data to extract the number of cab requests generated every minute based on the pick-up and drop-off dates and times [44]. This dataset provides an appropriate workload for a speech recognition component in the ride-sharing application to request a ride. The second dataset contains data on the number and types of vehicles that entered from each entry point on the toll section of the Thruway with their exit points [45]. This dataset can be used to represent a real-world workload of an image recognition based predictive analytics service that aims to automatically detect the license plate number of the entering or exiting car from a toll plaza. We processed the data to extract the total number of cars entering a toll plaza every minute.

---

[3] We considered 47 different VM configurations with 2, 4 and 8 cores. Any GPU-based or SSD-based configuration was not considered.

[4] We did not run the experiments on Amazon cloud because of monetary constraints.

30

A total of 10,000 data-points from each dataset was used in our study. We utilized 6000, 500 and 2500 data points for training, validation and testing the Prophet-based Forecaster model in both datasets, respectively. We performed *hyper-parameter tuning* for the Fourier series order N by iterating over five different values $\{10, 15, 20, 25, 30\}$, and with different sizes of the training window $W \in \{4000, 5000, 6000\}$. Out of the 15 possibilities, the configurations with $N = 30, W = 6000$ and $N = 20, W = 6000$ produced the least absolute percentage error (95th percentile) for the first and second datasets, respectively. The mean absolute error and absolute percentage error (95th percentile) for the first and second datasets are $(27.66, 29\%)$ and $(27.84, 30.26\%)$, respectively.

Fig. 9: Cumulative absolute percentage error distribution of first dataset.

Fig. 10: Cumulative absolute percentage error distribution of second dataset.

Prophet which often lags and leads. Figures 9 and 10 show the cumulative percentage error distributions for both approaches. Barista outperforms Prophet by 37% and 46% in the first and second datasets, respectively.

D. Resource Selection and Provision

Barista makes resource selection and provision decisions based on the algorithms described in Sections IV-D and IV-E. We evaluated our resource estimator and provisioner at different time points in the life cycle of the prediction services. We uniformly distributed the workload traces from one minute to five seconds in our experiment. We met the target SLOs (2 seconds and 1.5 seconds) 99% of the time over 12000 seconds for the Resnet and Wavenet services as shown in Figures 12a and 12b. The SLO (2 seconds) compliance rate marginally dropped to 97% for the Xception service as shown in Figure 12c.
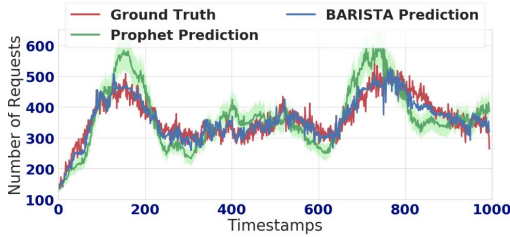

Fig. 7: Performance comparison of Barista (blue) and Prophet (green) along with ground truth (first dataset) (red).
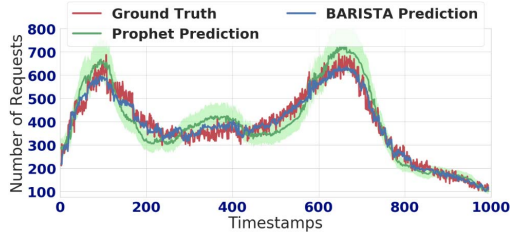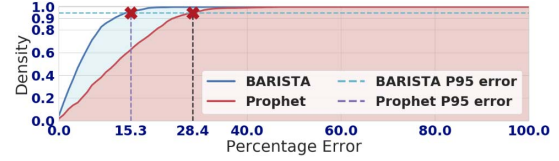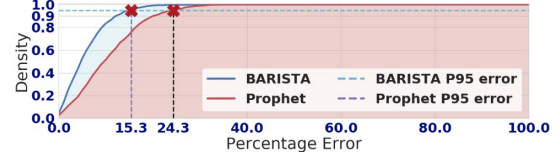

Fig. 8: Performance comparison of Barista (blue) and Prophet (green) along with ground truth (second dataset) (red).

In Barista, we extended Prophet with a machine learning based compensator to adjust the forecast based on the last five prediction errors. We decided to use five errors based on empirical results. Apart from these five prediction errors, the recent forecast of Prophet along with the upper and lower estimation bounds are used as features for learning the model (see Equation (5)). We used 3000 data points of Prophet to train the compensator model, and another 1000 data points to test and analyze our hybrid approach. We used H2O's implementation of AutoML framework [41] to identify the best family of learning algorithms and tune the hyper-parameters. We found that XGBoost-based gradient boosted trees outperformed other machine learning models such as neural networks, random forest model, and was selected as the best model. The mean absolute errors for training, cross-validation and testing in the first and second datasets are $(12.65, 15.10, 21.26)$ and $(12.24, 15.13, 22.65)$, respectively.

The forecasting results of Barista and Prophet along with the actual workloads on the test dataset are shown in Figures 7 and 8. It is visible from the figures that the Barista prediction curves closely resemble the actual workloads and it predicts the sudden bursts of requests with more accuracy compared to
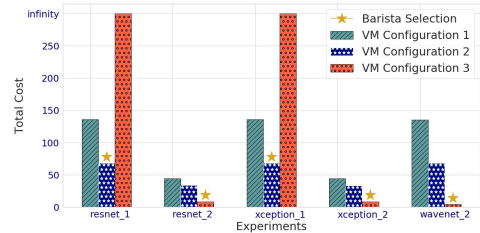

Fig. 11: Cost comparison between multiple VM configurations (cost infinity means the VM is infeasible option; it cannot serve the request within the SLO bound).

Barista not only guarantees SLO compliance but also minimizes the running and management cost by intelligently selecting the VM configuration. As described in Section IV-D, Barista selects the VM configuration depending on the SLO, the cost, and the estimated execution time of a prediction service. We emulated the prices of the VMs according to the Amazon EC2 instances and considered VM expiration time on an hourly basis (instance hour as an example scenario). Figure 11 shows the total cost of hosting the backend VMs for 10 hours (600 mins) while guaranteeing the SLO bound for the workload traces [44], [45]. Here, Configuration 1 represents t3.2xlarge, Configuration 2 represents t3.xlarge, and Configuration 3 represents t3.small (our min_mem constraint

31

(a) Upper image shows how we guaranteed the 2-second SLO for the Resnet prediction service and the experienced latency by using the VM configuration selected by Barista on toll dataset. Lower image shows the actual request rate, predicted request rate, and number of allocated VMs (t3.small (2cores)).

(b) Upper image shows how we guaranteed the 1.5-second SLO for the Wavenet prediction service and the experienced latency by using the VM configuration selected by Barista on taxi dataset. Lower image shows the actual request rate, predicted request rate, and number of allocated VMs (t3.small (2cores)).

(c) Upper image shows how we guaranteed the 2-second SLO for the Xception prediction service and the experienced latency by using the VM configuration selected by Barista on taxi dataset. Lower image shows the actual request rate, predicted request rate, and number of allocated VMs (t3.xlarge (4cores)).
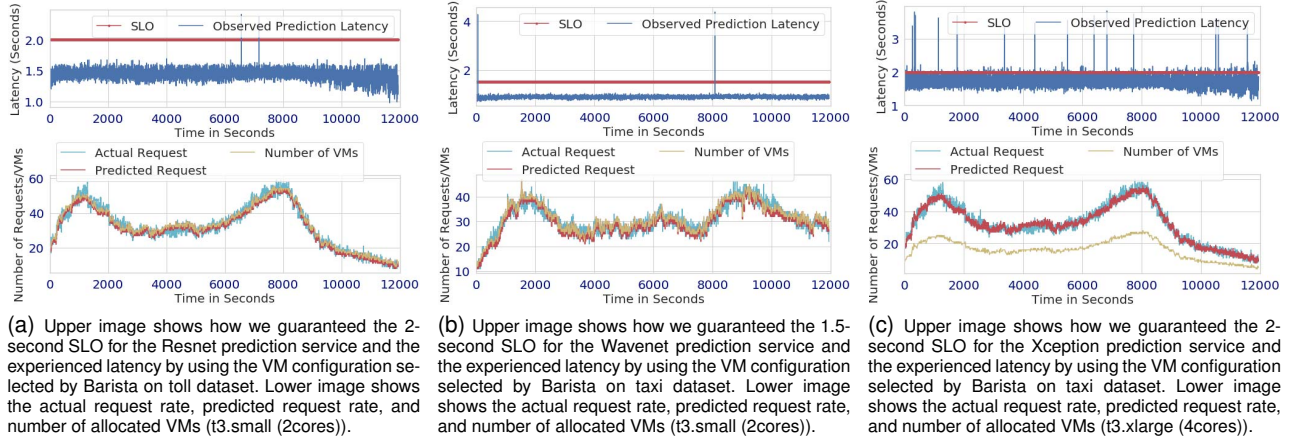
Fig. 12: Barista performance results on selected VM configuration as backend.

is 2GB). We solved the optimization problem for a given SLO bound to select a VM type, and considered this VM type as one of the configurations for our experiment. We then considered two other VM types of the same VM group (here, a group means Amazon EC2 instance type group, e.g., general purpose (t3, compute-intensive (c4-5 group)) with different core capacities. Even if assigning more cores reduces the running time of a prediction service, because of the cost difference of different VM configurations, the naive approach of selecting the VM with the highest number of cores or the most powerful VM is not always the best option, as more VMs from a less powerful type can be more cost effective. This happens because the prediction services are stateless and utilize all the cores available in that VM, and requests are always served sequentially. We observe from Figure 11 that Barista performs 50-95% better than the naive approach.

E. Reactive Vertical Scaling for Model Correction

We monitored the latency of the services every five seconds and make decisions accordingly based on the monitored latency and the SLO bound. We considered a deployment scenario where prediction services can run with other co-located low-priority batch jobs. In this experiment, the goal is to demonstrate that we can vertically allocate and de-allocate CPU cores by monitoring the SLO.
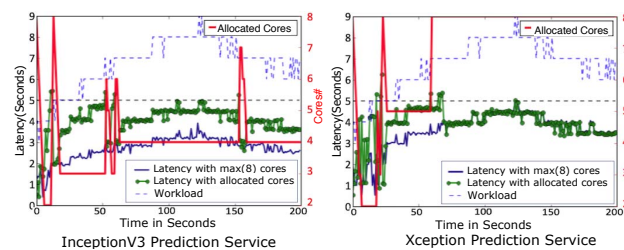

Fig. 13: Vertical Scaling to allocate the number of CPU cores(red line) while maintaining the SLO bound of 5 seconds. The blue dotted line shows the workload pattern, and the solid navy blue line shows the latency of the prediction services on a VM of 8 cores. The green line shows the latency if we dynamically (de)-allocate the cores.

Figure 13 shows how over-provisioning for two prediction services (Xception and InceptionV3) could be handled reactively by adjusting the CPU cores at runtime. Using our reactive approach for vertical scaling, we saved approximately 15% and 30% of CPU shares of an eight cores OpenStack VM for Xception and InceptionV3, respectively, on a particular workload trace. Our reactive approach also achieves over 98% of the SLO hits while optimizing the CPU shares significantly. We also observed similar behaviors for other prediction services, thus demonstrating the capabilities of Barista to make the model correction if there is any resource over-provisioning due to over-estimated time-series prediction.

VI. CONCLUSION

Summary: Predictive analytics services based on deep learning pre-trained models can be hosted using serverless computing paradigm due to their stateless nature. However, meeting their service level objectives (SLOs), i.e., bounded response times and bounded hosting costs, is a hard problem because workloads on these services can fluctuate, and the state of infrastructure can result in different performance characteristics. To resolve these challenges, this paper describes Barista, which is a dynamic resource management framework for providing horizontal and vertical autoscaling of containers based on predicted service workloads.

Discussions: The Barista approach can be broadly applied to other compute-intensive and parallelizable simulation services, where the simulation model needs to be loaded in memory, and the simulation behavior is determined based on user requests and user-specified SLO. In this paper, we did not consider running different co-located prediction services together on the same machine, where workload patterns for each prediction services can be different. Vertical scaling the different prediction services at the same time on the same machine is a dimension of our future work.

32

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[2] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-end attention-based large vocabulary speech recognition," in *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 2016, pp. 4945–4949.

[3] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 82–89.

[4] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[5] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 2017, pp. 109–120.

[6] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 613–627.

[7] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics." in *NSDI*, vol. 2, 2017, pp. 4–2.

[8] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[9] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics." in *NSDI*, 2016, pp. 363–378.

[10] S. K. Barker and P. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*. ACM, 2010, pp. 35–46.

[11] Y. Ukidave, X. Li, and D. Kaeli, "Mystic: Predictive scheduling for gpu based cloud servers using machine learning," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 353–362.

[12] E. Blasch, S. Ravela, and A. Aved, *Handbook of dynamic data driven applications systems*. Springer, 2018.

[13] Y. Barve, S. Shekhar, S. Khare, A. Bhattacharjee, and A. Gokhale, "Upsara: A model-driven approach for performance analysis of cloud-hosted applications," in *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2018, pp. 1–10.

[14] S. Shekhar, A. D. Chhokra, A. Bhattacharjee, G. Aupy, and A. Gokhale, "Indices: exploiting edge resources for performance-aware cloud-hosted services," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 2017, pp. 75–80.

[15] A. Bhattacharjee, Y. Barve, S. Khare, S. Bao, A. Gokhale, and T. Damiano, "Stratum: A serverless framework for the lifecycle management of machine learning-based data analytics tasks," in *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*, 2019, pp. 59–61.

[16] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 159–169.

[17] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 257–262.

[18] "AWS lambda," https://aws.amazon.com/serverless/, 2018.

[19] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Towards recovering the software architecture of microservice-based systems," in *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. IEEE, 2017, pp. 46–53.

[20] A. Bhattacharjee, Y. Barve, A. Gokhale, and T. Kuroda, "A model-driven approach to automate the deployment and management of cloud services," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 109–114.

[21] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "Opentosca–a runtime for tosca-based cloud applications," in *International Conference on Service-Oriented Computing*. Springer, 2013, pp. 692–695.

[22] R. Di Cosmo, A. Eiche, J. Mauro, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski, "Automatic deployment of services in the cloud with aeolus blender," in *Service-Oriented Computing*. Springer, 2015, pp. 397–411.

[23] A. Bhattacharjee, Y. Barve, A. Gokhale, and T. Kuroda, "(wip) cloud-camp: Automating the deployment and management of cloud services," in *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 2018, pp. 237–240.

[24] A. Brogi and S. Forti, "Qos-aware deployment of iot applications through the fog," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1185–1192, 2017.

[25] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth, "Towards faster response time models for vertical elasticity," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2014, pp. 560–565.

[26] E. Kalyvianaki, T. Charalambous, and S. Hand, "Adaptive resource provisioning for virtualized servers using kalman filters," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, no. 2, p. 10, 2014.

[27] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*. IEEE, 2017, pp. 472–479.

[28] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self-adaptive workload classification and forecasting for proactive resource provisioning," *Concurrency and computation: practice and experience*, vol. 26, no. 12, pp. 2053–2078, 2014.

[29] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "Dejavu: accelerating resource allocation in virtualized environments," in *ACM SIGARCH computer architecture news*, vol. 40, no. 1. ACM, 2012, pp. 423–436.

[30] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 607–618.

[31] A. Y. Nikravesh, S. A. Ajila, and C.-H. Lung, "An autonomic prediction suite for cloud resource provisioning," *Journal of Cloud Computing*, vol. 6, no. 1, p. 3, 2017.

[32] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, 2012.

[33] M. B. Sheikh, U. F. Minhas, O. Z. Khan, A. Aboulnaga, P. Poupart, and D. J. Taylor, "A bayesian approach to online performance modeling for database appliances using gaussian models," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 121–130.

[34] S. J. Taylor and B. Letham, "Forecasting at scale," *The American Statistician*, no. just-accepted, 2017.

[35] 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, no. 4, pp. 449–458, 2015.

[36] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[37] R. Wilcox, "Kolmogorov–smirnov test," *Encyclopedia of biostatistics*, 2005.

[38] H. G. Tucker, "A generalization of the glivenko-cantelli theorem," *Ann. Math. Statist.*, vol. 30, no. 3, pp. 828–830, 09 1959. [Online]. Available: https://doi.org/10.1214/aoms/1177706212

[39] T. Hastie and R. Tibshirani, "Generalized additive models: some applications," *Journal of the American Statistical Association*, vol. 82, no. 398, pp. 371–386, 1987.

[40] A. C. Harvey and N. Shephard, "Structural time series models," *Handbook of statistics*, vol. 11, no. 10, pp. 261–302, 1993.

[41] (2018) H2OAutoML. http://docs.h2o.ai/h2o/latest-stable/h2o-docs/_sources/automl.rst.txt.

[42] (2018) Amazon ec2 instances. https://aws.amazon.com/ec2/pricing/.

[43] (2018) Docker swarm. https://docs.docker.com/engine/swarm/.

[44] (2018) NYC taxi and limousine commission. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[45] (2018) NYC thruway. https://catalog.data.gov/dataset/nys-thruway-origin-and-destination-points-for-all-vehicles-15-minute-intervals-latest-full.

33