Razor: Scaling Backend Capacity for Mobile Applications

Yanjiao Chen, *Member, IEEE*, Long Lin, *Student Member, IEEE*, and Baochun Li, *Fellow, IEEE*

Abstract—The dramatic growth of mobile application usage has posed great pressure on application developers to better manage their backend capacity. Rule-based or schedule-based auto-scaling mechanisms have been proposed, but it is difficult or expensive to frequently adjust the backend capacity to track the burstiness of mobile traffic. In this paper, we explore a fundamentally different approach. Instead of scaling the backend in line with the mobile traffic, we smooth out traffic profiles to reduce the required backend capacity and increase its utilization. Our proposed solution, called *Razor*, is inspired by two key insights on mobile traffic. First, mobile traffic exhibits high short-term fluctuations but steady long-term trend, so that we may temporarily delay user requests and periodically adapt backend capacity based on the predicted traffic volume. Second, user requests have different priorities: while some requests are urgent (e.g., sending a message), some are delay-tolerant (e.g., changing the profile photo) and can be postponed without much influence on the user experience. Based on these observations, our design features a two-tier architecture: on a long timescale, Razor predicts future traffic using machine learning algorithms and plans the optimal backend capacity to minimize the budget with performance guarantee; on a short timescale, Razor schedules which requests to delay and by how much time to delay according to their delay tolerance. We implement a fully-functional prototype of Razor, and evaluate its performance with both real and synthetic traces. Extensive experimental results show that Razor can effectively help mobile application developers reduce their backend cost while guaranteeing the user experience.

Index Terms—Mobile application, backend management, dynamic request scheduling

1 Introduction

Consumers worldwide downloaded 149 billion mobile applications to their connected devices in 2016, and it is projected that 353 billion will be downloaded per year by 2021 [1]. For mobile application developers, there are mainly two concerns: frontend design and backend support. As shown in Fig. 1, users interact with the frontend, a user-friendly interface, and the operations will be sent via application programming interface (API) requests to the backend for processing.

Managing the backend is of significant importance to mobile application developers, whose objective is to maintain service quality at the lowest possible budget. Normally, the backend of mobile applications can be built on ready-made and customizable Infrastructure-as-a-Service (IaaS) cloud platforms (e.g., Amazon EC2 [2], Google Compute Engine [3]) or Mobile-backend-as-a-Service (MBaaS) cloud platforms. These cloud platforms provide on-demand capacities and adopt the pay-as-you-go model. For example, Amazon EC2 charges users for compute capacity by the hour or second. In addition, it takes several minutes to start or terminate an instance, making it difficult or impossible to adjust the backend capacity quickly, enough to track real-time fluctuations in user request demand [4]. All these factors make it challenging for mobile application developers to manage the backend in a cost-effective way.

Several solutions have been proposed to address the mismatch between fluctuated user traffic and capacity provisioning. These solutions can be broadly classified into two categories: dynamic resource adaptation and user traffic scheduling. However, existing approaches in the literature have some limitations. Most academic solutions related to dynamic resource adaptation fail to consider the latency in adjusting backend capacities. Solutions from the industry, on the other hand, are usually heuristic but inefficient. For example, Autoscaling in Amazon EC2 allows developers to define policies to specify when and how much to scale up or down the number of instances (e.g., add one instance when the CPU utilization is greater than 60 percent).

In contrast, an ideal autoscaling policy requires developers to have a much deeper understanding on the relationship between user traffic and the required number of instances, which is not available in most cases. Meanwhile, it is not always beneficial (or economical) for developers to frequently start or terminate instances, since the traffic peaks are usually short-lived, and the instances are charged on an hourly basis. Existing user traffic scheduling mechanisms are mostly designed for jobs that would last for several minutes with deadlines, but not mobile user requests that have large volumes but short processing times.

In this paper, we take a different perspective to tackle this problem. Instead of frequently altering backend configurations,

- *Y. Chen and L. Lin are with the School of Computer Science, Wuhan University, Wuhan, Hubei 430072, P.R. China. E-mail: {chenyanjiao, william_lin}@whu.edu.cn.*
- *B. Li is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada. E-mail: bli@ece.toronto.edu.*

Manuscript received 11 Mar. 2018; revised 1 Apr. 2019; accepted 11 Apr. 2019. Date of publication 17 Apr. 2019; date of current version 3 June 2020. (Corresponding author: Yanjiao Chen.)
Digital Object Identifier no. 10.1109/TMC.2019.2911935

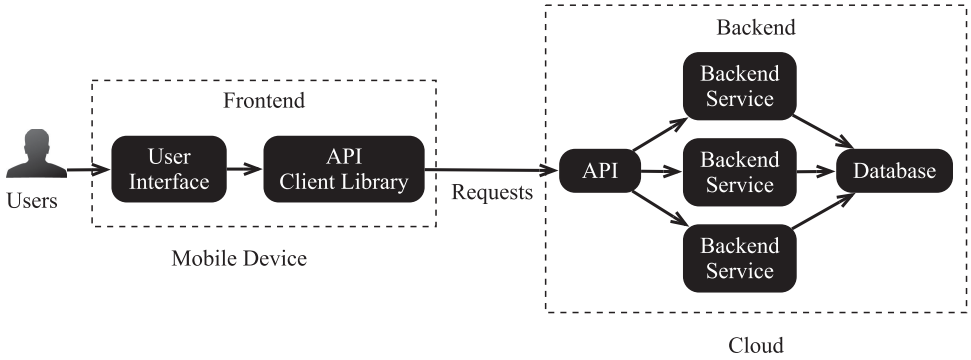

Fig. 1. Application development: Frontend and backend.

we choose to smooth out mobile traffic to reduce the required backend capacity and increase its utilization. We are inspired by two key insights on mobile traffic. *First*, the burstiness of mobile traffic is transitional, and the general trend can be predicted. The usage of mobile applications often follows diurnal patterns, and can be learned with historical data. Instantaneous traffic burstiness may be handled by delaying some user requests for a short period of time. *Second*, not all user requests are "created equal." For instance, in a messaging application, users expect timely processing on requests such as *sending a message*, but are less anxious about requests such as *changing the profile photo*. During traffic peaks, prioritizing urgent requests and postponing delay-tolerant requests may not necessarily result in a degraded user experience. These two observations motivate us to help developers make suitable decisions on backend capacity planning and user request scheduling.

In this paper, we present *Razor*, a new backend management mechanism that enables mobile application developers to optimize underlying infrastructures while maintaining service quality. Based on our key insights, Razor features a two-tier structure. On a long timescale, Razor periodically predicts future mobile traffic using machine learning algorithms and derives the optimal backend capacity for the developer to make the appropriate adjustment in the next time period. On a short timescale, Razor reduces the instantaneous peak traffic by deciding which user requests to delay, and by buffering these requests with their delay-tolerance restriction.

Highlights of our original contributions are as follows. *First*, with Razor, we are able to help mobile application developers address the dynamic backend capacity management problem with less cost. *Second*, our algorithm smooths out the peak load in backend, without compromising the quality-of-experience from the perspective of mobile users. *Finally*, Razor allows the backend to handle different delay-tolerance requests with rescheduling, which increases backend utilization.

The remainder of the paper is organized as follows. Section 2 describes the background of this work and presents an overview of Razor's design. Section 3.1 demonstrates our solution for optimal capacity planning. Section 3.2 compares four machine learning algorithms and Section 3.3 introduces a new request scheduling mechanism in the backend. Section 4 resolves implementation issues of Razor. Section 5 provides an extensive evaluation of Razor using three different traffic traces. Section 6 discusses the limitations of Razor. Section 7 reviews related work in the context of this paper. Section 8 finally summarize the paper.

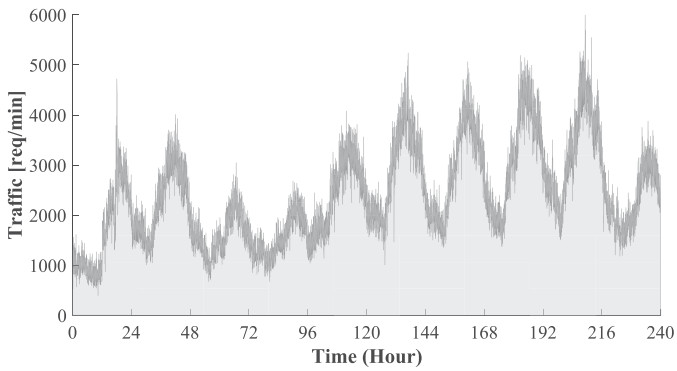

Fig. 2. Traffic fluctuations and diurnal pattern.

2 Background

2.1 Mobile Traffic

Mobile traffic features significant short-term variations and steady long-term trend. An analysis on real-world mobile traffic traces is presented in [5]. It is shown that the mobile data traffic exhibits burstiness over short timescale (e.g., 30 seconds) and demonstrated that to delay mobile traffic for a few seconds will not affect user experience. Similar observations are made in some other works that use unpublished private datasets. In Fig. 2, we plot the requests per-minute of the World Cup 1998 website access records [6]. It is clear that the traffic has a diurnal pattern but with instant wild fluctuations. The general trend enables us to predict future mobile traffic based on historical data, and the short-term burstiness can be smoothed through request delay.

2.2 Auto Scaling

Mainstream cloud service providers [7], [8] have offered rule-based (e.g., CPU utilization thresholds) or schedule-based (e.g., time-of-the-day) auto-scaling mechanisms that allow developers to scale the capacity up or down automatically according to predefined policies. Amazon Web Service (AWS) offers Auto Scaling, a rule-based auto-scaling mechanism that enables developers to use one or more performance metrics to define a scaling up or scaling down policy. The developer should specify the threshold, the time for the conditions to last to trigger the scaling, and the actions to take. These autoscaling mechanisms seem simple but are

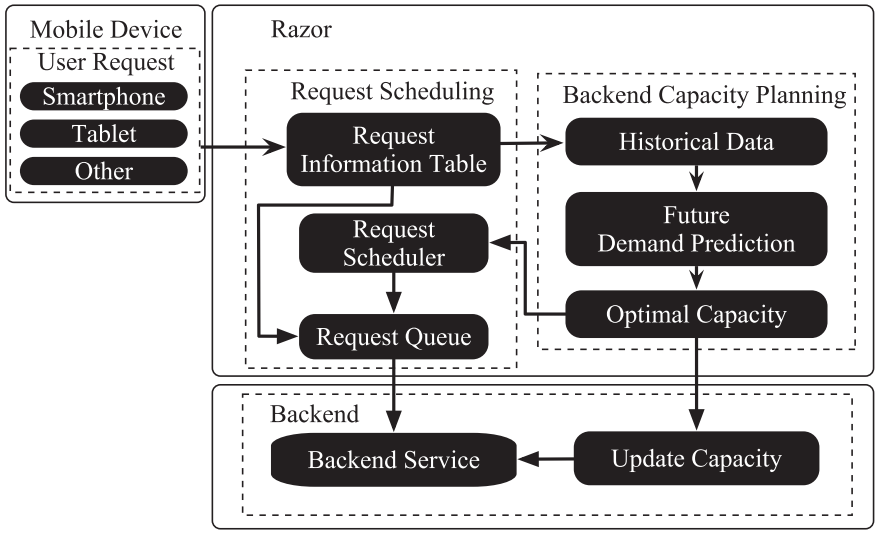

Fig. 3. Architecture of Razor.

inefficient in practice. To designate an ideal autoscaling policy, the developer needs to have a deep understanding of the relationship between the mobile traffic and the backend capacity, which is largely unavailable. For instance, the traffic fluctuations may lead to high or low CPU utilization. Nevertheless, it may not be necessary to scale up or down the capacity as the fluctuations may be momentary. Furthermore, even if capacity scaling is required, it is not sure how much capacity to scale up or down. Thus it is difficult for developers to take advantage of such autoscaling mechanisms for efficient backend management.

2.3 Startup and Shutdown Time

Another concern is the latency in backend capacity adjustment in the cloud environment. The time it takes to increase or decrease the number of instances is influenced by various factors such as the instance type, the OS image size, and the number of instances in concern. It is shown in [4] that the startup time of an instance can be as high as 15 minutes. Our experimental results is given in Fig. 4, showing that it takes more than one minute to launch or terminate an instance, during which the mobile traffic may have already changed. In Fig. 4, we can observe that the number of instances only slightly increases the launching or termination latency, while the type and family (i.e., t2 or c4) of instances has a more appreciable influence. Understanding the latency help Razor to plan ahead and make better resource provisioning decisions.

2.4 Pricing Model

Cloud platforms adopt a pay-as-you-go billing model, i.e., developers need to pay for their resource usage. For instance, Amazon EC2 offers on-demand instances, spot instances, reserved instances and dedicated hosts, each with a different pricing model [9]. Throughout this paper, we focus on the on-demand instance. Amazon on-demand EC2 instances usually charges users for compute capacity by the hour, namely each partial instance-hour consumed will be billed as a full hour or per-second. Additionally, users are charged from the time instances are launched until they are terminated or stopped. In other words, startup and shutdown time will also be charged. Therefore, it is critical for developers to make careful decisions on whether or not to scale up or down the backend capacity.

2.5 Architecture

Fig. 3 illustrates the two-tier architecture of Razor: long-term backend capacity planning and short-term request scheduling.

Backend capacity planning is conducted periodically on a relatively long timescale, e.g., per hour, since it is either difficult or costly to frequently change the backend capacity. The objective is sufficient capacity provision with reduced cost. The optimal backend capacity for the next time stage is calculated based on the predicted future mobile traffic and the delay-tolerance of each type of requests specified by the developer.

Request scheduling is performed on a short timescale, e.g., per minute, to smooth the short-term peaks by prioritizing delay-sensitive requests and buffering delay-tolerant requests based on the request dynamics. Such request scheduling aims at reducing the peak-to-average ratio of requests within the current time stage, but is not designed for persistent request congestion, which will be prevented through periodic backend capacity adjustment.

3 Razor: System Design

In this section, we first present the optimization model that minimizes the cost of backend under constraints of request delay. We then proceed to describe how future request demand can be predicted, and to evaluate the performance of different machine learning algorithms. Finally, we show how dynamic request scheduling can be performed.

3.1 Backend Capacity Optimization

We define the backend capacity as the maximum number of requests that can be served per minute, which is closely related to the cost of the backend (e.g., the number of instances required on Amazon EC2). We assume that backend capacity planning is conducted every hour, and the developer will be notified of the optimal backend capacity for the

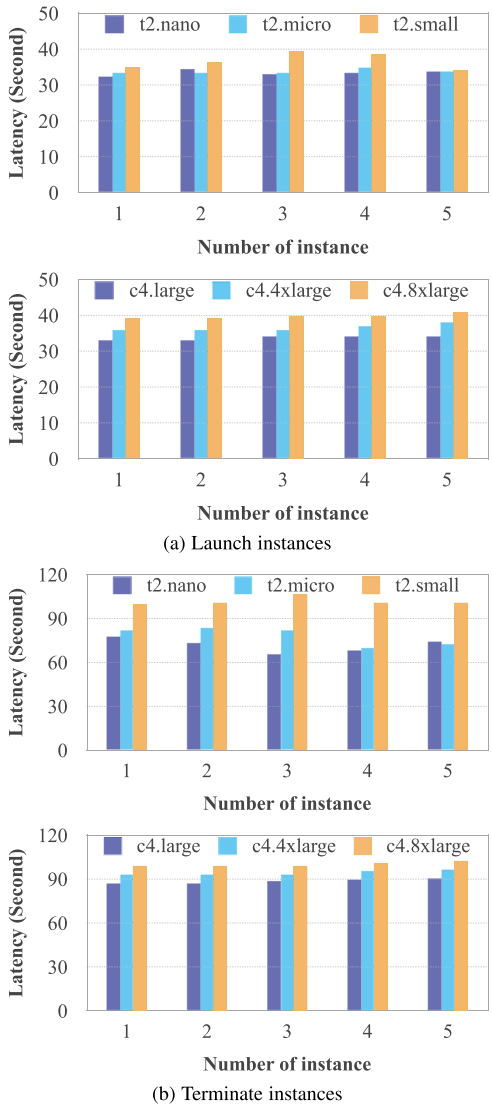

Fig. 4. Time for launching or terminating instances.

next hour with enough time to make the adjustment. The adjustment can also be automated through an API overlaying the existing autoscaling mechanisms in public cloud platforms.

Let N denote the required backend capacity, to be determined by the optimization model. The developer classifies all possible requests of the application into K types according to their delay tolerance, and feed the designated request types as input to Razor. An hour consists of $i = 1, 2, \ldots, 60$ minutes, during which the backend capacity is fixed. Let n_i^k denote the estimated number of type k requests that will arrive at the ith minute in the next hour. Without Razor, to guarantee the performance of the backend, the developer has to make sure that the backend capacity is greater than the peak demand, i.e., $N \geq \max_i \sum_k n_i^k$.

We make the simplified assumption that all requests generated in a specific hour will be served within that hour. In other words, during the current hour, the backend will neither handle requests from the previous hour, nor put off requests to the next hour. At the ith minute, let $\delta_{ij}^k \in [0, 1]$ denote the proportion of the n_i^k requests that will be postponed to the jth minute. All requests to be served at the jth minute, denoted by N_j, include those deferred from the previous minutes to the jth minute, and those generated and instantly served in the jth minute, i.e., $N_j = \sum_{i=1}^{j-1} \sum_{k=1}^{K} \delta_{ij}^k n_i^k + \sum_{k=1}^{K} \delta_{jj}^k n_j^k = \sum_{i=1}^{j} \sum_{k=1}^{K} \delta_{ij}^k n_i^k$. To guarantee the performance of the backend, its capacity should be larger than the peak demand after request scheduling, i.e., $N \geq \max_{j \in [1,60]} N_j$.

Razor classifies requests from the frontend according to the impact of their delay on the quality of service experienced by users. To illustrate this, we consider the following mainstream traffic classes with different delay tolerance (tolerant to urgent): (a) Streaming requests (e.g., video streaming). Since streaming data is often buffered beforehand, these requests are delay tolerant when their playback buffer is not empty. (b) Download requests (e.g., large file download). As large file downloading usually runs in the background, these requests can tolerate short delays since users usually expect to wait a little while for the download to complete. (c) Batch requests (e.g., batch database batch query). The results to the batch query can be returned in several parts with certain delay. (d) Large page requests (e.g., web browsing [5]). Due to their small size, mobile devices can only display a small portion of a website at any given time, thus off-screen contents can be downloaded a little bit later without impacting the user experience. (e) Customized requests (e.g., changing account information). The app developer can specify delay tolerance for customized requests based on use case.

Delaying requests for a long time will affect the user experience, thus the developer needs to have a control over how many and how long a certain type of requests can be delayed. Razor allows the developer to set the upper-bound of δ_{ij}^k as $\overline{\delta_{ij}^k}$, which depends on the request type k and the length of delay $j - i$. For instance, in a web browsing application, mobile devices like smartphones and tablets can display only a small portion of the whole webpage at a given time due to their small screen sizes. Requests for contents that will be shown on the screen are the most urgent type of requests and should not be delayed, so that the developer can simply set $\forall j > i, \overline{\delta_{ij}^{\text{download showing content}}} = 0$; if no request should be delayed for more than 2 minutes, the developer can simply stipulate that $\forall k, \overline{\delta_{ij}^k} = 0, \text{if } j - i > 2$.

As the backend cost will monotonically increase with the backend capacity, we set the objective of Razor as to minimize the required backend capacity, without violating the constraint on δ_{ij}^k designated by the developer

$$\min_{\delta_{ij}^k} \ N, \tag{1}$$

$$\text{subject to } N \geq \max_{j \in [1,60]} N_j, \tag{2}$$

$$N_j = \sum_{i=1}^{j} \sum_{k=1}^{K} \delta_{ij}^k n_i^k, \forall j, \tag{3}$$

$$\sum_{j=i}^{60} \delta_{ij}^k = 1, \forall i, \forall k, \tag{4}$$

$$0 \leq \delta_{ij}^k \leq \overline{\delta_{ij}^k}, \forall i, \forall j, \forall k. \tag{5}$$

Constraint (2) guarantees that the backend capacity is greater than the smoothed peak demand. Constraint (4) ensures that all the requests initiated in the ith minute are

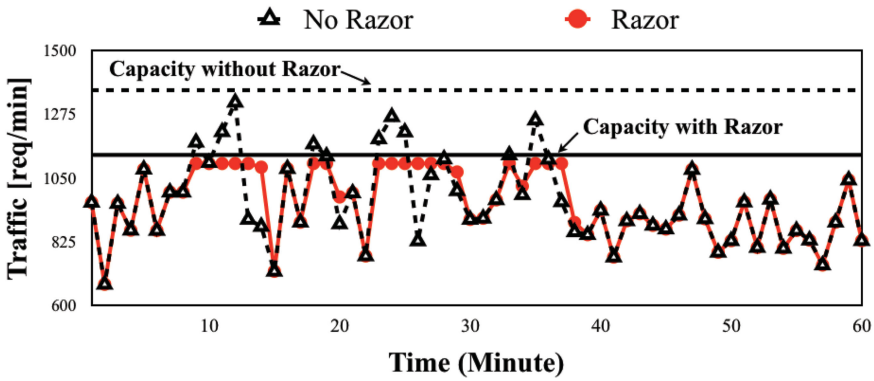

Fig. 5. Request demand smoothing by Razor.

served, either instantly or in later minutes. The objective function is minimized through variables δ_{ij}^k, and we can get the optimal backend capacity as $N^* = \max_{j\in[0,60]} N_j^*$, in which $N_j^* = \sum_{i=1}^{j}\sum_{k=1}^{K} \delta_{ij}^{k\,*} n_i^k$. The optimization problem (1) is a linear programming problem, and can be solved by existing algorithms such as Simplex and Interior point algorithms. Fig. 5 gives an example of how the backend capacity optimization of Razor works with the FIFA trace. It is shown that the demand profile is smoothed and the required backend capacity is lower.

3.2 Future Demand Prediction

The expected number of requests generated at the ith minute in the next hour is needed as input for the backend capacity optimization. Network traffic prediction is a well-studied problem, with many proposals for better and more expressive prediction algorithms, which can be classified into two categories: linear methods and nonlinear methods. The most widely adopted linear prediction methods are based on ARMA/ARIMA [10], [11], which heavily rely on the mean values of historical series data. With the recent rise of machine learning, there has been increased interest in nonlinear adaptive prediction methods based on machine learning algorithms [12], [13], which are able to capture the rapid variations underlying the traffic load and outperform linear methods. For instance, Neural Networks (NN) are widely used for modeling and predicting network traffic since they can learn complex non-linear patterns and unusual traffic patterns thanks to their strong self-learning and self-adaptive capabilities.

In this paper, we leverage machine learning algorithms to predict future demand based on historical data. There is a wide variety of machine learning algorithms for data prediction, each of which has their advantages and disadvantages. We focus on four widely-used algorithms: linear regression (LR), single-hidden-layer multilayer perceptron (sMLP), deep belief networks (DBN), and convolutional neural networks (CNN). Linear regression and sMLP are simple machine learning models, while DBN and CNN are deep learning models. In addition, we discuss the compromise between the desired prediction accuracy and training time of the four machine learning algorithms.

Historical data are collected in the form of the number of requests generated in each minute. The input vector x to the machine learning model should contain the best predictors for n_i^k. Two potential factors should be taken into consideration. One is *temporal-proximity*, i.e., the most recent demand indicates the near future. The other is *diurnal effect*, i.e., demands at the same time of each day have a similar trend, as shown in Fig. 2. Therefore, we use the demand of the previous two hours and the demand around the same time of the previous two days as the input vector to predict n_i^k, i.e., $x = (n_{i-180}^k, \ldots, n_{i-61}^k, n_{i-1440-180}^k, \ldots, n_{i-1440+179}^k, n_{i-1440*2-180}^k, \ldots n_{i-1440*2+179}^k)$, and $Y = n_i^k$. Furthermore, we normalize entries in the input vectors as $x \rightarrow x/\max\{n_i^k\}$.

In this paper, we use the World Cup 1998 trace during April 30, 1998 $\sim$ July 26, 1998 to evaluate the performance of the four machine learning models. The World Cup trace consists of access logs and we count the number of request generated in each minute as one sample. Thus, there are 80,000 samples for training, 20,000 samples for validation, and 20,000 samples for testing. We run the four machine learning algorithms on a desktop computer with a 3.6 GHz Inter Core i7-4790 CPU and 8 GB memory. As shown in Fig. 6a, deep learning models outperform simple models in terms of accuracy, since deep learning models are more powerful in discovering the intricate relationship between the input and the output. As shown in Fig. 6b, the training time of deep learning models is far higher than that of simple models, which is not surprising since the deep learning models contain far more parameters to be learned. The training time of DBN and CNN also depends on the choice of the number of layers and neurons in each layer. Though the training time diverges considerably, given a new input, it takes almost the same time (less than a second) for the trained models of all four algorithms to yield the prediction result. Therefore, it is possible for Razor to re-train the deep learning models on a daily or weekly basis, and use the trained model for online prediction.

3.3 Dynamic Request Scheduling

After the optimal backend capacity has been calculated according to Equation (1), the developer will adjust the instance configuration on Amazon EC2 accordingly, and such backend capacity will be fixed for the next hour. Given the backend capacity for the hour, to ensure that the backend will not be overwhelmed by the instantaneous request demand, we design and implement a dynamic request scheduling strategy.

Razor schedules requests from users based on the backend capacity and delay tolerance of these requests. The requests of a more delay-sensitive type get a higher priority and will be scheduled earlier. The most urgent requests will

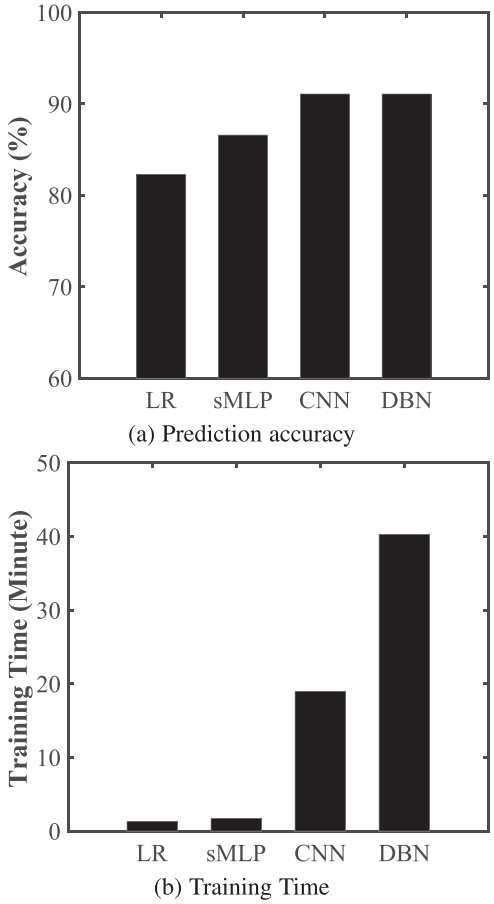

Fig. 6. Prediction accuracy and training time.

be handled instantly without delay, and the most delay-tolerant requests will be postponed for a relatively longer period of time. For requests with the same delay tolerance, those with an earlier arrival time will get a higher priority.

As shown in Fig. 7, at each minute t, the requests in the system are in one of the four states: new arrivals, pending, processing and finished. At the beginning of minute t, there are $R(t)$ requests being processed, $F(t)$ new arrivals and $P(t)$ pending requests. First of all, the most urgent type of requests should not be delayed, thus the backend will process these request immediately. In addition, an initial priority will be assigned to each request in the pending status according to their delay tolerance. A delay-sensitive request will have a higher initial priority, and a delay-tolerant request will have a lower initial priority. The priority of pending requests will increase along with time. During minute t, $N - R(t)$ requests with the highest priority (either pending or new arrival requests) will be launched into the processing status. The new arrivals with lower priority are then pended, and some pending requests remain pended until the next time slot. During minute t, a proportion θ of the requests in the processing state will be finished, while the remaining $(1-\theta)R(t)$ requests will still be under processing.

A major problem facing dynamic request scheduling on the backend is that the new backend capacity calculated based on the optimization problem (1) may not be able to accommodate the newly arrived requests during the next hour. More specifically, future request demand cannot be precisely predicted, and it is possible that the real traffic is higher than expected. To mitigate this problem, we design a threshold-based request scheduling mechanism as summarized in Algorithm 1.

Algorithm 1. Threshold-Based Dynamic Request Scheduling Algorithm

Input: Newly-arrived requests r_i, capacity for each minute N, number of time slots T, threshold $\widetilde{n} \leq N/T$.

```
Calculate the available capacity for each time slot as c̃ = N/T.
if Request r_i is most urgent then
    Process request r_i instantly.
    Update the threshold ñ = ñ − 1.
    Update the available capacity c̃ = c̃ − 1.
end if
if Request r_i is delay-tolerant then
    if Pending queue is empty and ñ is non-zero then
        Process request r_i instantly.
        Update the threshold ñ = ñ − 1.
        Update the available capacity c̃ = c̃ − 1.
    else
        Put request r_i into the pending queue.
    end if
end if
Rank requests in the pending queue in a non-ascending order of priorities.
if c̃ is non-zero then
    Process the first c̃ requests in the pending queue.
end if
```

We divide each minute into T smaller time slots, say 5 seconds, so the capacity for each time slot will be N/T. The granularity of the time slot should be small, since newly-arrived requests that are delay-tolerant will be buffered in a queue to be scheduled based on their priority, which means

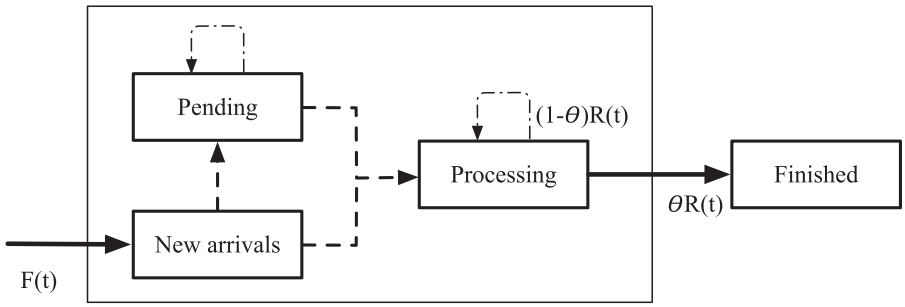

Fig. 7. Dynamic request scheduling.

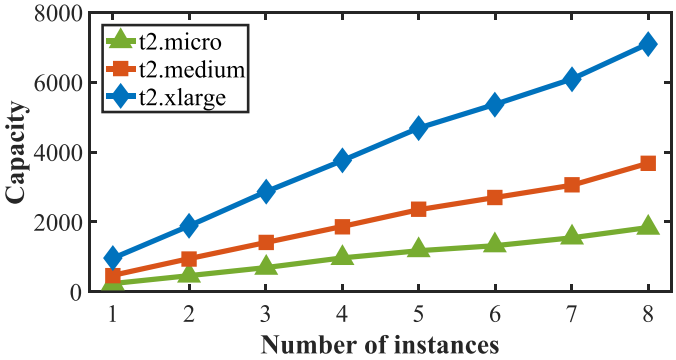

Fig. 8. Relationship between the backend capacity and the number of needed instances.

that these requests will be delayed for at least one time slot. The most urgent newly-arrived requests will be processed instantly in any circumstances to avoid user experience degradation. We set a threshold $\widetilde{n} \leq N/T$ for dynamic request scheduling. If the pending queue is empty and there are fewer than $\widetilde{n}$ requests being processed in the current time slot, other newly-arrived requests will be processed instantly; otherwise, these newly-arrived requests will be put into the pending queue, ranked in a non-ascending order of priorities that are jointly determined by the delay tolerance and arrival time. Requests in the pending queue will be scheduled later according to their priorities.

4 Implementation

In this section, we first discuss issues of backend capacity implementation on public cloud platforms, and then address the problem of implementing dynamic request scheduling.

4.1 Backend Capacity Optimization

Backend Capacity Realization. In Section 3.1, we have derived the optimal backend capacity in terms of the maximum number of requests that can be served in unit time. We need to identify the number of instances needed in Amazon EC2 to realize such an optimal backend capacity. In this paper, we use t2 family of instances because it is commonly used in almost all existing general-purpose workloads, e.g., websites and mobile applications. There is a wide selection of instance types provided by AWS, and developer can select an optimal instance type for their applications and exploit the relationship between the backend capacity and the number of needed instances as follows. We adopt the autoscaling policy of AWS and set the CPU utilization threshold as 50 percent, i.e., we fix the number of instances as α and gradually increase the number of requests to see when the CPU utilization will hit 50 percent. In this way, we can get the maximum number of requests $N(\alpha)$ that can be served by α instances. In turn, it is indicated that to realize a backend capacity of $N(\alpha)$, a number of α instances are needed. Through experiments, we can obtain the relationship between the backend capacity and the number of needed instances as shown in Fig. 8. It is shown that the backend capacity is approximately a linear function of the number of instances, and the slope of the linear function depends on the type of instances.

Backend Capacity Adjustments. If the optimal backend capacity for the next hour turns out to be the same as that of the current hour, the developer does not have to make any changes. If the optimal backend capacity for the next hour is higher or lower than that of the current hour, the developer can increase or decrease the number of instances according to the relationship between the backend capacity and the number of needed instances as discussed above. More specifically, when scaling down backend capacity, the developer should choose which instances to shut down. Razor closely monitors the operation time of each instance and selects the instances that are approaching full hour as potential ones to terminate. In our implementation, we set one hour as the cycle for backend capacity optimization and adjustment. Considering the launching and termination latency as shown in Fig. 4 and the time to solve the optimization problem, we perform backend capacity optimization 10 minutes before the upcoming hour. We dynamically scale up or down the number of instances in the cloud by changing the configuration file base on Section 3.1. We also record historical request processing time and request information as the basis to improve the system performance.

4.2 Request Scheduling Cost

In our request scheduling mechanism, we use a queue to temporarily buffer delay-tolerant requests, which will lead to storage overhead. We analyze this storage overhead under two different circumstances: queue in memory and queue in disk.

In order to make full use of the CPU and memory, we keep all requests in the memory (newly arrived, pending, and in-processing requests) in our implementation. In this way, we can reduce latency as much as possible, and such latency is also evaluated through experiments in Section 5. In our implementation, we find that the memory of the instances deployed according to our optimal backend capacity planning is enough for queueing the requests during dynamic request scheduling. Therefore, there is no extra cost since the instances have already been paid for. In the case that a huge number of requests have arrived such that a large space is needed for request buffering, we may need to save some of the requests in disk using serialization. This might cause additional cost but it is much cheaper than the payment for instances in the cloud.

5 Performance Evaluation

In this section, we first develop a testbed of Razor on Amazon EC2 on-demand instances, then compare Razor with two benchmarks in terms of the backend cost and utilization, using both real and synthetic datasets. We also evaluate Razor under different parameter settings.

5.1 Testbed

The testbed of Razor is developed using the t2.micro on-demand instance in the US west (Oregon) in Amazon EC2 with 1 vCPU, 1 GiB memory [9]. To simulate backend, we created an Amazon Machine Image (AMI), where a shell script is added to /etc/rc.laocal so that Razor can keep running once the instance is launched using this AMI. To record the real delay of every request, we created a mysql database in Amazon Relational Database Service (RDS), where we can note down every request log from each instance in service.

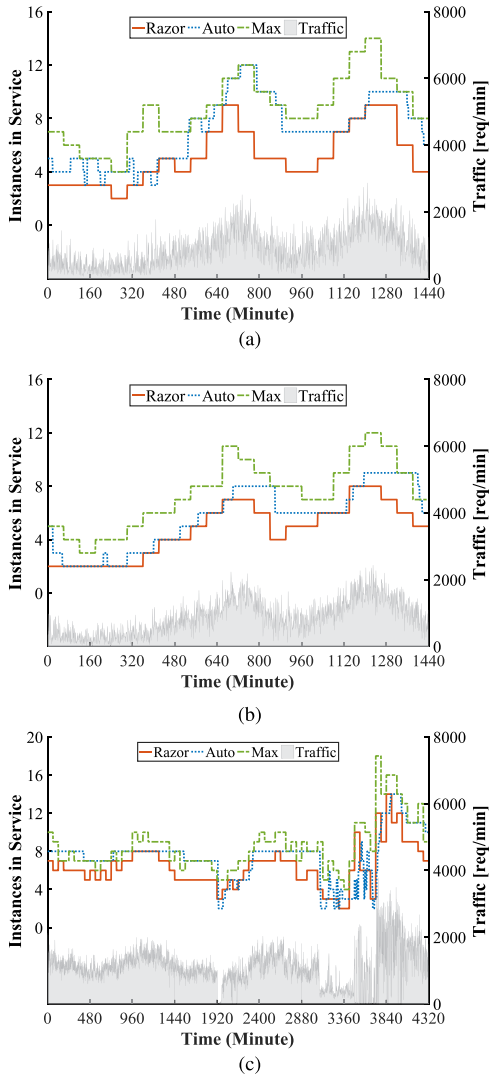

Fig. 9. Comparison of Razor, Autoscaling, and Maxscaling when the traffic is drawn from three datasets. The graph shows the traffic and the number of instances in service. (a) Poisson dataset. (b) Uniform dataset. (c) The World Cup 1998 trace.

We use the collected data trace as input to the optimization problem in Equation (1). We use bills from Amazon to show the real cost. Since Amazon EC2 does not break the bills into individual requests, we do not report the cost of each request but the overall cost.

We have developed a series of test generator, each of which can send emulated traffic that consists of different types of requests, such as resource request, database requests and so on. We divide them into $K = 5$ types: type 1 requests are the most urgent and type 5 requests are the most delay-tolerant. No request should be delayed for more than 2 minutes, and maximum delayed time of type $1 - 5$ requests is 10, 30, 60, 90 and 120 seconds, respectively.

We choose two benchmarks to compare with Razor: peak scaling (Maxscaling) and Autoscaling in AWS [7]. In Maxscaling, we allocate enough EC2 instances to scale the backend capacity to meet the (predicted) peak demand in the upcoming hour and the number of instances are updated hourly. For Autoscaling in AWS, we choose the rule-based autoscaling mechanism, and the number of instances can change at any time once the autoscaling policy is triggered. Autoscaling policies in AWS are set as follow: 1) scale-down policy: remove 1 instance when the average CPU utilization is between 10 and 20 percent for 600 seconds, remove 2 instance when the average CPU utilization is between 0 and 10 percent for 600 seconds; 2) scale-up policy: add 1 instance when the average CPU utilization is between 50 and 60 percent for 300 seconds, add 2 instances when the average CPU utilization is between 60 and 80 percent for 300 seconds, add 3 instances when the average CPU utilization is between 80 and 100 percent for 300 seconds. To maintain a reasonable user experience, we target at a CPU utilization of no more than 50 percent. In other words, we keep each instance running at no more than 50 percent capacity. Note that there are potentially infinite number of ways to set the autoscaling policies in AWS. We have experimented with numerous different policies, and choose the one with the best performance for comparison.

5.2 Dataset

Since the patterns of mobile traffic are infinite, we adopt three different datasets for a fair comparison, including one real and two synthetic dataset for evaluation.

Real Dataset. Web browsing applications are one of the top generators of mobile traffic [5]. Unfortunately, to the best of our knowledge, there is no public traces of mobile web application traffic. To emulate the real-world traffic, we use the World Cup trace to evaluate Razor, as mentioned in Section 3.2. This trace has been extensively used in the literature. It contains all HTTP requests made to the World Cup website in 92 days. During this period, the website has received 1,352,804,107 requests. In our future demand prediction, we need to calculate the aggregate number of requests per minute from these traces for training and prediction.

Synthetic Dataset. As poisson and uniform distribution are often used to model distributions of users for network services [14], [15], we generate a poisson and a uniform distribution dataset as follows. Assume that there are 100 users, each generating requests according to a poisson or a uniform distribution. For each type of requests, we first generate a series of values with a diurnal pattern (low demand at working time, and high demand at leisure time), to represent the request arrival rate at different time of a day for an average user. Then, we compute the arrival rate for each individual user. To simulate the burstiness over a short timescale, we add noises to the arrival rate of the average user. Finally, aggregating the number of requests from all users at each minute yields the request demand dataset.

5.3 Simulation Result

Costs. Fig. 9 shows how the backend capacity changes as the traffic changes. It is obvious that the number of instances increases or decreases whenever the traffic goes up or down. Compared with Maxscaling, Razor always allocates fewer instances. In order to match the request demand, Autoscaling changes the backend capacity more frequently than the other two mechanisms when request demand fluctuates dramatically. But such actions always fall behind the traffic changes because of the rapidly-changing request demand and the latency in launching or terminating instances. Razor periodically (per hour) plans the optimal backend capacity in advance and the resulting backend capacity closely matches

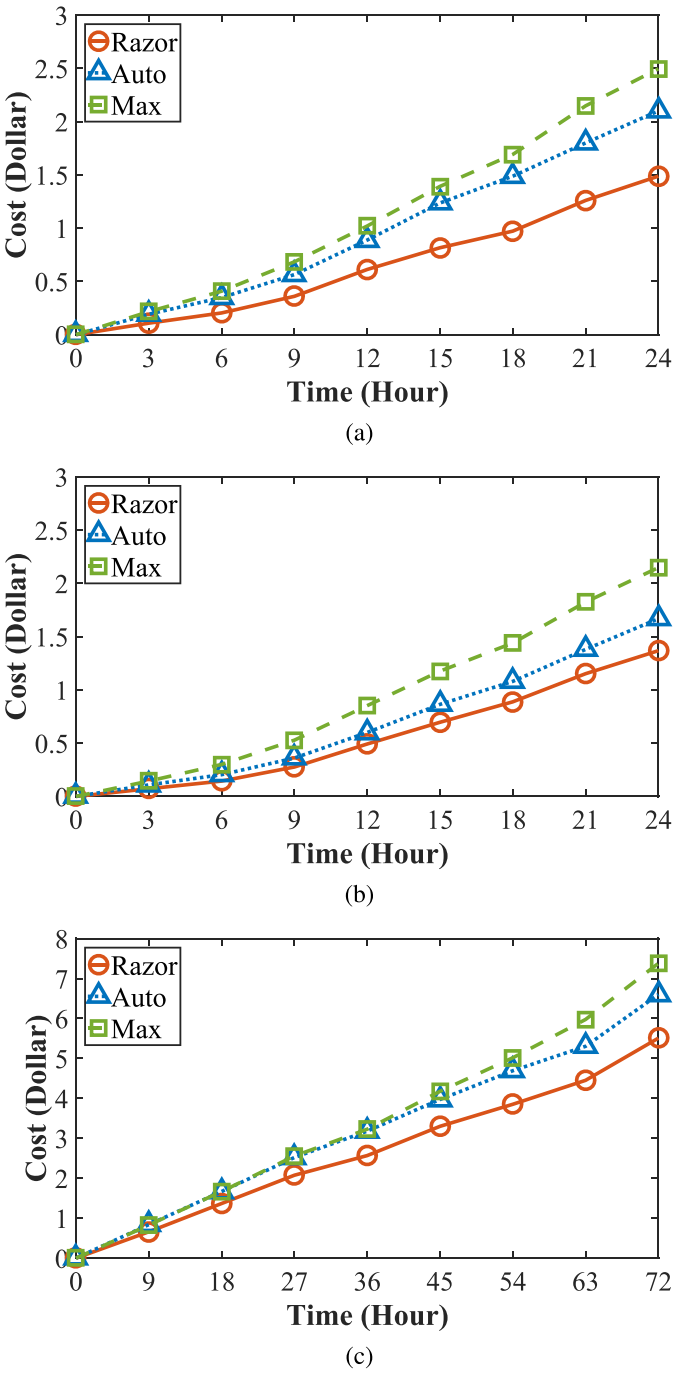

Fig. 10. Accumulated cost of Razor, Autoscaling and Maxscaling for three datasets. (a) Poisson dataset. (b) Uniform dataset. (c) The World Cup 1998 trace.

the incoming traffic with the least required instances. We can observe that Razor may deploy a larger number of instances (at the start of the hour) than AWS autoscaling if Razor predicts that the demand in the next hour is high. But on average, Razor will deploy a lower number of instances than AWS autoscaling since AWS autoscaling may respond to a burst of traffic by adding too many instances while Razor changes the configuration less frequently and more properly by carefully predicting future traffic demand.

In this experiment, we measured the average costs per hour and accumulated costs by different scaling schemes in the cloud. Fig. 11 shows that when the request demand

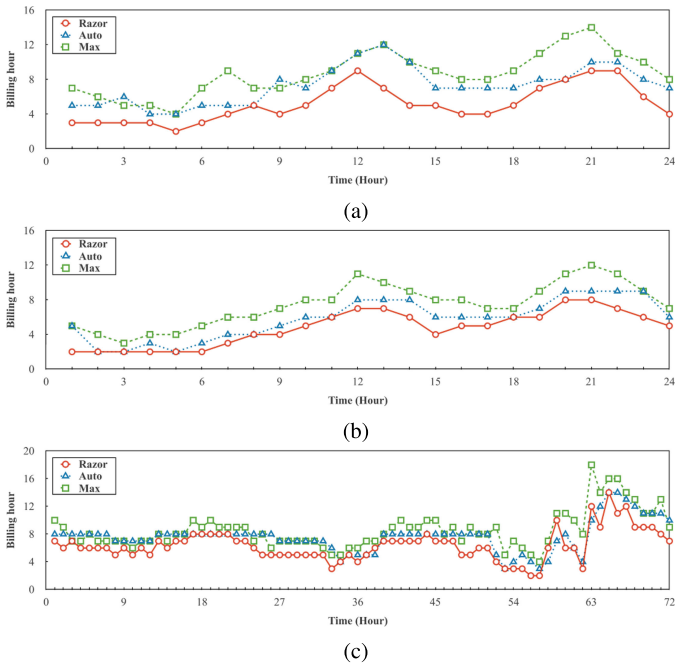

Fig. 11. Billing hours of Razor, Autoscaling and Maxscaling for three datasets. (a) Poisson dataset. (b) Uniform dataset. (c) The World Cup 1998 trace.

follows the Poisson dataset and Uniform dataset, Razor always uses fewer billing hours than Autoscaling and Maxscaling. This result confirms that Razor can make proper decisions for backend planning. Fig. 10 shows the accumulated costs for the three scaling mechanisms. All the experimental results on three datasets show a similar trend in accumulated costs. When the request demand follows the World Cup trace, Razor incurs 16.5 percent less cost than Autoscaling and saves 25.4 percent cost compared to Maxscaling. Overall, Razor outperforms the other two baselines and achieves the least cost. The benefits of Razor stem from optimally planning backend capacity.

Backend Utilization. Razor performs capacity adjustment according prediction to minimize the incurred cost. This, in turn, improves the average backend utilization. Fig. 12 shows the backend utilization achieved by the three scaling mechanisms, computed on an hourly basis. Processed requests are calculated as utilized capacity while others are not. Autoscaling and Maxscaling have lower backend utilizations as they request developers to set more instances to handle user requests. Most of the time, Razor has the highest backend utilization, compared with Autoscaling and Maxscaling. In our experiments, the backend utilization can be improved by as much as 54.9 percent with Razor.

Prediction Accuracy. Core elasticity regarding under-provisioning and over-provisioning is an important aspect to evaluate the effectiveness of resource provisioning mechanisms [16], [17], [18]. We can adopt the delay tolerance non-miss rate or CPU utilization to differentiate under-provisioning and over-provisioning, e.g., we deem the backend deployment as over-provisioning if the CPU utilization is higher than a threshold. We show the prediction accuracy in terms of over-provisioning and under-provisioning in Fig. 13. It is shown that the resource supply of Razor can closely track the resource demand. Since Razor aims to

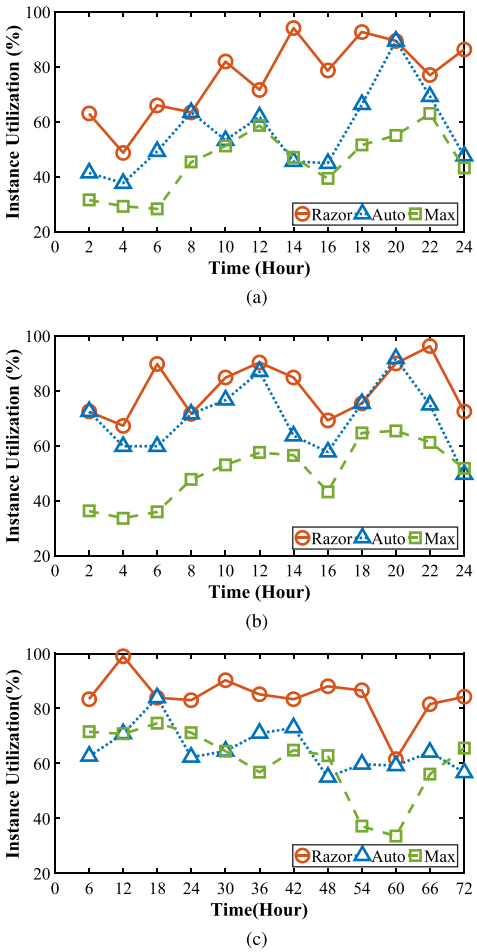

Fig. 12. Backend utilization of Razor, Autoscaling and Maxscaling for three datasets. (a) Poisson dataset. (b) Uniform dataset. (c) The World Cup 1998 trace.

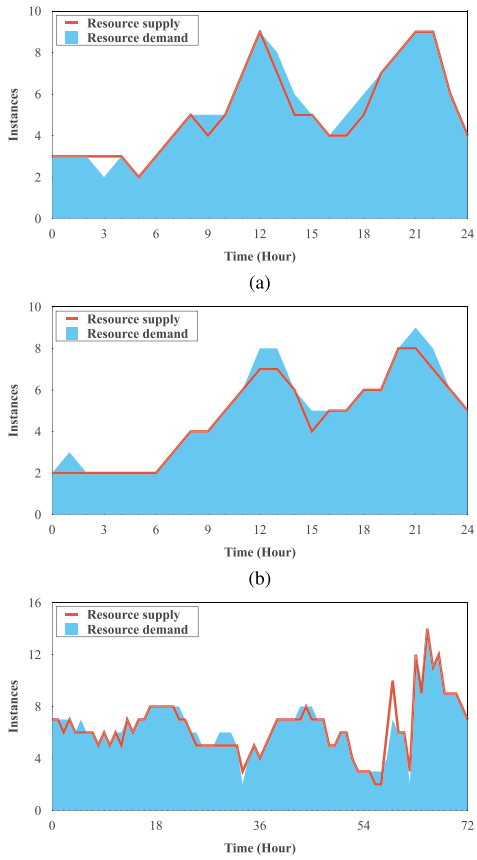

Fig. 13. Prediction accuracy of Razor for three datasets. (a) Poisson dataset. (b) Uniform dataset. (c) The World Cup 1998 trace.

reduce the backend cost for developers, the instance deployment tends to incur more under-provisioning than over-provisioning. Developers who are less cost-sensitive and want to cater to user demand can adjust the objective function in (1) to achieve a balance of under-provisioning and over-provisioning, e.g., set the instance deployment higher than N.

User Experience. Razor delays requests to smooth the traffic profile and reduce the cost, so we check the reschedule rate and delay tolerance non-miss rate to evaluate the influence of Razor on user experience. It is shown in Fig. 14 that Razor provides performance guarantees as it plans the backend capacity to meet estimated future demand based on historical information. As shown in Fig. 14a, in Razor, only 12.9 percent of type 5 requests, the most delay-tolerant type of requests, are rescheduled, while far less than 10 percent other types of requests are rescheduled. For all requests, more than 90 percent are finished within the delay tolerance, which indicates that Razor has little impact on user experience.

Delay Constraints. We vary the parameters of delay constraints in Razor to evaluate the corresponding backend cost. Fig. 15 shows accumulated billing hours under different delay constraints. If Razor can delay requests for a longer time, meaning that the constraint on request delay is looser, a lower backend capacity is needed and fewer instances are required, e.g., a delay constraint of 10 minutes induces the lower cost. The reduction of cost is high when the delay constraint is tight, but will be insignificant when the delay constraint is loose enough, e.g., the costs under delay constraints of 5 minutes and 10 minutes are almost the same.

6 Discussion

In this section, we discuss some of the issues and limitations of Razor.

6.1 Historic Data Collection

Razor relies on historic traces to learn the trend of request demand in order to suggest an appropriate backend capacity to application developers. It is essential to collect enough training data samples for building a predictive model. However, for newly deployed mobile application backend, there is a lack of historic data. We may resort to the observation in cloud computing that recurring jobs have similar trend [19], thus recurring user requests may reflect similar user usage behaviors. Therefore, we may extend the limited data using data augmentation techniques [20]. Note that the trend of mobile application requests may change over time. Therefore, we should only use data augmentation techniques at the early deployment stage when there is scarce historic data, and retrain the predictive model periodically to catch the newest trend of request demand.

6.2 Backend Performance Variation

One of the concerns with cloud-based backend is that there could be performance variations over time even if the same

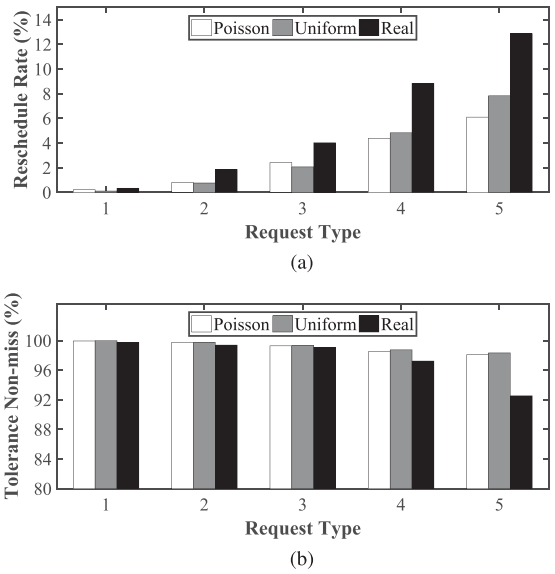

Fig. 14. (a) Reschedule requests rate. (b) Delay tolerance non-miss rate.

instance type and number of instances are used [21]. A variation in machine performance will change the relationship between the backend capacity and the number of instances, which may lead to over-provisioning or under-provisioning that may hurt the backend performance or utilization. Performance variations in cloud are caused by various reasons that are hard for us to monitor or control, e.g., multi-tenancy, networks, and instance types. In our future work, we will explore how to address performance variations, e.g., seek for key indicators that reflects performance variations to adjust the backend planning.

6.3 Heterogeneous Instance Types

Cloud service providers provide a wide selection of instance types that are optimized to fit different usage cases. In Razor, we simplify the backend capacity as the number of requests per minute, and suggests the corresponding number of instances, but not consider the influence of different type of instances on the cost. Using different types of instances to cater for the same capacity (as defined as the number of requests per minute) will result in different costs. In the future, we will further model the relationship between the instance type and the cost, and to suggest not only the number of instances, but also the most desirable instance type to the application developers.

6.4 Switching Costs

Frequent changes in backend configuration will incur considerable switching costs. In our implementation of Razor, we choose one hour as the cycle for backend optimization and adjustment, which is relatively long and the switching cost is neglected. However, if the developer chooses a short adjustment cycle, the switching costs in the optimization problem should be taken into consideration. We can introduce a penalty term in the optimization problem to control frequent changes of backend configurations, e.g., $\beta \log |N - N_0|$, in which N_0 is the current configuration, and the value of β can be adjusted to avoid significant deviation from the current configuration.

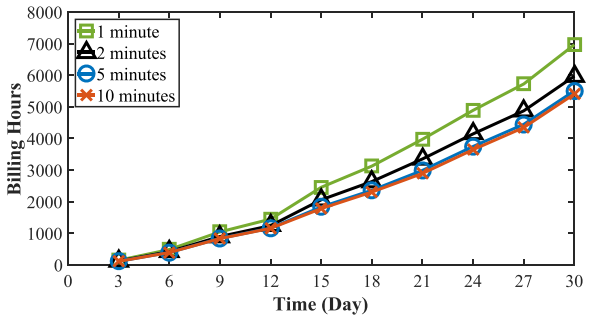

Fig. 15. Instance hours with different delay constraints.

6.5 Pending Queue Starvation Problem

The backend capacity planning optimization problem (1) aims to ensure that the backend capacity planned for the hour is able to serve all user traffic generated in the hour, thus the pending queue will be empty at the end of the current hour and will not affect the capacity planning for the next hour. In practice, if the pending queue at the end of an hour is non-empty, the backend planning for the next hour can be adjusted as $N + \Delta N$, in which N is derived from (1) and ΔN depends on the size of the pending queue. The computation of ΔN based on the size of the pending queue will be an interesting future direction.Another problem is that the pending queue may grow too long due to a large deviation from the real user demand to the prediction. In this case, the developer can set a threshold for the size of the pending queue, beyond which extra instances will be launched to avoid the pending queue starvation problem. In our experiments, we set a CPU-threshold-based scaling policy. If the CPU utilization is higher than the given threshold, one more instance will be added to avoid the pending queue starvation problem.

7 Related Work

Dynamic Resource Provisioning. Several dynamic resource provisioning approaches have been proposed to meet the application performance requirement in cloud environment [16], [22], [23], [24]. These works mainly adopt control theory to achieve an autonomic dynamic resource system. In [22], Ruth et al. designed designed a virtualization-based computational resource sharing platform, which allows dynamic machine trading to meet the request demand. Ali-Eldin et al. [16] introduced a hybrid adaptive elasticity controller with a proactive controllers for scaling down resources to meet the service level agreements (SLAs) requirements and a reactive approach for scaling up resources. Bodík et al. [23] combined statistics and machine learning techniques to address the perceived shortcomings of using closed-loop control. In [24], Wang et al. proposed a fuzzy model predictive controller that combines the control theory and the fuzzy rules to automatically manage the resources in a virtualized system. These works show interesting results of improving performance of cloud computing platforms but do not target at minimizing the total running cost. Besides, the mobile traffic shows high short-term variations, making it difficult or expensive to scale resources frequently to closely match the request demand.

Workload Scheduling. Scheduling bas been applied to automate the management of traffic with multiple optimization criteria, such as budget constraints, security deadlines and

other SLAs. Wu et al. [25] developed a selection model to minimize infrastructure cost and meet multiple service provider SLAs parameters in the cloud. In [26], a greedy algorithm is proposed to efficiently schedule workload for practical multimedia cloud to minimize the response time. Mao et al. [27] proposed to schedule instances and determine the suitable execution plan in a cost-efficient way, considering both the number and the type of instances. Besides, one of the most popular solution is to convert the resource allocation optimization problem into a linear programming problem with SLAs constraints [28], [29], but the linear programming approach may not scale well when the number of parameters becomes large. In [30], Pandey et al. considered the cost of bulky data transfer between tasks in scientific applications. Different from the time-consuming workload in the cloud, mobile applications requests features short processing time, large volume and high fluctuations, thus existing workload scheduling algorithms that are based on required CPU/memory resource and deadlines are not applicable for mobile request scheduling.

Prediction Techniques. Workflow or resource usage prediction has been largely used for scaling resources in the cloud [31], [32], [33], [34], [35], [36]. In [31], Islam et al. presented prediction-based resource measurement and provisioning strategies using neural network and linear regression to satisfy upcoming resource demands. Iqbal et al. [32] used time-series forecasting with reactive techniques to automatic allocate resources on a cloud. Niu et al. [33], [34] introduced a statistical model and used time-series analysis techniques to forecast the demand of videos and the performance in peer-assisted VoD services, based on which the video service provider can dynamically book bandwidth resources to match the fluctuated demand. In [35], Wu et al. proposed an epidemic model to predict the viewing requests in a social media application, and scale the application to efficiently store and migrate contents among different data centers in a geo-distributed cloud. In [36], Roy et al. described a look-ahead resource allocation algorithm using predictive models control based on a limited horizon. In this paper, we have utilized machine learning algorithms and historical data to predict future request demand, which serves as input for backend capacity optimization.

8 Conclusion

One of the major objectives of mobile application developers is to provide guaranteed service quality with minimum costs. In this paper, we present Razor, a novel mobile backend management mechanism that can significantly reduce the cost of backend provisioning. In stark contrast with existing solutions that try to match backend resources with highly-fluctuated traffic, Razor exploits a different way, that is, smoothing out mobile traffic profiles by postponing delay-tolerant user requests. Razor features an integrated two-tier structure that combines periodic backend capacity optimization with dynamic request scheduling to achieve an effective backend management. A fully-functional prototype of Razor has been implemented on Amazon EC2, and experimental results on 3 representative mobile traffic traces confirm that Razor can help developers cut down expenses on backend while preserving the user experience.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61702380, Hubei Provincial Natural Science Foundation of China under Grant No. 2017CFB134, Hubei Provincial Technological Innovation Special Funding Major Projects under Grant No. 2017AAA125, and the Wuhan University Independent Research fund under Grant 2042018kf0005. The co-authors would also like to acknowledge the generous research support from a NSERC Discovery Research Program and the National Natural Science Foundation of China with grant number 61772406.

References

[1] Statistas, "Number of mobile app downloads worldwide in 2016, 2017 and 2021 (in billions)." [Online]. Available: https://www.statista.com/statistics/271644/forecast-of-mobile-app-downloads/
[2] Amazon EC2. [Online]. Available: https://aws.amazon.com/cn/ec2/, Last accessed: 2019.
[3] Google Compute Engine. [Online]. Available: https://cloud.google.com/compute/, Last accessed: 2019.
[4] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2012, pp. 423–430.
[5] C. Shi, K. Joshi, R. K. Panta, M. H. Ammar, and E. W. Zegura, "CoAST: Collaborative application-aware scheduling of last-mile cellular traffic," in *Proc. ACM Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2014, pp. 245–258.
[6] World Cup 1998 Trace. [Online]. Available: ftp://ita.ee.lbl.gov/html/contrib/WorldCup.html, last accessed: 2019.
[7] EC2 Auto Scaling. [Online]. Available: https://aws.amazon.com/autoscaling, Last accessed: 2019.
[8] Right Scale. [Online]. Available: https://www.rightscale.com/, Last accessed: 2019.
[9] Amazon EC2 Price. [Online]. Available: https://aws.amazon.com/ec2/pricing/, Last accessed: 2019.
[10] M. Van Der Voort, M. Dougherty, and S. Watson, "Combining Kohonen Maps with ARIMA time series models to forecast traffic flow," *Transp. Res. Part C: Emerging Technol.*, vol. 4, no. 5, pp. 307–318, 1996.
[11] B. M. Williams and L. A. Hoel, "Modeling and forecasting vehicular traffic flow as a seasonal ARIMA process: Theoretical basis and empirical results," *J. Transp. Eng.*, vol. 129, no. 6, pp. 664–672, 2003.
[12] K. Y. Chan, T. S. Dillon, J. Singh, and E. Chang, "Neural-network-based models for short-term traffic flow forecasting using a hybrid exponential smoothing and Levenberg–Marquardt algorithm," *IEEE Trans. Intell. Transp. Syst.*, vol. 13, no. 2, pp. 644–654, Jun. 2012.
[13] Y. Lv, Y. Duan, W. Kang, Z. Li, and F.-Y. Wang, "Traffic flow prediction with big data: A deep learning approach," *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 2, pp. 865–873, Apr. 2015.
[14] F. Baccelli, B. Błaszczyszyn, et al., "Stochastic geometry and wireless networks: Volume II applications," *Found. Trends® Netw.*, vol. 4, no. 1/2, pp. 1–312, 2010.
[15] S. Sen, Y. Jin, R. Guérin, and K. Hosanagar, "Modeling the dynamics of network technology adoption and the role of converters," *IEEE/ACM Trans. Netw.*, vol. 18, no. 6, pp. 1793–1805, Dec. 2010.
[16] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Proc. IEEE Netw. Operations Manage. Symp.*, 2012, pp. 204–212.
[17] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proc. Int. Conf. Autonomic Comput.*, 2013, pp. 23–27.
[18] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscaling policies for complex workflows," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2017, pp. 75–86.
[19] A. D. Ferguson, P. Bodík, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2012, pp. 99–112.
[20] S. Frühwirth-Schnatter, "Data augmentation and dynamic linear models," *J. Time Series Anal.*, vol. 15, no. 2, pp. 183–202, 1994.

[21] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2016, pp. 363–378.
[22] P. Ruth, P. McGachey, and D. Xu, "VioCluster: Virtualization for dynamic computational domains," in *Proc. IEEE Int. Cluster Comput.*, 2005, pp. 1–10.
[23] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," in *Proc. USENIX Workshop Hot Topics Cloud Comput.*, 2009, Art. no. 12.
[24] L. Wang, J. Xu, M. Zhao, and J. Fortes, "Adaptive virtual resource management with fuzzy model predictive control," in *Proc. Int. Conf. Autonomic Comput.*, 2011, pp. 191–192.
[25] L. Wu, S. K. Garg, and R. Buyya, "SLA-Based resource allocation for software as a service provider (SaaS) in cloud computing environments," in *Proc. IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2011, pp. 195–204.
[26] X. Nan, Y. He, and L. Guan, "Optimization of workload scheduling for multimedia cloud computing," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2013, pp. 2872–2875.
[27] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 1–12.
[28] A.-M. Oprescu and T. Kielmann, "Bag-of-tasks scheduling under budget constraints," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 351–359.
[29] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove, "Cost-optimal scheduling in hybrid IaaS clouds for deadline constrained workloads," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2010, pp. 228–235.
[30] S. Pandey, L. Wu, S. M. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *Proc. IEEE Int. Conf. Adv. Inf. Netw. Appl.*, 2010, pp. 400–407.
[31] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Comput. Syst.*, vol. 28, no. 1, pp. 155–162, 2012.
[32] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Generation Comput. Syst.*, vol. 27, no. 6, pp. 871–879, 2011.
[33] D. Niu, Z. Liu, B. Li, and S. Zhao, "Demand forecast and performance prediction in peer-assisted on-demand streaming systems," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2011, pp. 421–425.
[34] D. Niu, H. Xu, B. Li, and S. Zhao, "Quality-assured cloud bandwidth auto-scaling for video-on-demand applications," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2012, pp. 460–468.
[35] Y. Wu, C. Wu, B. Li, L. Zhang, Z. Li, and F. Lau, "Scaling social media applications into geo-distributed clouds," *IEEE/ACM Trans. Netw.*, vol. 23, no. 3, pp. 689–702, Jun. 2015.
[36] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2011, pp. 500–507.

Yanjiao Chen received the BE degree in electronic engineering from Tsinghua University, in 2010, and the PhD degree in computer science and engineering from the Hong Kong University of Science and Technology, in 2015. She is currently a professor with the School of Computer Science, Wuhan University, China. Her research interests include computer networks, wireless system security, cloud computing, and network economy. She is a member of the IEEE.

Long Lin received the BE degree in computer science from Central South University, China, in 2016 and is currently working toward the master's degree in the School of Computer Science, Wuhan University, China. His research interest include cloud computing. He is a student member of the IEEE.

Baochun Li received the BE degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995, and the MS and PhD degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000, respectively. Since 2000, he has been with the Department of Electrical and Computer Engineering, University of Toronto, where he is currently a professor. His research interests include large-scale distributed systems, cloud computing, peer-to-peer networks, applications of network coding, and wireless networks. He has co-authored more than 290 research papers, with a total of more than 13,000 citations, an H-index of 59, and an i10-index of 189, according to Google Scholar Citations. He was the recipient of the IEEE Communications Society Leonard G. Abraham Award in the field of communications systems, in 2000. He is a member of the ACM and a fellow of the IEEE.