Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms[*]

Eli Cortez
Microsoft
eli.cortez@microsoft.com

Anand Bonde
Microsoft Research
abonde@microsoft.com

Alexandre Muzio
ITA, Brazil
alemuzio@aluno.ita.br

Mark Russinovich
Microsoft
mark.russinovich@microsoft.com

Marcus Fontoura
Microsoft
marcusfo@microsoft.com

Ricardo Bianchini
Microsoft Research
ricardob@microsoft.com

ABSTRACT

Cloud research to date has lacked data on the characteristics of the production virtual machine (VM) workloads of large cloud providers. A thorough understanding of these characteristics can inform the providers' resource management systems, e.g. VM scheduler, power manager, server health manager. In this paper, we first introduce an extensive characterization of Microsoft Azure's VM workload, including distributions of the VMs' lifetime, deployment size, and resource consumption. We then show that certain VM behaviors are fairly consistent over multiple lifetimes, i.e. history is an accurate predictor of future behavior. Based on this observation, we next introduce Resource Central (RC), a system that collects VM telemetry, learns these behaviors offline, and provides predictions online to various resource managers via a general client-side library. As an example of RC's online use, we ~~modify Azure's VM scheduler to leverage predictions in oversubscribing servers~~ (with oversubscribable VM types), while retaining high VM performance. Using real VM traces, we then show that the ~~prediction-informed schedules increase utilization and prevent physical resource exhaustion~~. We conclude that providers can exploit their workloads' characteristics and machine learning to improve resource management substantially.

[*]Cortez and Bonde contributed equally. Muzio was an intern at Microsoft.

SOSP'17, October 28, 2017, Shanghai, China
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132772

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Computing methodologies → Machine learning approaches**;

KEYWORDS

Cloud workloads, machine learning, predictive management

ACM Reference Format:
Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of SOSP'17*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3132747.3132772

1 INTRODUCTION

Motivation. Cloud computing has been expanding at a fast pace, especially as enterprises continue to move their operations to large cloud providers such as Microsoft Azure, Amazon Web Services (AWS), and Google Cloud Platform (GCP). Due to heated marketplace competition, providers have been under pressure to produce attractive features and services, while controlling their datacenter costs. These factors combine to expose providers to a wide variety of workloads (from both external customers and their own internal services) that must share a common datacenter infrastructure. Providing good performance, availability, and reliability under these conditions can be expensive without sophisticated (but practical and scalable) resource management.

Unfortunately, research on cloud resource management to date has lacked a thorough understanding of the key characteristics of the workloads of large commercial providers. For example, no prior study has explored the lifetime (time between creation and termination) or resource consumption distributions of these providers' production virtual machines

(VMs). Instead, the prior work has mostly used real but non-VM workloads, synthetic VM workloads, and/or focused on managing resources via general but often impractical techniques for a large cloud provider. For example, many papers explore (sometimes offline) workload profiling and aggressive online resource reallocation, via dynamic monitoring, scheduling, and/or live VM migration [2, 7, 20, 21, 24, 27]. In practice, offline profiling is infeasible because the workloads' inputs are often unavailable until VMs run in production. Online profiling is challenging, as it is hard to determine when an arbitrary VM has shown representative behavior. Application-level performance (e.g., tail latency) monitoring is usually not possible, as it requires help from applications. Finally, live migration retains contended resources for a relatively long time (e.g., it cannot free up memory pages before successfully migrating them) and can cause widespread network traffic bursts. Practical uses of these techniques require extreme care.

We argue that resource management can become more effective and practical for large providers with a deeper understanding of their VM workloads' key characteristics. Moreover, if these characteristics can be accurately predicted, improvements could be even greater. For example, accurately predicting resource utilization at VM deployment time would allow resource-contention-aware VM co-location (mitigating the need for VM migration). Similarly, run-time lifetime predictions would allow the health management system to estimate when all the VMs running on a misbehaving server will likely terminate (facilitating server maintenance without VM migration or downtime). We are unaware of prior work showing that the characteristics of large providers' production VMs can be accurately predicted for better resource management.

Thus, there is a need for software that can produce such predictions and enable the providers' resource management systems (e.g., the VM scheduler, the server health manager) to leverage them. Some prediction-serving systems [5, 6, 10] have been proposed recently, but have not been studied in the context of prediction-based resource management.

Our work. In this paper, we first introduce a characteri-~~zation of Azure's (first- and third-party) VM workload~~, including distributions of the VMs' size, lifetime, resource consumption, utilization pattern, and deployment size. Researchers can use these distributions to produce realistic cloud workloads for their own work. Alternatively, they can use the sanitized production traces we have placed at https://github.com/Azure/AzurePublicDataset. The traces contain a subset of our data, but exhibit roughly the same overall trends as the full dataset.

Our characterization shows that many types of VM behavior are fairly consistent over multiple lifetimes, when observed from the perspective of each cloud customer. This observation suggests that prior history may be an accurate predictor of the future behavior of the customers' VMs, so machine learning algorithms could be used online to produce VM behavior predictions.

Based on this observation, we introduce Resource Central (RC), a system that collects VM telemetry, periodically learns these behaviors into prediction models offline, and provides behavior predictions online to various resource management systems. Unlike other systems, RC serves predictions from a client-side library, which caches prediction results, models, and feature data. The library API is simple yet general, so it can be used with many types of resource managers, learning algorithms, prediction models, and feature data. RC's models and feature data are currently in production, being used manually by engineers and data scientists for analysis and system design. The changes to the systems that will leverage RC are still being productized.

As an example of RC's online use, we describe our modifications to Azure's VM scheduler, which selects a physical server for each new VM. Specifically, we modify the scheduler to collect high-percentile utilization predictions to use in oversubscribing physical servers with "oversubscribable" VM types (e.g., first-party VMs that run non-customer-facing workloads), while retaining good VM performance.

Our evaluation starts by quantifying the accuracy of RC's predictions of six metrics, using our VM telemetry. The results show overall prediction accuracies between 79% and 90%, depending on the metric. For example, RC predicts the average CPU utilization of a new VM with 81% accuracy. We then quantify the performance of RC's components, and confirm that models and feature data are compact and fast enough to be executed on the client-side. To evaluate the benefit of predictions, we explore our modified VM scheduler using real VM traces and simulation. Our results show that prediction-informed VM schedules enable safe oversubscription, while keeping VM deployment failures low.

Summary and conclusions. Our contributions are:

- We present a detailed characterization of several VM workload behaviors from Microsoft Azure;
- We show that these behaviors can be accurately predicted;
- We describe Resource Central, a large-scale system for producing, storing, and using such predictions;
- We describe modifications to Azure's VM scheduler that leverage predictions to improve server selection;
- We present extensive results evaluating Resource Central and our modified VM scheduler; and
- We make a large VM dataset available to the community.

We conclude that cloud providers can exploit the characteristics of their workloads and machine learning techniques to improve resource management substantially.

2 RELATED WORK

Cloud VM workload characterization. We are unaware of other characterizations of the VM workload of real public cloud providers, except for [15] which focused solely on resource demand volatility and pricing. Other works studied cloud platform performance [16] and how users implement their front-end services in the cloud [11].

The closest prior works [8, 19, 22] did *not* address public cloud workloads. Rather, they characterized a month-long trace of 12k bare-metal servers running first-party container-based (i.e., non-VM) workloads at Google [25]. In contrast, we characterize Azure's *entire VM workload* over three months, including third-party VMs. In fact, third-party VMs often exhibit different characteristics than first-party ones, as we show in Section 3.

More importantly, VM workloads fundamentally differ from bare-metal container workloads. Mainly for security reasons, public cloud providers must encapsulate their customers' workloads using VMs. Unfortunately, VMs impose higher creation/termination overheads than containers running on bare metal, so they are likely to live longer, produce lower resource utilization, and be deployed in smaller numbers. For example, a user who wants to run multiple MapReduce jobs may create a set of VMs sized for the jobs' maximum resource needs; she would only destroy/shutdown the VMs when all jobs have completed. In a container-based system, each map and reduce task of each job would likely receive a different container, which can be more accurately sized for its task. As another example, services that assign a VM to each user (e.g., gaming) may pool and reuse VMs, so that the overhead of VM creation becomes invisible to users.

Machine learning and prediction-serving systems. Prior works have recently proposed several frameworks, such as TensorFlow [1], Caffe [13], and MLLib [18], for producing machine learning models. Our work is orthogonal to them, as RC relies on TLC, a Microsoft-internal state-of-the-art framework that implements many learning algorithms.

Instead, RC is comparable to recent prediction-serving systems [5, 6, 10]. In contrast with these systems, RC caches prediction results, models, and feature data on the *client* side. This approach enables the system to operate even when the data store or the connectivity to it is unavailable. In addition, it provides higher performance by removing the interconnect and store from the critical performance path.

Predicting cloud workloads. The literature on predicting workload behaviors is extensive. These works predict resource demand, resource utilization, or job/task length for provisioning or scheduling purposes [4, 9, 12, 14, 23]. In contrast, we predict a broader set of VM behaviors (including VM lifetimes, maximum deployment sizes, and workload classes) for a broader set of purposes (including health management and power capping [26]). Most importantly, our prediction results derive from a real cloud platform, its workload and machine learning framework.

Prediction-based scheduling. The literature on task/container/VM scheduling is also vast. Many of these works use online predictions of resource usage or performance interference [3, 7, 20, 27]. Unfortunately, they are often impractical for a large provider, relying on (offline or online) workload profiling, application-level performance monitoring, *short-term* load predictions, and/or aggressive resource reallocation (e.g., via live migration). Live migration is particularly problematic, as it retains contended server resources and may produce traffic bursts; it is better to place VMs where they can stay. Thus, as an example of RC's online use, we propose changes to Azure's VM scheduler that leverage predictions of *long-term* high-percentile resource usage to implement oversubscription in a safe and practical manner.

To guarantee within-server performance isolation across VMs, we also need mechanisms for interference detection (e.g., hardware counters [20, 27]) and prevention (e.g., fine-grained resource partitioning [17]), but they are orthogonal to our work. For example, Heracles (fine) and RC (coarse) operate at different granularities and time-scales. RC is needed in first placing VMs on servers, so Heracles does not have to overly punish whatever workloads are considered low-priority on the chosen server.

3 CHARACTERIZING CLOUD VM WORKLOADS

We now explore several aspects of Azure's VM workload. Understanding them enables many platform optimizations.

Context and terminology. Azure hosts both first- and third-party workloads, split into Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) VMs. In Azure's offering, PaaS defines functional roles for VMs, e.g. Web server VM, worker VM.

The first-party workloads comprise internal VMs (e.g., research/development, infrastructure management) and first-party services (e.g., communication, gaming, data management) offered to third-party customers. The third-party workloads are VMs created by external customers. Other than the limited information contained in our dataset (details below), we have no visibility into third-party uses of internal services or third-party workloads. The dataset corresponds to many tens of millions of VMs from many tens of thousands of first- and third-party users. [1]

Regardless of their first- or third-party status, customers create one or more Azure subscriptions. After creating a subscription, the customer can deploy VMs into a region

[1]Due to confidentiality reasons, we omit certain exact numbers, instead focusing on more relevant workload statistics and trends.

(one or more datacenters) she selects. We refer to a "VM deployment" as a set of VMs that the customer groups and manages together. All VMs in a deployment execute in one "cluster" in the chosen region, i.e. a large aggregation of servers within which a deployment needs to fit. Each VM belongs to a "role", which refers to the type or functionality performed by the VM (e.g., IaaS VM, PaaS Web server).

Dataset. Our dataset contains information about every VM running on Azure from November 16, 2016 to February 16, 2017. The information includes (1) identification numbers for the VM, and the deployment and subscription to which it belongs; (2) the VM role name; (3) the VM size in terms of its maximum core, memory, and disk allocations; and (4) the minimum, average, and maximum VM resource utilizations (reported every 5 minutes). A sanitized subset of these data is available at https://github.com/Azure/AzurePublicDataset.

Focus. We focus on aspects of real cloud workloads that have an impact on resource management. Thus, we next explore distributions of VM type (IaaS vs PaaS), virtual resource (CPU) usage, VM size (CPU cores and memory space), maximum deployment size, VM lifetime, workload class (likely interactive vs delay-insensitive), and VM inter-arrival times. We present both full platform and per-subscription perspectives. The subscription is a natural unit for us, as it is tied to billing, naming, quotas, and access control. Thus, it typically represents users who (sometimes repeatedly) execute logically related workloads (e.g., many load-balanced copies of a Web server, or Web servers and databases for an ecommerce service), and users from the same organization (e.g., the ecommerce company). As such, the subscription embodies commonalities that enable accurate predictions. At the end of each subsection, we discuss the implications for cloud resource management. To close the section, we explore correlations between metrics.

3.1 VM type

Entire cloud platform perspective. The workload is almost exactly split between IaaS (52%) and PaaS (48%) in terms of VM counts; first-party workloads have slightly more IaaS VMs (53% vs 47%), whereas third-party workloads have slightly more PaaS VMs (53% vs 47%). However, PaaS actually dominates the resource consumption with roughly 61% of the total core hours. Even more interestingly, third-party IaaS VMs consume significantly more core hours than their PaaS counterparts (85% vs 15%, respectively), whereas the opposite is true of first-party workloads (23% vs 77%).

Per-subscription perspective. We find that VMs from the same subscription are almost always of the same type; 96% of the subscriptions create VMs of a single type.

Implications for resource management. Some PaaS VMs reveal information that the cloud provider can use in resource management. For example, a PaaS Web server VM is likely to be customer-facing, and may serve as the front-end to an interactive service. The provider must ensure the best possible performance for such services. In contrast, IaaS VMs reveal no information and must be treated carefully. In Section 3.6, we describe an approach to inferring the class of VM workload that works well for both IaaS and PaaS.

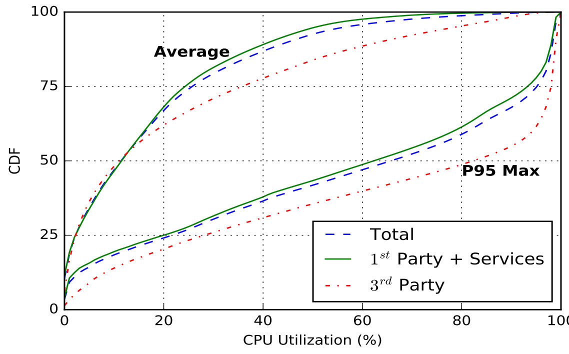

Figure 1: Average and P95 of max CPU utilizations.

3.2 Virtual resource usage

Entire cloud platform perspective. We illustrate the virtual resource usage by quantifying CPU utilization per VM. Figure 1 depicts the Cumulative Distribution Function (CDF) of the average virtual CPU utilizations for each VM, and the CDF of the 95th-percentile of the maximum virtual CPU utilizations (P95 Max). Recall that the utilization measurements correspond to 5-minute intervals. For example, the figure shows that 60% of the VMs (Y-axis) have an average CPU utilization (X-axis) lower than 20%. Similarly, 40% of them have a 95th-percentile utilization lower than 50%.

We draw two key observations from this figure. First, a large percentage of VMs exhibit low average CPU utilizations, especially for first-party workloads. However, average utilizations do not reflect the VMs' full needs, so we must consider the 95th percentiles. Though more than one third of VMs exhibit low CPU utilizations even at the 95th percentile, a large percentage of them exhibit very high utilizations ($>$ 80%), especially for third-party workloads.

Second, the first-party distributions tend to show lower average and 95th-percentile utilizations than the third-party distributions. At least two factors contribute to this observation: (1) many first-party services require consistently low latency, which can be easily achieved by overprovisioning VMs; and (2) our dataset includes a non-trivial percentage (15%) of first-party VMs used for testing the performance and scalability of VM creation (i.e., VMs get created and quickly killed without doing any actual work).

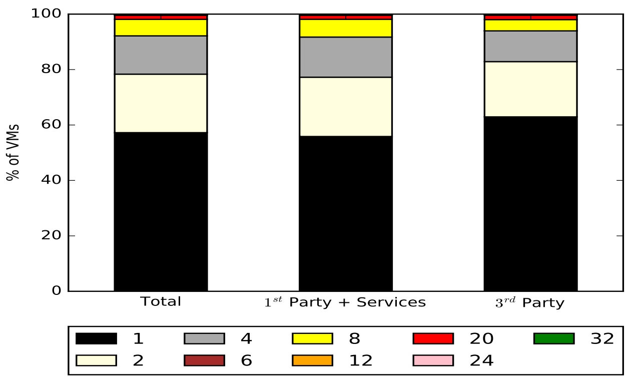

Figure 2: Number of virtual CPU cores per VM.

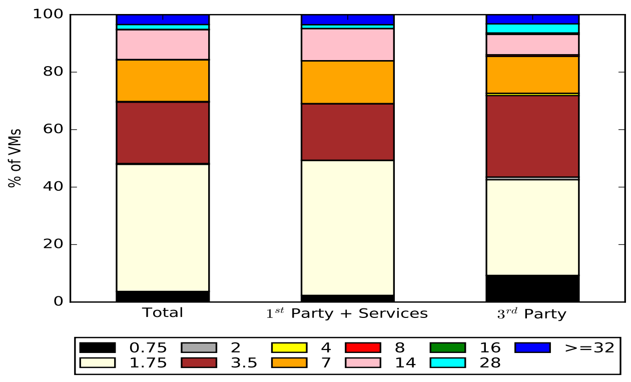

Figure 3: Amount of memory per VM in GBytes.

Per-subscription perspective. We find that VMs from the same subscription tend to exhibit similar behaviors with respect to utilization. Specifically, the coefficient of variation (CoV = standard deviation divided by average) of the average and 95^{th}-percentile virtual CPU utilizations for each subscription indicates low variance in these metrics for most subscriptions. For example, 80% of subscriptions exhibit a CoV of their average CPU utilizations that is smaller than 1.
Implications for resource management. Since at high percentiles many VMs require nearly their entire resource allocation, the provider must be careful when selecting physical servers for new VMs. Even when not oversubscribing resources, the high resource usage may cause resource contention. In this case, the provider has to resort to (expensive and slow) live VM migration or accept potential workload performance loss.

Nevertheless, a large percentage of VMs require many fewer resources than their allocations. Thus, there is a significant potential for oversubscribing physical resources, as long as the provider can predict (at VM creation time) with high confidence which VMs will require most of their allocations; these VMs can then be scheduled on different servers.

Focusing on individual subscriptions should simplify this prediction task, as the space of high utilizations is more consistent than across the entire population of VMs.

3.3 VM size

We now study the distribution of VM sizes. We define the size of a VM as the amount of CPU and memory that the VM's owner requested for it.
Entire platform perspective. We illustrate the VM sizes via the number of virtual CPU cores and amount of memory per VM. Figures 2 and 3 present the corresponding breakdowns using stacked bars, one each for first-party, third-party, and all workloads. The figures show that most VMs require few virtual cores (almost 80% of VMs require 1-2 cores) and relatively little memory (70% of VMs require less than 4 GBytes). This reflects a scale-out design pattern where one prefers more numerous small VMs over fewer large VMs.

The figures also show that first- and third-party customers create VMs of comparable sizes, except that the latter users create a larger percentage of 3.5-GByte and 0.75-GByte VMs, and a smaller percentage of 1.75-GByte VMs.
Per-subscription perspective. Subscriptions are remarkably consistent in terms of their VM sizes; nearly all subscriptions show CoVs of cores and memory lower than 1.
Implications for resource management. As small VMs dominate, it is easier to fill holes in the server packing and reduce fragmentation. Despite this, packing is complex as it must consider multiple resource dimensions and constraints.

3.4 Maximum deployment size

Users do not always deploy their VMs to each region (a cluster in the region) in one shot, so each deployment may grow (and shrink) over time before it is terminated.
Entire platform perspective. Thus, we now consider the deployment sizes across the platform. While studying this data, we realized that some users are not using the concept of deployment in the way it was intended, i.e. to cleanly group and manage logically related VMs within a subscription. In fact, at least one large first-party service creates many single-VM deployments, instead of extending an existing deployment each time. To approximate the original intent, we redefine a deployment as the set of VMs from each subscription that are deployed to a region during a day. Assuming this definition, the CDFs of maximum deployment sizes appear in Figure 4. The figure shows that most deployments are small: roughly 40% of them include a single VM, and 80% have at most 5 VMs. Moreover, the figure shows that third-party users deploy VMs in smaller groups than first-party ones. These observations reflect patterns that favor smaller VM groups; when users deploy multiple groups, they prefer them to be spread across multiple regions.

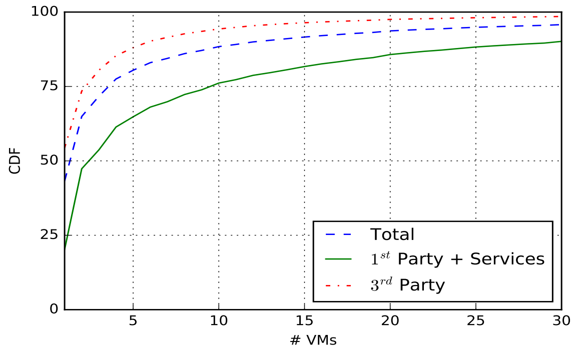

Figure 4: Max number of VMs in each deployment.

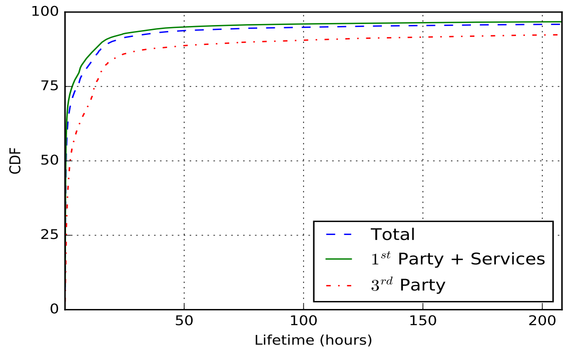

Figure 5: VM lifetime.

Looking at deployment sizes in terms of cores or memory shows the expected trends. We do not depict these data to keep deployment and VM sizes separate until Section 3.8.

Per-subscription perspective. Again, subscriptions show remarkably consistent behavior with respect to their maximum deployment sizes; nearly all subscriptions exhibit a CoV for this metric less than 1.

Implications for resource management. The cluster must have enough capacity to host the maximum size of a deployment, and avoid eventual deployment failures (or long communication delays across VMs of the same deployment). The dominance of small deployments suggests that the provider need not reserve large amounts of capacity within each cluster for deployment growth, as long as it can predict (when deployments are first created) with high confidence the few deployments that could become large. The per-subscription data shows that focusing on individual subscriptions may increase prediction accuracy.

3.5 VM lifetime

Entire platform perspective. Figure 5 presents the CDFs of VM lifetimes (how long VMs last from creation to termination) in our dataset, including only VMs that started and completed in our observation period. As this group represents 94% of the VMs in our dataset, we can confidently make statements about VM lifetimes. (There are 2% of VMs that started before and completed after the period, and 4% of VMs that either started before or completed after the period.)

The figure shows that a large percentage of first-party VMs tend to live shorter (less than 15 minutes) than their third-party counterparts. The main reason is the first-party VM-creation testing workloads we mention above. The figure also shows a broad spectrum of lifetimes, but most lifetimes are relatively short. The curves show a knee around 1 day (more than 90% of lifetimes are shorter), and then almost flatten out. This suggests that, if a VM runs for 1 day, it will very likely run much longer. Perhaps most interestingly, the relatively small percentage of long-running VMs actually account for >95% of the total core hours (not shown).

Per-subscription perspective. Many subscriptions show consistent lifetime behavior. For example, roughly 75% of them exhibit lifetime CoVs lower than 1.

Implications for resource management. The health management system can schedule non-urgent server maintenance without producing VM unavailability or requiring live migration, if it can predict lifetimes accurately. In addition, with accurate lifetime predictions at VM creation time, the provider can schedule VMs that will complete at roughly the same time on the same servers. The per-subscription data shows that considering subscriptions individually may increase lifetime prediction accuracy.

3.6 Workload class

In considering tightly packing VMs onto physical servers, the provider might consider the VMs' resilience to resource contention and interference; highly interactive, customer-facing workloads require low tail response times (measured in milliseconds), so they are typically less resilient than batch or background (delay-insensitive) workloads. However, this resilience is difficult to ascertain, because the provider has little or no information about what is running on the VMs, especially in third-party IaaS scenarios. Even if the provider were to request this information in advance, users may be unwilling or unable to provide it.

For these reasons, we perform an analysis of the (average) CPU utilization time series to infer whether a VM's workload is likely to be interactive. Since interactive workloads tend to exhibit diurnal cycles (people are active during the day and sleep at night), our analysis attempts to find periodicity in the utilization time series over 3 days, using the Fast Fourier Transform (FFT) algorithm. The FFT is ideal because it can detect periodicity at multiple time scales. Our algorithm categorizes VMs into two classes: potentially interactive (those that exhibit periodic behavior at the diurnal

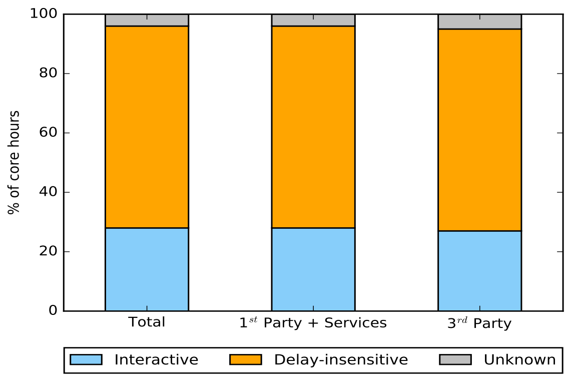

Figure 6: Workload classes and their use of core hours.

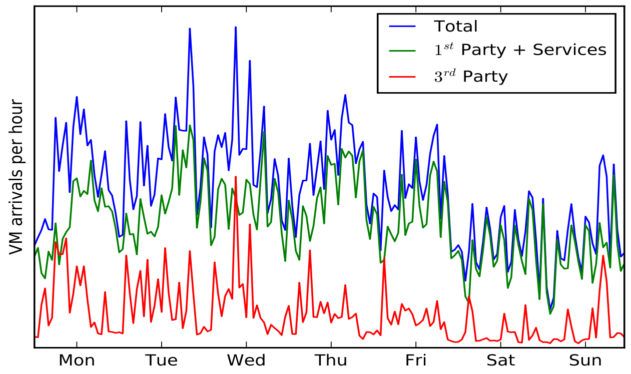

Figure 7: Time series of arrivals per hour at one region.

scale) and delay-insensitive (e.g., batch workloads, development and test workloads). Obviously, it is conceivable that some background VMs could increase/decrease activity at periodic intervals and appear periodic. We do not isolate these cases, because our classification seeks to be conservative (i.e., it is fine to classify a delay-insensitive workload as interactive, but not vice-versa). Moreover, some critical daily batch jobs have strict deadlines, so classifying them as interactive correctly reflects their performance needs. Finally, note that the algorithm works well even when the dominant resource is not the CPU, because the CPU is a good proxy for periodicity (e.g., network-bound interactive workloads exhibit more CPU activity during the day than at night).

Our periodicity analysis targets the VMs that run long enough for us to detect a reliable pattern (at least 3 days). We could have used other heuristics to classify VMs that live shorter. However, since VMs running longer than 3 days consume 94% of the core hours, our classification already covers the vast majority of the used resources.

We have validated our classification algorithm with customers accounting for more than 50% of the first-party VMs.

Entire platform perspective. Figure 6 depicts the percentage of total (left), first-party (middle), and third-party (right) core hours in each class. The "Unknown" class represents the VMs that do not last 3 consecutive days. The figure shows that delay-insensitive VMs consume most (roughly 68%) of the core hours, regardless of whether they are first-party or not. Still, a significant percentage of VMs execute potentially interactive workloads and consume roughly 28% of the core hours; these VMs must be managed carefully.

Per-subscription perspective. Inspecting the classification per subscription again shows that most subscriptions behave consistently; 76% of subscriptions with long-running VMs (at least one VM runs for 3 days) create VMs dominated by a single (usually delay-insensitive) class.

Implications for resource management. Our classification enables the provider to apportion resources according to performance constraints. For example, upon a power emergency, the power capping system [26] can assign full power to VMs running interactive workloads and a reduced budget to delay-insensitive VMs. The VM scheduler can tightly pack VMs that run delay-insensitive workloads onto physical servers, while loosely packing VMs that run interactive ones. Similarly, the provider can use the former classes of VMs to oversubscribe physical servers, while avoiding oversubscription of servers that run an interactive VM. In all scenarios, the provider must be able to predict with high confidence which VMs will run or are running delay-insensitive workloads. Even if such high confidence cannot be achieved, the provider may leverage our classification to help users select good VM and deployment sizes based on their workloads' characteristics (e.g., tighter sizes for delay-insensitive workloads). Focusing on subscriptions should increase prediction accuracy, as they behave consistently over time.

3.7 VM inter-arrival times

We now study the VM inter-arrival times at one of Azure's regions. Figure 7 depicts the arrival time series at hourly granularity over a random week. We can see that that arrivals are very bursty and diurnal with lower load on weekends, regardless of the type of workload. This pattern is representative of other regions as well. The burstiness is due to the small percentage of large deployments. We verified that the arrival times are heavy-tailed by fitting Weibull distributions (nearly perfectly) to them.

Implications for resource management. Diurnal arrivals suggest that the provider can schedule its internal batch computing or infrastructure testing workloads for the night without risking competition with customer workloads. Heavy-tailed arrivals suggest that burstiness can be significant, so the VM scheduler must be optimized for high throughput.

3.8 Correlations between metrics

Finally, we correlate the VM metrics we study using Spearman's method. Figure 8 shows the pair-wise correlations for

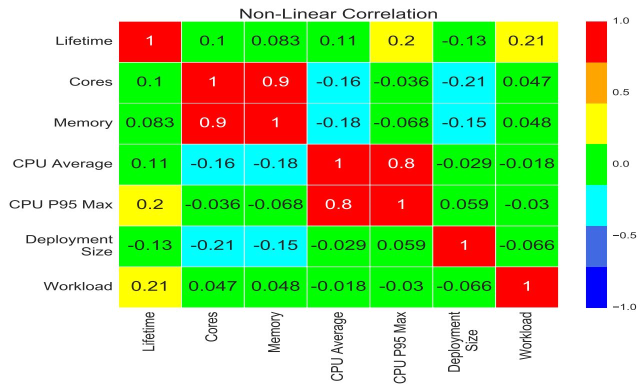

Figure 8: Spearman's correlations between metrics.

the entire platform as a heat map; we include the workload class by numbering the classes 1 (delay-insensitive) and 2 (interactive). A correlation factor near 1.0 reflects a positive relationship (a higher value of one metric means a higher value for the other), a factor near −1.0 reflects a negative relationship, whereas a factor near 0.0 reflects no relationship.

Obviously, each metric is strongly positively correlated with itself. In addition, the two utilization metrics are strongly positively correlated, and so are numbers of cores and amount of memory (the latter relationship derives from Azure's VM offerings). More interestingly, we see that the utilizations (especially 95^{th}-percentile) show a lightly positive relationship to lifetime and a slightly negative relationship to cores and memory per VM, meaning that VMs with higher utilization tend to be smaller and live longer. In contrast, lifetime has no relationship with number of cores or amount of memory per VM, whereas deployment size has a slightly negative relationship with these metrics (larger deployments require fewer resources per VM). Finally, the class shows a lightly positive relationship to lifetime, meaning that interactive VMs tend to live longer. The other metrics show no relationship to workload class.

The correlations are different when considering only third-party or only first-party workloads. Most strikingly, the utilization metrics are moderately positively correlated with deployment size in third-party workloads, whereas they are lightly negatively correlated in first-party ones.

4 RESOURCE CENTRAL

The previous section discussed several VM behaviors and metrics related to resource management, and the potential benefits of predicting them accurately.

To produce these predictions, we introduce Resource Central (RC), a system for ingesting VM telemetry, learning from past VM behaviors, producing models that can predict these behaviors, and executing the models (i.e., providing predictions) when client systems request them. Though we focus on predicting VM behaviors in this paper, RC is general and can be used for learning/predicting server effects as well, such as hardware failures. RC is in production, supporting engineers and data scientists. The systems that will use it are being productized and qualified within Azure.

Next, we discuss some RC use-cases and its design.

4.1 Example RC use-cases

We envision many uses for RC predictions of the metrics from Section 3. The following is a non-exhaustive list that expands on some examples from Section 3:

• **Smart VM scheduling.** Before selecting servers to run a set of new VMs, the VM scheduler can contact RC for predictions of the VMs' expected *resource utilizations*. With this information, the scheduler can select servers to balance the disk IOPS load, or to reduce the likelihood of physical resource exhaustion in oversubscribed servers.

• **Smart cluster selection.** Before selecting a cluster in which to create a VM deployment, the cluster selection system can query RC for a prediction of *maximum deployment size*. With this information, this system can select a cluster that will likely have enough resources.

• **Smart power oversubscription and capping.** During a power emergency (when the power draw is about to exceed a circuit breaker limit), the power capping system can query RC for predictions of VM *workload interactivity*, before apportioning the available power budget across servers. Ideally, VMs executing interactive workloads should receive all the power they may want, in detriment of VMs running batch and background tasks. Alternatively, the VM scheduler can request interactivity predictions before selecting servers, so that interactive and delay-insensitive workloads are segregated in different sets of servers.

• **Scheduling server maintenance.** When a server starts to misbehave, the health monitoring system can query RC for the expected *lifetime* of the VMs running on the server. It can thus determine when maintenance can be scheduled, and whether VMs need to be live-migrated to enable maintenance without unavailability. In addition, the VM scheduler can use the lifetime predictions to co-locate VMs that are likely to terminate roughly at the same time. This could facilitate other types of maintenance, such as OS updates.

• **Recommending VM and deployment sizes.** The cloud platform could provide a service to its customers that recommends the appropriate VM size and number of VMs at the time of each deployment. Using RC predictions of *workload class and resource utilization*, the service could recommend deployments where VMs predicted to be delay-insensitive would be more tightly sized than interactive VMs. (Existing rightsizing services from Azure, AWS, and GCP recommend new sizes for currently running VMs after long observation periods and do not account for the likely workload class.)

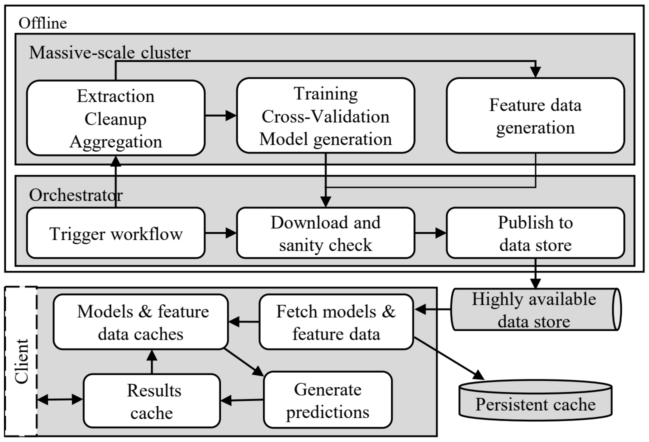

Figure 9: RC architecture (pull version).

Exploring all these use-cases is beyond the scope of this paper. As an example, Section 5 details the use of RC predictions for smart VM scheduling with server oversubscription.

4.2 Design and implementation

Principles. Our design for RC follows a few principles:

(1) For performance and availability, RC should be an independent and general system that is off the critical performance and availability paths of the systems that use it.
(2) For maintainability, it should be simple and rely on any existing well-supported infrastructures.
(3) For usability, it should require minimal modifications to the systems that use it, and provide an interface that is general enough for many use-cases.

Overview. Based on these principles, we designed RC as in Figure 9. As the figure illustrates, RC has offline and online components. The *offline* workflow consists of several tasks: data extraction, cleanup, aggregation, feature data generation, training, validation, and machine learning (ML) model generation. RC performs these phases on a centralized massive data processing cluster that collects all the VM telemetry from the cloud fabric. RC orchestrates these phases, sanity-checks the models and feature data, and publishes them (with version numbers) to an existing highly available store. The store is present in each datacenter.

In our current design, RC does not automatically select the appropriate ML modeling approach (e.g., regression tree, random forest) for each metric, leaving this task for data analysts who also (a) provide a "specification" describing the inputs to each model and (b) record them in the store.

The *online* part of RC uses a single, general, and thread-safe client dynamically linked library (DLL), within which the ML models execute to produce predictions. This DLL is the only view of RC for all external clients. ~~The client (e.g., VM scheduler, health monitoring system) calls the DLL~~ passing as input the model name and information about the VM(s) for which it wants predictions. We refer to this information as the *client inputs* to the models. Examples of client inputs are subscription id, VM type and size, and deployment size. Besides the client inputs, the model may require historical *feature data* as additional inputs, which RC fetches from the highly available store. For example, the subscription id for a new VM deployment may be one of the client inputs provided by the VM scheduler to a lifetime model. As an example of feature data, the lifetime model would also require information on historical lifetimes (e.g., percentage of short-lived and long-lived VMs to date) for the same subscription from the store.

ML modeling approaches. RC is agnostic to the specific modeling approach data analysts select; many approaches fit the framework. In our current implementation, analysts can select models from a large ML repository that runs on the data processing cluster. The repository also provides a library for executing the models at the clients.

The leftmost three columns of Table 1 list the modeling approaches we currently use for the metrics of Section 3: ~~Random Forests and Extreme Gradient Boosting Tree~~s as classifiers, and ~~Fast Fourier Transform (FFT) to detect periodicity in the utilization timeseries~~. Each model takes many features as inputs, deriving from a smaller number of attributes (e.g., VM type, VM size, guest operating system). Due to the predictive value of these features, RC can make predictions about VMs it has not seen before.

For classifying numeric metrics, we divide the space of possible predictions into a small number of buckets. Formulating these models as classifiers with buckets rather than regression algorithms, makes the metrics easier to predict. For example, it is easier to predict that utilization will be in the 50% to 75% bucket than predict that it will be exactly 53%. When the prediction must be converted to a number, the client can assume the highest value for the predicted bucket, the middle value, or the lowest value.

Client DLL. It is configurable per client. Given a model name, it finds out how to interpret the client inputs from the specifications for the model. It caches the prediction results, model, and feature data from the store in memory.

Table 2 lists the main DLL API methods. On a prediction request (a call to the Predict method), the DLL looks up the results cache first by hashing the model name and client inputs. On a hit, it returns the cached result. On a miss, the DLL looks up the other caches, and executes the model using client inputs and feature data inputs. The DLL caches the result of this execution. Each prediction result is typically a predicted value and a score. The score reflects the model's confidence on the predicted value. The client may choose to ignore a prediction when the confidence score is too low.

Metrics	Approach	#features	Model size	Feature data size
Avg CPU utilization	Random Forest	127	312 KB	376 MB
P95 max CPU utilization	Random Forest	127	311 KB	376 MB
Deployment size in #VMs	Extreme Gradient Boosting Tree	24	305 KB	368 MB
Deployment size in #cores	Extreme Gradient Boosting Tree	24	305 KB	368 MB
Lifetime	Extreme Gradient Boosting Tree	127	329 KB	376 MB
Workload class	FFT, Extreme Gradient Boosting Tree	34	152 KB	311 MB

Table 1: Metrics, ML modeling approaches, model and full feature dataset sizes.

API method	Parameters	Return	Description
initialize	none	boolean	Initializes client DLL
get_available_models	none	char*	Gets list of available models
predict_single	model_name, client_inputs*	struct prediction	Produces one prediction
predict_many	model_name, client_inputs**	struct prediction*	Produces multiple predictions
force_reload_cache	none	void	Refreshes memory & disk caches
flush_cache	none	void	Flushes memory & disk caches

Table 2: Main DLL API methods.

When the DLL cannot produce a prediction, it simply returns a no-prediction flag to the client system, which must be able to handle this situation.

Cache management. Result caching is configurable and implements a hash table accessible by hashing the model name and the client inputs. Thus, it works well when the client does not provide any rapidly changing inputs in the prediction calls. Each result cache entry stores only the corresponding prediction value and score.

The DLL caches model and feature data using either a pull (fetch from the store on-demand) or push (update cache when a change occurs at the store) approach. In one pull configuration, the DLL returns a no-prediction on *any* result cache miss; accesses to the remote store and/or model executions happen in the background. Clients can use this configuration when the models or full feature dataset are too large for their memory, and they never want remote accesses or model executions on the critical path of predictions.

We have not needed pull-based caching however: as the two rightmost columns of Table 1 show, all the model sizes and feature datasets we have considered so far are small enough to fit entirely in the client's memory. Thus, we rely on push-based caching, where RC periodically produces new models and feature data for all subscriptions, and pushes them in the background to the caches in the client DLL. In the push approach, RC returns a no-prediction if it cannot find a model or feature data in its caches. For example, this may occur if the prediction request refers to a recently created subscription; information about this subscription will appear in the feature data cache after a future data push.

Regardless of the configured caching approach, RC also stores the content of the model and feature data caches in the local file system. It only looks up the local disk cache in either of two cases: (1) there is an in-memory model or feature data cache miss and the store is unavailable; or (2) the client crashes and restarts and the store is unavailable. In both cases, the DLL ignores the local disk cache, if it has expired (the expiration time is also configurable).

Justification. Our DLL-based design follows the principles listed above. First, our caches attain high hit rates and our models are light enough to be instantiated/run at the client (principle #1). Though high prediction performance is not required for some RC use-cases, it can be critical when models are large or expensive to execute, or when the client has a low time budget. Implementing RC as a service with a REST API would have placed the interconnect and the prediction-serving stack on the critical paths of all predictions. Second, the DLL is simple and is backed by an existing highly available store, so it does not require provisioning additional servers/VMs for serving predictions (principle #2). Finally, the interface the DLL provides makes requesting predictions seem like a simple method call (principle #3).

5 CASE STUDY: RC-INFORMED VM SCHEDULING

In this section, we describe how Azure's production VM scheduler can leverage the predictions from RC to enable careful physical CPU oversubscription. Other VM schedulers can use the same predictions in a similar way. Recall that a complete description of the Azure scheduler is beyond the scope of this paper. Instead, we overview the system and describe the changes we propose to it. We are productizing our changes and expect to deploy them to production in the next few months. In production, we will only oversubscribe servers running first-party workloads.

VM scheduler. Azure executes one VM scheduler per server cluster. The scheduler selects physical servers on which to place each VM assigned to its cluster. This decision is complex, because it must consider multiple resource dimensions, as well as several constraints on the VM placements, at the same time. For example, given a deployment defined by the VMs' maximum CPU, memory, and disk space requirements, the scheduler must find servers with enough resources, while spreading the VMs across multiple failure and update domains. The system must also reserve resources for deployment growth and healing (re-starting) VMs affected by hardware failures. Ultimately, the scheduler must maximize the number of VMs that can be accepted, while minimizing the number of VM scheduling failures.

Clearly, scheduling VMs fast and at scale under these conditions is challenging. In fact, computing an optimal solution for such a bin-packing problem is infeasible, so prior works have used heuristics [2, 3, 24]. In a similar vein, our scheduler sequentially applies a set of rules that progressively narrow the choice of servers that are candidates for a placement. Some rules represent hard constraints that cannot be violated (e.g., must have enough available resources), whereas others can be violated if they would excessively restrict the set of candidate servers (e.g., prefer servers that will be more tightly packed). Despite its heuristic nature, our multi-year experience with the scheduler shows that it can pack the VMs' requested resources tightly, producing little fragmentation and few scheduling failures.

Leveraging RC predictions for oversubscription. Our proposed changes seek to increase server utilization via CPU oversubscription, while preventing scenarios where servers would actually run out of physical CPUs.

Our new code appears in Algorithm 1, which includes a hard rule (SELECTCANDIDATESERVERS) and two bookkeeping functions (PLACEVM and VMCOMPLETED). The rule determines the servers where the new VM would "fit"; it passes those eligible servers to the next rule in the chain. Though our rule must check that the VM fits with respect to all resources, for clarity, the algorithm only shows the handling of the CPU resource (other resources are not oversubscribed). Similarly, we elide the single lock that protects the VM and server data structures in the three routines.

Our rule receives the existing production vs non-production annotation associated with first-party subscriptions as an input (called VM type in Algorithm 1). We only oversubscribe physical CPUs using non-production VMs. In fact, our code logically splits the servers into two groups: (1) oversubscribable, i.e. those that only host non-production workloads; and

Algorithm 1 CPU oversubscription rule and bookkeeping.

```
 1: Given: VM V to schedule, current set of candidate servers C
 2: Client inputs include V.subscription, V.type, V.alloc
 3: function SelectCandidateServers(V, C)
 4:     if V.type == PROD then
 5:         for each non-oversubscribable or empty server c ∈ C do
 6:             if c.alloc+V.alloc <= SERVER_CAPACITY then
 7:                 Mark c eligible
 8:     else
 9:         Pred = predict_single(VM_P95UTIL, client_inputs_list)
10:         if Pred.score >= .6 then
11:             V.util = Highest_Util_in_Bucket[Pred.value] * V.alloc
12:         else
13:             V.util = V.alloc
14:         for each oversubscribable or empty server c ∈ C do
15:             if c.alloc+V.alloc <= MAX_OVERSUB and
16:               c.util+V.util <= MAX_UTIL then
17:                 Mark c eligible
18:     return List of eligible servers
19: function PlaceVM(V, c)
20:     if c is empty then                    ▷ i.e., if c.alloc == 0 then
21:         if V.type == PROD then
22:             Tag c as non-oversubscribable
23:         else
24:             Tag c as oversubscribable
25:     c.alloc += V.alloc
26:     if c is oversubscribable then
27:         c.util += V.util
28: function VMCompleted(V, c)
29:     c.alloc -= V.alloc
30:     if c is oversubscribable then
31:         c.util -= V.util
```

(2) non-oversubscribable, i.e. those that only host production workloads. A later rule tries to fill up non-oversubscribable servers before it places VMs in empty servers. For oversubscribable servers, the rule avoids oversubscribing a server if another server can accommodate the new VM without oversubscribing resources.

Our rule uses RC predictions of 95^{th}-percentile virtual core utilization (line 9). If RC's confidence in this prediction is low, we conservatively assume that the VM will consume all its allocated cores (lines 10-13). The rule then marks as candidates any oversubscribable servers for which: the sum of the VMs' allocations is less than MAX_OVERSUB, and the sum of the VMs' predicted 95^{th}-percentile utilizations is less than MAX_UTIL (lines 15-17). MAX_OVERSUB is the maximum allowed CPU oversubscription (e.g., 15% more virtual cores than physical cores), and MAX_UTIL is the maximum allowed physical CPU utilization. We can set MAX_UTIL to 100% (or another high value that provides some slack) to prevent resource exhaustion.

Implementation as a soft rule. Implementing the utilization check as part of the (hard) "fit" rule as above gives more importance to preventing resource exhaustion than reducing scheduling failures. An approach that inverts this tradeoff implements the utilization check as a soft rule after the fit rule. In this version, the scheduler treats limiting the CPU utilization as best effort, i.e. if enforcing the soft rule would cause all servers that have the needed resources to be marked ineligible, the rule is simply disregarded.

Mispredictions and lack of predictions. A prediction error for a VM is unlikely to cause problems, unless RC underpredicts these values for many co-located VMs. Consistent under-predictions would cause the target peak utilization to be exceeded frequently. Utilization mispredictions would be more prone to problems if we decided to limit the combined predicted average utilizations, instead of the combined predicted 95^{th}-percentile utilizations. In case RC does not produce a prediction (not shown), it is safest to assume that the VM will exhibit 100% utilization.

Justification. As we seek to balance packing tightness and the ability to eliminate resource exhaustion in oversubscribed servers, our approach uses 95^{th}-percentile instead of maximum utilization predictions. Thus, resource exhaustion might occur when higher percentile utilizations for multiple non-production VMs happen to align in time, even when predictions are perfectly accurate. Our results show that these events happen very rarely, so we can limit resource exhaustion while achieving tight packing.

Finally, we do not consider network and disk bandwidth in VM scheduling, because Azure's interconnect is overprovisioned and persistent storage is remote to compute servers.

6 EVALUATION

We now evaluate the quality of RC's predictions, its performance, and the benefits of RC-informed VM scheduling.

6.1 RC prediction quality and performance

Prediction quality. As we suggest in Section 3, the customers' subscriptions are often a good unit upon which to base predictions, as VMs from the same subscription tend to exhibit similar behaviors. For this reason, we produce models using features that exist in all subscriptions, but produce predictions using feature data from the input VMs' subscription. This approach nicely matches the uses of RC for resource management, as we always know the subscription to which the VMs we want to predict belong.

To evaluate RC's prediction quality, we train our models with two months of data and test them on the third month of our dataset (Section 3). To improve quality, we experimented with multiple techniques, including feature engineering and normalization, feature selection and regularization.

We divide the space of possible predictions for each metric into the buckets listed in Table 3. Given these buckets, Table 4 summarizes our prediction results for every metric and

Metric	Bucket 1	Bucket 2	Bucket 3	Bucket 4
Avg and P95 util	0-25%	25-50%	50-75%	75-100%
Deployment size (#VMs and #cores)	1	>1 & ≤10	>10 & ≤100	>100
Lifetime	≤15 mins	>15 & ≤60 mins	>1 & ≤24 hs	>24 hs
Workload class	Delay-insensitive	Interactive	NA	NA

Table 3: Metrics and their bucket sizes.

bucket. The leftmost column lists the metric we are predicting. The second column lists the accuracy of the full set of predictions (i.e., the percentage of predictions that were correct), assuming the predicted bucket is that with the highest confidence score. The next set of columns present the results per bucket: they list the percentage of VMs that truly fall in the bucket, the predictions' *precision* for the bucket (i.e., the percentage of true positives in the set of predictions that named this bucket), and the predictions' *recall* (i.e., the percentage of true positives in the set of predictions that should have named this bucket). The final two columns list the precision and recall, respectively, assuming that RC replies with a no-prediction if the highest bucket score is lower than 0.6 (i.e., confidence is low).

Table 4 demonstrates RC's high prediction accuracy, which ranges between 79% (lifetime) and 90% (workload class) depending on the metric. The prediction quality is even higher when we discard predictions with low confidence: precision ranges between 85% (lifetime) and 94% (95^{th}-percentile CPU utilization) without substantially hurting recall, which ranges between 73% (95^{th}-percentile CPU utilization) and 98% (workload class). For all metrics, the most important attributes in determining prediction accuracy are the percentage of VMs classified into each bucket to date in the subscription. Other relevant attributes are the service name (i.e., the name of a top first-party subscription or "unknown" for the others), deployment time, operating system, and VM size; their relative importance depends on the metric. VM roles have little predictive value, e.g. all IaaS VMs have role "IaaS" but may behave quite differently.

RC tends to achieve higher precision and recall for the more popular buckets. This is expected since it has more data to learn from in these cases. An interesting case is workload class: bucket 1 shows that 99% of the VMs run delay-insensitive workloads, and RC achieves 100% precision (every delay-insensitive prediction is for a delay-insensitive VM) and 90% recall (90% of the delay-insensitive VMs are predicted as such). Bucket 2 shows that RC achieves 7% precision and 84% recall for the relatively few interactive VMs. This low precision reflects our desire to maximize the recall for this class, and means that many interactive predictions are for delay-insensitive VMs; mistakes in this direction are acceptable whereas those in the reverse direction (predicting interactive VMs as delay-insensitive) are undesirable.

Metric	Acc.	Bucket 1			Bucket 2			Bucket 3			Bucket 4			P^θ	R^θ
		%	*P*	*R*	%	*P*	*R*	%	*P*	*R*	%	*P*	*R*		
Avg CPU utilization	0.81	74%	0.92	0.83	19%	0.73	0.78	6%	0.78	0.74	2%	0.74	0.71	0.87	0.78
P95 CPU utilization	0.83	25%	0.83	0.89	15%	0.72	0.74	14%	0.73	0.72	46%	0.91	0.86	0.94	0.73
Deploy size (#VMs)	0.83	49%	0.85	0.87	40%	0.79	0.79	10%	0.80	0.74	1%	0.87	0.75	0.88	0.90
Deploy size (#cores)	0.86	19%	0.81	0.84	60%	0.88	0.90	19%	0.81	0.76	3%	0.86	0.66	0.90	0.92
Lifetime	0.79	29%	0.78	0.81	32%	0.70	0.79	32%	0.71	0.77	7%	0.69	0.80	0.85	0.80
Workload class	0.90	99%	1.00	0.90	1%	0.07	0.84	NA	NA	NA	NA	NA	NA	0.91	0.98

Table 4: RC's prediction quality. Acc = accuracy; P = precision; R = recall.

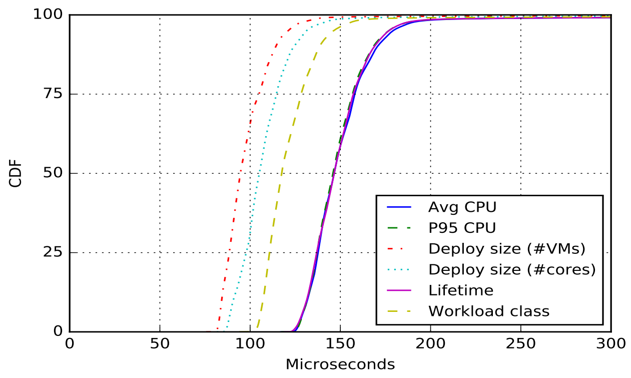

Figure 10: Latency of model execution for our metrics.

Performance. The key drivers of RC's online performance are the client DLL caches, model execution, and the data store, so our experiments assess: (1) the hit rate and latency of result cache accesses; (2) the latency of model execution on the client; and (3) the latency of model and feature data cache misses when using pull-based caching. For these experiments, we create a dummy client that repeatedly requests predictions for the VMs in our month-long test set. The client and DLL run on a 2-core VM.

We find that the RC result cache achieves a high hit rate for our dataset. On average, an entry is accessed between 18 and 68 times, depending on the metric, after the corresponding model execution. Since the result cache is very small (at most around 25 MB) for our dataset and could be allowed to grow significantly, a longer VM trace would produce an even higher average number of hits per model execution. The 99^{th}-percentile latency of a hit is only 1.3 μs, essentially corresponding to preparing a key and accessing a hash table.

Figure 10 shows that latencies are low (and predictable) even when RC executes models on-the-fly due to result cache misses. Median latencies range from 95 to 147 μs, whereas 99^{th}-percentile latencies range from 139 to 258 μs.

Finally, we find that accesses to the data store when using pull-based caching take substantially longer. Specifically, the store we use has median and 99^{th}-percentile latencies of 2.9 and 5.6 *ms*, respectively, for an 850-byte record (the size of the feature data for each subscription).

6.2 Case study: RC-informed VM scheduling

We now evaluate the impact of using 95^{th}-percentile CPU utilization predictions in server CPU oversubscription. Our goal is *not* to fully evaluate the Azure VM scheduler. Rather, we assess the potential benefits of RC's resource utilization predictions, using the Azure scheduler as an example.

Methodology. As our changes to the VM scheduler are not yet in production, we rely on simulations of it using real VM traces. The simulator is faithful to the real scheduler. In fact, Azure evaluates all changes to the production scheduler using the simulator. We have extended the simulator to (1) associate real first-party utilization traces (from the dataset of Section 3) with the VMs being scheduled, (2) implement our changes to the scheduler, (3) consume RC predictions for the real VMs, and (4) aggregate CPU utilization data for all simulated servers. We aggregate utilization data in the simulator by adding up the co-located VMs' *maximum* utilizations in each 5-minute period. This is pessimistic as it assumes that the maximum utilizations last for the entire period. Given this approach, *simulated server utilizations may actually exceed 100%*. We assume that each VM's virtual core is pinned to a physical core, so exceeding 100% utilization in a period means that more than one virtual core would have shared a physical core (via timeslicing) for part of the period.

The simulator reports the server CPU utilizations over time, and the number of scheduling "failures" (VMs that could not be scheduled due to lack of resources or fragmentation). Using these metrics, we compare schedules without oversubscription or differentiation between production and non-production VMs (*Baseline*), with RC-informed oversubscription as a soft rule (*RC-informed-soft*), with RC-informed oversubscription as a hard rule (*RC-informed-hard*), and with oversubscription but without RC's predictions (*Naive*). We also compare against schedules in which RC always predicts the correct bucket (*RC-soft-right*) or always predicts an incorrect random bucket (*RC-soft-wrong*) for the 95^{th}-percentile CPU utilization. By default, we set MAX_OVERSUB and MAX_UTIL in Algorithm 1 to 125% and 100% of the servers' CPU capacity, respectively, but we also study the sensitivity of the RC results to these parameters. Finally, we simulate

336k VM arrivals to a cluster of 880 servers (each with 16 cores and 112 GBytes of RAM) over a period of 1 month. The VMs exhibit the same size and lifetime distributions as in Section 3, and their original tags (71% production VMs).

Comparing schedulers. Given our default settings, RC-informed-soft produces no scheduling failures and only 77 individual server utilization readings above 100% over the entire month and across all servers. RC-informed-hard produces the same results, as VM utilizations are not high enough to cause deployment failures (our utilization analysis below describes cases in which RC-informed-soft and RC-informed-hard behave differently). Naive does not perform as well: it also produces no failures, but shows 6× more server utilizations higher than 100%. In contrast, the Baseline system produces no server utilizations higher than 100% (since there is no oversubscription) and 0.25% of failures. This percentage of failures is 2.5× higher than what we consider acceptable. These results show that RC-informed oversubscription produces more capacity from the same hardware, while controlling resource exhaustion for non-production workloads.

We also find that predicting utilization accurately is important. RC-soft-right produces similar behavior to RC-informed-soft, with some more utilization readings higher than 100%. This is not surprising since (1) RC-informed-soft already has high prediction accuracy (Table 4), and (2) RC-soft-right assumes perfect predictions of 95^{th}-percentile utilization, whereas the simulator aggregates maximum utilizations. In contrast, RC-soft-wrong does much worse, producing 3× more above-100% utilization readings than RC-informed-soft (while still producing no scheduling failures).

Sensitivity to amount of oversubscription (MAX_OVERSUB). We now compare RC-informed-soft for allowable oversubscriptions of 125% (results above), 120%, and 115% of the servers' CPU capacity. Lowering this value progressively increases the number of scheduling failures, as there is less available capacity for non-production workloads. Even at 115% oversubscription however, the number of failures in RC-informed is still 65% lower than in Baseline. At the same time, each server receives fewer VMs as oversubscription decreases, so the likelihood of multiple concurrent utilization spikes decreases. This leads to even lower numbers of utilization readings above 100%: at 115% allowable oversubscription, RC-informed-soft produces only 22 such readings.

Sensitivity to target max server utilization (MAX_UTIL). Now, we investigate the impact of lowering the target maximum server utilization from 100% (results above) to 90% or 80%, while keeping the amount of allowable oversubscription at 125%. Reducing the target effectively reduces the available server capacity for non-production workloads, leading to a significant increase in scheduling failures for the same VM arrivals. For example, for a target of 80%, the number of scheduling failures increases to 903 or 0.27%, i.e. well beyond the acceptable 0.1% failures. Nevertheless, lower target maximum utilizations can be used in more lightly loaded systems. For instance, with 20% less load, an 80% target maximum utilization leads to no failures.

Sensitivity to VM resource utilization. Finally, we study the impact of higher virtual core utilization by artificially adding 25% to all real utilization values, and adding 1 to all utilization bucket predictions. As expected, higher utilization causes RC-informed-hard to produce more scheduling failures than RC-informed-soft. However, the difference is just 4 failures, since our utilization predictions must be above 100% for *all* servers with enough available resources for the hard rule to cause an extra failure.

7 CONCLUSION

In this paper, we detailed Azure's VM workload, and discussed how the workload's characteristics can be exploited in improving resource management. We then introduced Resource Central, a system for generating, storing, and efficiently using predictions of these characteristics. Finally, we described changes to Azure's production VM scheduler that leverage such predictions. Our results show that Resource Central performs well, produces accurate predictions, and enables safe CPU oversubscription. We conclude that providers can exploit the characteristics of their workloads in many management tasks, using machine learning and efficient prediction-serving systems.

Acknowledgements

We would like to thank our shepherd, Mike Dahlin, and the anonymous reviewers for their extensive comments on our work. Christina Delimitrou, Inigo Goiri, Alok Kumbhare, Knut Risvik, and Yunqi Zhang also provided many comments that helped us improve our paper. Finally, we thank Bertus Greeff and Abhisek Pan for their help with the VM scheduler and its simulator.

REFERENCES

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating System Design and Implementation* (2016).

[2] Beloglazov, A., and Buyya, R. Energy Efficient Resource Management in Virtualized Cloud Data Centers. In *Proceedings of the 10th International Conference on Cluster, Cloud and Grid Computing* (2010).

[3] Bobroff, N., Kochut, A., and Beaty, K. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *Proceedings of the International Symposium on Integrated Network Management* (2007).

[4] Calheiros, R. N., Masoumi, E., Ranjan, R., and Buyya, R. Workload Prediction Using ARIMA Model and its Impact on Cloud Applications' QoS. *IEEE Transactions on Cloud Computing 3*, 4 (2015).

[5] Crankshaw, D., Bailis, P., Gonzalez, J. E., Li, H., Zhang, Z., Franklin, M. J., Ghodsi, A., and Jordan, M. I. The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research* (2015).

[6] Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation* (2017).

[7] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014).

[8] Di, S., Kondo, D., and Cirne, W. Characterization and Comparison of Cloud versus Grid Workloads. In *Proceedings of the 2012 International Conference on Cluster Computing* (2012).

[9] Gong, Z., Gu, X., and Wilkes, J. Press: Predictive Elastic Resource Scaling for Cloud Systems. In *Proceedings of the International Conference on Network and Service Management* (2010).

[10] Google. TensorFlow Serving. http://tensorflow.github.io/serving/.

[11] He, K., Fisher, A., Wang, L., Gember, A., Akella, A., and Ristenpart, T. Next Stop, the Cloud: Understanding Modern Web Service Deployment in EC2 and Azure. In *Proceedings of the 2013 Internet Measurement Conference* (2013).

[12] Islam, S., Keung, J., Lee, K., and Liu, A. Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud. *Future Generation Computer Systems 28*, 1 (2012).

[13] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd International Conference on Multimedia* (2014).

[14] Khan, A., Yan, X., Tao, S., and Anerousis, N. Workload Characterization and Prediction in the Cloud: A Multiple Time Series Approach. In *Proceedings of the International Conference on Network and Service Management* (2012).

[15] Kilcioglu, C., Rao, J., Kannan, A., and McAfee, R. P. Usage Patterns and the Economics of the Public Cloud. In *Proceedings of the 26th International World Wide Web Conference* (2017).

[16] Li, A., Yang, X., Kandula, S., and Zhang, M. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (2010).

[17] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015).

[18] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. MLLib: Machine Learning in Apache Spark. *Journal of Machine Learning Research 17*, 34 (2016).

[19] Mishra, A. K., Hellerstein, J. L., Cirne, W., and Das, C. R. Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters. *SIGMETRICS Perform. Eval. Rev. 37*, 4 (Mar. 2010).

[20] Novakovic, D., Vasic, N., Novakovic, S., Kostic, D., and Bianchini, R. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the USENIX Annual Technical Conference* (2013).

[21] Padala, P., Hou, K.-Y., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., and Merchant, A. Automated Control of Multiple Virtualized Resources. In *Proceedings of the 4th European Conference on Computer systems* (2009).

[22] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the 3rd Symposium on Cloud Computing* (2012).

[23] Roy, N., Dubey, A., and Gokhale, A. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *Proceedings of the International Conference on Cloud Computing* (2011).

[24] Verma, A., Ahuja, P., and Neogi, A. pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems. In *Proceedings of the International Conference on Distributed Systems Platforms and Open Distributed Processing* (2008).

[25] Wilkes, J. More Google Cluster Data. Google research blog, Nov. 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[26] Wu, Q., Deng, Q., Ganesh, L., Hsu, C.-H., Jin, Y., Kumar, S., Li, B., Meza, J., and Song, Y. J. Dynamo: Facebook's Data Center-Wide Power Management System. In *Proceedings of the International Symposium on Computer Architecture* (2016).

[27] Yang, H., Breslow, A., Mars, J., and Tang, L. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013).