Generating Complex, Realistic Cloud Workloads using Recurrent Neural Networks

Shane Bergsma
Timothy Zeyl
Huawei Research, Canada

Arik Senderovich
J. Christopher Beck
University of Toronto

Abstract

Decision-making in large-scale compute clouds relies on accurate workload modeling. Unfortunately, prior models have proven insufficient in capturing the complex correlations in real cloud workloads. We introduce the first model of large-scale cloud workloads that captures long-range inter-job correlations in arrival rates, resource requirements, and lifetimes. Our approach models workload as a three-stage generative process, with separate models for: (1) the number of batch arrivals over time, (2) the sequence of requested resources, and (3) the sequence of lifetimes. Our lifetime model is a novel extension of recent work in neural survival prediction. It represents and exploits inter-job correlations using a recurrent neural network. We validate our approach by showing it is able to accurately generate the production virtual machine workload of two real-world cloud providers.

Keywords: cloud workload modeling, trace generation, recurrent neural networks, deep learning, survival analysis

ACM Reference Format:
Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J. Christopher Beck. 2021. Generating Complex, Realistic Cloud Workloads using Recurrent Neural Networks. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483590

1 Introduction

Many decisions must be made in the operation of large-scale compute clouds. Scheduling systems must decide how to route workload requests to different physical resources (e.g., servers). Capacity planners must decide which resources to provision, and when, in order to support expected future workloads. Better scheduling and planning decisions can improve resource utilization and reduce costs. Given both the enormous scale of resources [31], and the surprisingly low levels of utilization currently observed in cloud data centers [46, 58], there is tremendous potential for improvement.

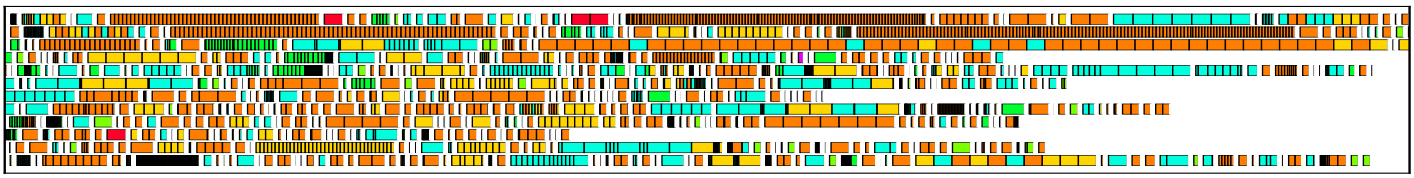
Accurate models of cloud workload can improve decision-making in a number of ways [9]. For example, models can provide estimates of future workload to the scheduler; the scheduler may then pro-actively take steps to reduce future resource contention and fragmentation [17, 46]. In this paper, we model the total job *demand* over time: the job arrival times, lifetimes, and resource requirements (e.g., requested CPU/memory). We do not address the topic of modeling a job's actual CPU or memory *usage* once running in the cloud (also known as "cloud workload prediction" [55, 69, 72]).

We propose a *generative* model of total job demand, capable of generating synthetic workload traces. Both historical and synthetic data are used to evaluate production cloud systems, such as VM schedulers [31]. Unfortunately, historical data is limited in size, and, because of so-called "workload churn" [5], quickly becomes obsolete. Meanwhile, synthetic data is unlimited, and can be be designed to reflect possible *future* workload dynamics. We can use it to optimize our decision-making processes, in advance. We can also adjust model parameters to simulate various conditions of interest (e.g., scaling up arrivals for scheduler stress-testing). Furthermore, by repeatedly sampling traces from our model, we can obtain a probability distribution over different future workload scenarios. We may then incorporate this information into downstream decision-making processes (e.g., we can assess whether we have enough servers to cover 95% of possible workload scenarios over the next month). Generative models of cloud workload can also provide the quantity and variety of input needed for training cloud systems based on deep reinforcement learning [49, 50, 52].

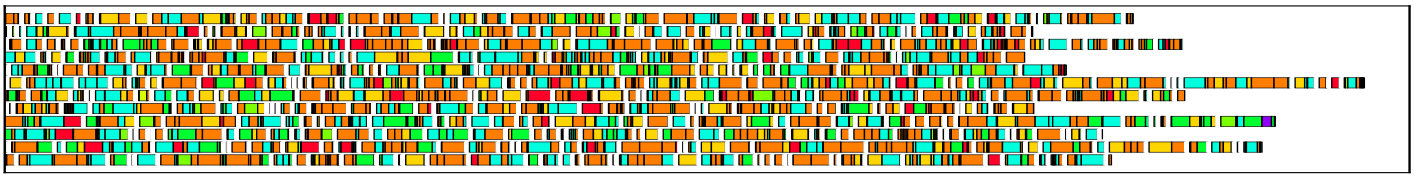
Unfortunately, "workload modeling is surprisingly hard to do well" [68]. Cloud workloads are commonly referred to as "heterogeneous" [57] and "imbalanced" [46], both spatially (in resources), and temporally (in lifetimes). Verma et al. [68] point to the variety of job types and shapes, as well as the complex inter-job dependencies, as factors that make cloud workloads unsuitable to what they call "naive" modeling techniques. And while there is much prior work analyzing real workload traces [17, 19, 46, 57], practitioners lack a means to generate the full workload process (including job arrivals, resources, and lifetimes) that reflects these real-world complexities. In practice, particularly in the development of cloud

SOSP '21, October 26–29, 2021, Virtual Event, Germany

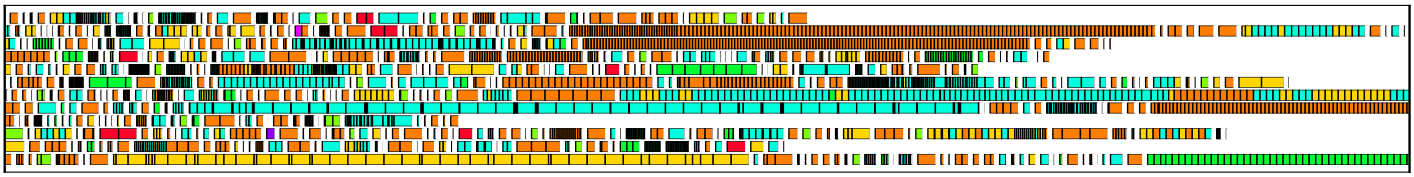
ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483590



(a) Real trace from Microsoft Azure



(b) Workload generated by naive approach



(c) Workload generated by our approach

Figure 1. Workload/model visualization: Each rectangle represents a VM, each row shows all VM arrivals in one 5-minute period. Color represents VM *flavor*, width represents lifetime (compressed non-linearly to discrete bins; bin indices are used as width). Rows are not timelines showing when VMs start/stop, but rather depict resource and lifetime properties of VMs arriving sequentially in a period. In real workloads (and in those from our generator), VMs arrive in user-specific batches (space-separated here), within each batch VMs have similar properties, and arrival rates vary a lot from period to period.

workload schedulers [34], job requests are assumed to be submitted independently, with a Poisson process for sampling arrivals (e.g., [12, 21, 26, 47, 48]), and simple distributions for sampling job type and lifetime (e.g., [12, 17, 21, 54]). Here, we use the term "naive" to specifically refer to models that ignore statistical inter-job correlations.

We propose a generative model with enough expressive power to represent complex inter-job relationships, and evaluate this model on the production virtual machine (VM) workload of two large-scale cloud providers: (1) Microsoft Azure and (2) Huawei Cloud. These workloads exhibit both the heterogeneity and complex correlations documented in other traces. Figure 1 provides a visualization of the Azure VM data [17]. We observe great variation in the arrival rate, lifetime, and flavor (specific resource requirement) of VMs. But these features of the workload are not random: VMs of the same flavor and similar lifetime tend to arrive consecutively. There is noticeable *momentum* — statistical dependence — in properties of VMs in the arrival sequence.

We propose the first model of large-scale cloud workloads that captures long-range inter-job correlations in job arrival rates, resource requirements, and lifetimes. Our generative process consists of three stages, each building on the output of the previous stage. First, we model the arrival of user-specific *batches* of work using Poisson regression (Section 2.1). Next, we use a long-short-term memory neural network (LSTM) (a type of recurrent neural network), to effectively model the sequence of requested resources (Section 2.2). Finally, we use a separate LSTM to model the discrete-time hazard rate for job lifetimes (Section 2.3).

Our lifetime LSTM is a key contribution of this work. It extends prior work in neural survival prediction [24, 39] by modeling inter-case relationships, allowing the predicted lifetime distribution of an individual (job) to depend on the lifetimes of all individuals that arrived in the past. To enable such a model, we enhance both the input and output representations of our network. In particular, we enhance the network to handle *censoring* (dependence on incomplete lifetime information), an important practical consideration.

A good generative model should not only generate qualitatively realistic data, it should assign high probability to new data and be useful in applications. We show that compared to standard approaches, our model computes higher probability of real production workloads (Section 5). We also show that end-to-end traces from our model are beneficial in two cloud applications: (1) capacity planning (Section 6.1) and (2) workload scheduling (Section 6.2).[1]

[1]Code is available at https://github.com/huaweicloud/trace_generation_rnn

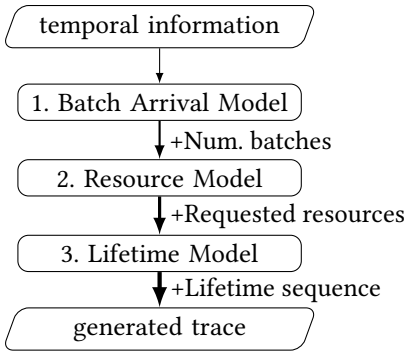


Figure 2. Three-stage workload generation process for one period. Temporal information includes a period's hour-of-day, day-of-week, etc. Each stage builds on information from previous stages. The generated trace outputs start times, end times, and resource requirements, for all jobs in each period.

2 Methods

Our workload model is a generative model, trained from a trace of historical jobs, where the trace indicates each job's:

1. Start time
2. End time (possibly empty, if the job is still running)
3. Requested resources (e.g., CPU, memory, network, etc.)

The trace should reflect raw workload demand, without delays due to limited server resources; for example, VM traces typically reflect raw demand, since cloud providers ensure sufficient capacity such that requested VMs are served with minimal delay. As an input to help organize the generative process, we also assume that each job in the data has a user ID. However, we do not generate specific IDs in our output.

Our workload model generates the data as visualized in Figure 1. We generate all jobs for one specific *period*, then generate for the next period, and so on. Each period is a fixed time interval (e.g. 5 minutes), corresponding to a row in Figure 1. Within each period, we generate all jobs from one user first, then from the next user, and so on. We refer to the sets of jobs from the same user and within the same period as *batches*. In the figure, each batch corresponds to a sequence of horizontally-contiguous rectangles, and different batches within the same period are separated by whitespace. Within a batch, jobs are generated in order of their arrival time. Batches themselves are generated in order of the arrival time of their first job. When we train our generative model, we organize and process the data in the same manner.

Within each period, our generative process consists of three stages, illustrated in Figure 2. Each stage relies on the output of the previous stage: we generate the number of batches in each period given the period's temporal information (Section 2.1). We generate the number of jobs and requested resources, given the number of batches (Section 2.2). Then we generate job lifetimes, given the batches of requested resources (Section 2.3). The generated data is then converted to job start/end times, plus resource requirements, and we output a trace (Section 2.4).

2.1 Batch Arrival Modeling

The arrival model provides a distribution over the number of batches in each time period (where batch is defined above).

2.1.1 Model. We fit the number of batch arrivals in each 5-minute period using Poisson regression. An (inhomogeneous) Poisson regression assumes the number of events N_p occurring in each time period, p, is modeled by a Poisson distribution with mean μ_p. Rather than being a constant, Poisson regression assumes this rate varies with some features of the period, according to $\mu_p = \exp(\mathbf{w} \cdot \mathbf{x_p})$, where $\mathbf{x_p}$ is a feature vector, and $\mathbf{w}$ is a vector of learned parameters.

Given some training examples consisting of a vector of event counts, $\mathbf{y}$, and a matrix of feature vectors, $\mathbf{X}$, we fit the parameters by minimizing the negative-log-likelihood (NLL) of the training data:

$$\begin{aligned} Loss &= -\log(\Pr(\mathbf{y}|\mathbf{X}, \mathbf{w})) \\ &= \sum_p \mu_p - y_p \log(\mu_p) \end{aligned}$$

We add an elastic net regularization term [75] to this loss function (with the penalty weight tuned on development data), and minimize the regularized NLL.

2.1.2 Features. Arrival features encode coarse-granularity temporal information about the period for which we are generating batches. We have three types of temporal features:

1. Hour-of-day (*HOD*), from 1 to 24 (one-hot-encoded)
2. Day-of-week (*DOW*), from 1 to 7 (one-hot-encoded)
3. Day-of-history (*DOH*), from 1 to N (survival-encoded), where there are N total days in the history

A one-hot-encoding represents the nth-of-N-features using a length-N vector of all zeros, except for the nth element, which is equal to 1. A survival-encoding represents the nth-of-N-features using a length-N vector such that all elements $\leq n$ are 1, and all elements $> n$ are zero. The *HOD* and *DOW* features capture the *seasonality* inherent in cloud workloads (typically fewer arrivals on nights and weekends). The *DOH* feature captures both the workload *trend*, as well as any *change-points* (a common attribute of real time series [64]).

When applying our model to a test period beyond our training window, we explore two options for setting the day used for the *DOH* feature: (1) encoding the last day of the history (day N) or (2) sampling a day from $1 \ldots N$. For sampling, we assume that more recent days (i.e., N, N−1, etc.) are more likely to reflect future behavior. We therefore sample a day k-days-before-N (i.e., $N-k$) using a geometric distribution for k, with success probability tuned on development data. Sampling the day for the *DOH* feature is one mechanism to mitigate the effects of forecasting in the presence of workload churn [5]. In the cloud, every day is unique, but with

DOH features, we can sample traces where the future varies in a manner similar to the past.

2.2 Resource Modeling

The resource model predicts the requested resources for all the jobs in one period. VM workloads are usually drawn from a discrete set of *flavors*, where each flavor represents a distinct bundle of resources. In HPC [19], jobs may request arbitrary combinations of resources, while in DAG-of-task workloads [57], jobs may be composed of multiple *phases*, where each phase is composed of a homogeneous set of short-lived tasks. We focus on VM flavors below in this paper, but briefly discuss possible extensions in Section 2.2.3.

2.2.1 Flavor Sequence Model. We model sequences of flavors using a recurrent neural network, specifically an LSTM [32], an established tool for generating "complex, realistic sequences containing long-range structure" [30]. In an RNN, at step t the network takes as input a feature vector $\mathbf{x}_t$ and generates as output a vector of scores, $\mathbf{y}_t = (y_t^1, y_t^2, \ldots)$. For flavors, these scores parameterize a distribution over the next flavor in the sequence (see below). The inputs to an RNN pass through a stack of connected hidden layers, parameterized by weight matrices, that compute the outputs at each step. As we iterate through subsequent steps, hidden layer values are updated from both new input features, and from the values of hidden layers at preceding steps in the network (allowing *internal state* to flow through the network). An LSTM is an enhanced RNN architecture that is better able to manage state over arbitrary step intervals through the use of special *gated* cells in the hidden layers.

At each step, t, the flavor LSTM returns a distribution over K possible flavors, plus a special end-of-batch (*EOB*) token to indicate the conclusion of one user's set of job requests for a period. That is, the LSTM generates K+1 output scores, which are input to a softmax function returning a multinomial distribution over the K+1 options for flavor $\hat{f}$ at step t:

$$\Pr(\hat{f}_t = k|\mathbf{y}_t) = \frac{\exp(y_t^k)}{\sum_{k'=1}^{K+1} \exp(y_t^{k'})}$$

When training, the true (observed) previous flavors are encoded as input for the next step, following [30].

To train the parameters of the network, we follow the standard approach in minimizing the negative-log-likelihood of training data [30]. The probability of an observed sequence of flavors of length T, $f_1, \ldots, f_T$ is the product of probabilities of each flavor in the sequence. Our loss is the negative-log of this likelihood (plus a regularization penalty). We compute the gradients of the loss with respect to our network parameters through backpropagation.

2.2.2 Flavor Sequence Features. Our primary flavor feature at step t is a one-hot-encoding of the flavor at the previous step, $t-1$. If the previous step was the end of a batch, or we are generating the first flavor in the sequence, we encode the *EOB* token as the input. The network can learn that after an *EOB* token, there is little correlation with the flavors occurring earlier in the sequence.

We also use the temporal features from Section 2.1.2. These features remain constant throughout one period. Temporal features allow the network to capture any time-varying patterns in the popularity of different flavors.

2.2.3 Beyond Flavors. To model jobs with arbitrary combinations of resources, we can use a different LSTM output layer. One approach is to replace the softmax with a mixture density, as was done for handwriting synthesis [30]. Another approach is to discretize the possible values in each resource dimension, as was done for RGB channels of image pixels [66]. Such a resource LSTM could have a softmax for generating CPU, then a separate softmax for generating memory (conditioned on the generated CPU), etc.

For generating DAG-of-task workloads, the key question is whether to model the within-job structure using a neural network. One option is to cluster jobs into distinct "pseudo-flavors" and use the flavor LSTM to generate a sequence of pseudo-flavors. Specific workloads can be generated by sampling an historical job for each generated pseudo-flavor, similar to how MapReduce workloads are synthesized in [14]. Another option is to have the LSTM generate the phase boundaries, tasks-per-phase, task resource requirements, etc. The proper approach will depend on the amount of training data and the level of detail desired in the generated traces.

For DAG-of-task workloads, the lifetime model (below) need not change. It can generate lifetimes for the sequence of *tasks*, rather than for the sequence of jobs.

2.3 Lifetime Modeling

The lifetime model returns a distribution over the possible lifetimes of every job in the resource sequence. We propose to again use an LSTM, but in this case, the outputs will parameterize the *hazard function* over a discrete set of lifetime bins. We first describe considerations for discrete lifetime estimation in general, and then the details of our network.

2.3.1 Discrete lifetime estimation. In discrete lifetime estimation, possible lifetimes are divided into discrete bins, $b_1, \ldots b_J$, representing J consecutive intervals of time. Let j indicate the bin index. There are three related statistical functions over these bins (each derivable from each other):

The probability mass function (PMF) $f(j)$ gives the probability that the lifetime falls into the bin j

The survival function $S(j)$ gives the probability that the lifetime falls into *any* bin i where $i > j$

The hazard function $h(j)$ gives the probability that the lifetime falls into the bin j, given that the lifetime did *not* fall into any bin i where $i < j$

Kvamme and Borgan [39] show that for feed-forward neural survival models, parameterizing the hazard function works "slightly better" than parameterizing the PMF.

We can estimate these functions using historical data. The main complication that arises is *right censoring*: how to handle jobs that have not yet terminated by the end of the observation window. We could discard all such jobs, but then our estimates will be biased (although this is common in systems work, e.g., Cortez et al. [17] analyze "only VMs that started and completed in [the] observation period"). In the field of survival analysis, it is common to instead use the Kaplan-Meier (KM) estimator [35]. KM is a non-parametric estimator that captures the empirical distribution of lifetimes in historical data through the hazard function, but only counting hazard events for bins where we observe either a survival or a termination (and ignoring bins beyond the observation window). KM is preferred over parametric estimators unless the sample size is very small [51]. KM inspires both the features and loss function of our lifetime RNN model (below).

To determine the bin boundaries, Kvamme and Borgan [39] propose setting boundaries at evenly-spaced quantiles of lifetimes in training data. We found this approach resulted in very coarse bins for the longest-lifetimes, so we instead took the approach of binning 5-minute intervals up to 1-hour, 1-hour intervals up to 10-hours, daily intervals up to 10 days, and a final bin boundary for greater than 20 days.

2.3.2 Model. Our model is an LSTM that, at each step, t, takes as input a feature vector, $\mathbf{x}_t$, and generates as output a vector of J scores, $\mathbf{y}_t = (y_t^1, \ldots, y_t^J)$, one for each of a job's possible lifetime bins. Unlike the flavor LSTM that uses the scores as logits in a softmax, here each score is input to a *logistic* function that maps the score into a hazard probability:

$$h(j|\mathbf{x}_t) = \frac{1}{1 + \exp(y_t^j)}$$

To train the parameters of the network, we minimize the negative-log-likelihood of training data. The likelihood of an observed sequence of job lifetimes is the product of probabilities of the lifetimes of each job (i.e., at each step).

Let us consider the probability, at one step, of the lifetime occurring in an observed bin k given a single set of outputs, $h(j)$, $j = 1 \ldots J$ (dropping the implicit dependence on $\mathbf{x}_t$ for convenience). This outcome requires avoiding the hazard for the first $k-1$ bins and suffering the hazard in bin k:

$$\Pr(\text{lifetime} = k) = h(k) \prod_{j=1}^{k-1} (1 - h(j))$$

Now consider when a job's lifetime is in bin c, but the job has not yet terminated (i.e, the lifetime is right-censored). In this case, we still get credit for surviving (avoiding the hazard) up until bin c (similar to the Kaplan-Meier estimator):

$$\Pr(\text{censored at } c) = \prod_{j=1}^{c-1} (1 - h(j))$$

Unlike RNNs that parameterize multinomial distributions, for our network, at every step of every sequence (and whether lifetimes are censored or not), there may be many network outputs that do not factor into the loss calculation.

For those outputs that factor into the loss calculation, the negative-log-likelihood of a dataset is also the binary cross entropy (BCE). We add a regularization penalty to the BCE loss, and compute the gradients with respect to our network parameters through backpropagation.

2.3.3 Features. At each step, we encode a variety of information that may influence the job's lifetime distribution:

- Temporal features (Section 2.1.2)
- Requested resources of current job (Section 2.2.2)
- Number of jobs in current batch
- Lifetime of *previous* job

To encode the previous job's lifetime, we use a survival-encoding of the previous job's lifetime bin (Section 2.1.2). To handle cases where preceding jobs have not terminated by the end of the observation window (i.e., lifetime is right-censored), we use another set of features: we have one unique feature for every discrete lifetime bin, but for these, we encode a value of 1 for all bins where the job is known to be *terminated*. If a job is censored, these features are all zero.

Censoring is important because real-world data is *always* right-censored: some jobs will still be running at the end of any observation window. Censoring is especially pervasive in VM workloads because VMs, for a variety of reasons, run much longer than other workloads [17]. We could continue collecting lifetimes after the end of the observation window, but the unfortunate trade-off is that as we wait for jobs to finish, data can grow obsolete. Accounting for censoring allow us to train on very recent data, while deriving maximum information from the lifetimes in that data.

2.4 Generating Traces

We use our trained model to generate traces for new periods by running our three-stage generative process (Figure 2): given the period's temporal features, we estimate the arrival rate for that period, μ_p, using our batch arrival model. We sample the number of batches, n_p, from the resulting Poisson distribution. Next, we use the resource LSTM to generate a sequence of requested resources for the period. We do this one-step-at-a-time, iteratively re-encoding the generated resources of the previous job as part of the input for the next step. As we generate resources, we track how many *EOB* tokens have been generated, and terminate the sequence after exactly n_p batches. Finally, we run the lifetime LSTM over the input resources to generate the hazard functions.

We sequentially sample a lifetime bin from each job's hazard function, and iteratively re-encode that lifetime as part of the input for the hazard prediction for the next job.[2]

We then convert our jobs and lifetimes to job start/end times (plus resource requirements) in order to create the output trace. To convert our discretized lifetime bins to real-valued durations, we use the continuous-density interpolation (CDI) method, which was shown to perform well in [39]. CDI assumes job terminations are distributed evenly within each lifetime bin. To get precise intra-period start times, we must also generate the inter-arrival times of jobs in each batch. Recall that job start/end times in our data have been quantized to 5-minute intervals (Section 3), so all jobs in one period effectively start at the period timestamp. For capacity planning (Section 6.1), where a "relatively small percentage of long-running VMs actually account for >95% of the total core hours" [17], modeling time with higher precision is unnecessary. For applications requiring specific orderings of arrivals and departures, such as scheduling (Section 6.2), we distribute the arrivals across the 5-minute period in their generative order (Section 2); the departures are randomly distributed within each period and interleave with arrivals.

By repeatedly sampling traces that cover the same time period, we can compute probability distributions over the projected workload. These distributions enable confidence intervals on various dimensions of generated data. For example, for capacity planning (Section 6.1), we use distributions over the total number of CPUs in a future time period.

For other applications, such as workload scheduling (Section 6.2), we may instead tune on a collection of generated traces (for the same period), ensuring our systems are optimized for a variety of possible future workloads.

3 Data

We evaluate on the VM workload of: (1) Microsoft Azure and (2) Huawei Cloud. We first describe general characteristics of each cloud, and then detail the experimental datasets.

3.1 Azure Data

Azure data is a large public trace of VM data from a 30-day window, originally released as AzurePublicDatasetV1 [17]. The trace contains VM start, stop, resource, and (anonymized) customer/user ID information for over 2 million VMs, as well as CPU utilization readings at 5-minute intervals (not used in this paper). There are 16 different CPU/memory combinations, which we take as the workload flavors.

All timestamps in the Azure trace are quantized to 5-minute intervals, and the data is both left and right censored (i.e., VM start/end times have been truncated to the start/end time of the trace observation window). We discard any VMs that are running at the beginning of the trace (avoiding *survivorship bias*). For VMs that have an end time equal to the end of the observation window, we mark the VM as *censored* in the data at that time. Even though timestamps are quantized to 5-minute intervals, the ordering in `vmtable.csv` reflects the actual arrival order of VMs,[3] so we are able to determine the within-period ordering.

The Azure data does not identify the exact date and time of the observation window, so we simply treat the offset timestamps as Linux epoch times. Since the offset of the mapping between real-world time and our temporal features is arbitrary, the temporal features remain equally effective at modeling daily and weekly seasonality.

	Window size (days)			Number of VMs		
	Train	Dev	Test	Train	Dev	Test
Azure	20.8	3.5	5.7	1.2M	259K	410K
Huawei Cloud	274	14	17	1.7M	116K	140K

Table 1. Experimental datasets

3.2 Huawei Cloud data

Huawei Cloud data comes from a subset of our own public cloud. We collect VM start, stop, flavor, and anonymized customer/user ID information from a 10-month observation window, and again discard VMs running at the beginning of this window. The collected data contains 259 distinct VM flavors (reflecting CPU/memory combinations, specific resource attributes such as local disk or GPU requirements, and multiple generations of server hardware).

To mitigate the impact of censoring in our held-out *evaluation* data, we continue monitoring test VMs for two months beyond the test window, and right-censor any VMs still running at the end of that time (i.e., marking VMs as *censored*, with a known lifetime up to the end of those two months). We order the arrivals within each 5-minute period, and then quantize our timestamps to 5-minute intervals.

3.3 Experimental Datasets

Table 1 provides statistics on the experimental datasets. We treat each experimental window as a distinct observation window. That is, whenever we train on a training set or tune on a development set, we only include information that is available by the end point of that respective window. Specifically, for all VMs that start in one observation window that are still running at the end of the window, we right-censor their lifetimes to the end of that window. Figure 3 illustrates this concept. The only exception is the test window of Huawei Cloud data, where, as mentioned earlier, we censor two months beyond the end of the test window.

[2]Rather than sampling a lifetime for every job incrementally as we iterate through the sequence, it is also possible to sample entire *sequences* of lifetimes according to their joint probability with the resources, using a variation of the forward-backward algorithm for HMMs [11].

[3]See https://github.com/Azure/AzurePublicDataset/issues/7. The absence of this property precluded our use of the Azure V2 dataset.

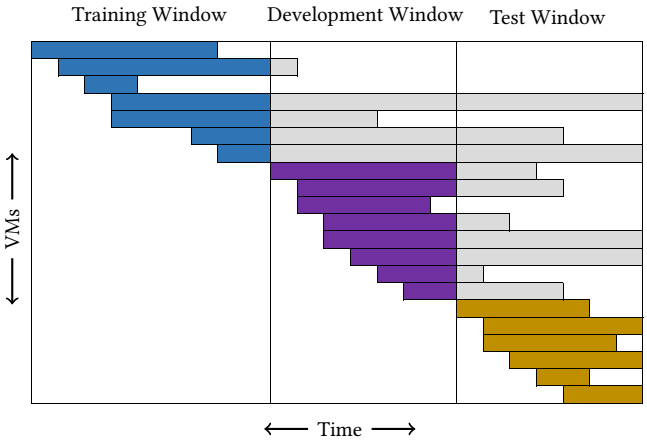


Figure 3. VM censoring: VMs are represented as horizontal bars spanning from the VM start to the VM end time. VM lifetimes are censored at the end of each respective observation window.

4 Experimental Setup

We now provide further details on model training and hyperparameter tuning (descriptions of evaluation metrics and baselines are in relevant subsections of the results).

4.1 Model Training

The Poisson regression models are trained using iteratively re-weighted least squares (IRLS) (via the GLM package in statsmodels (www.statsmodels.org)). The LSTM losses are iteratively minimized using minibatch gradient descent with the Adam optimizer [37]. All LSTM experiments are done using PyTorch (http://pytorch.org). Since our workloads are VMs, we use the flavor LSTM as our resource LSTM.

To achieve better numerical stability when computing the lifetime loss, we use a loss function that combines the logistic mapping together with the log likelihood computation into a single (vectorized) function (`BCEWithLogitsLoss` in Pytorch). We pass a mask vector into the `weight` parameter of this function in order to ignore the loss on those outputs that do not factor into the loss calculation.

4.2 Model Hyperparameters

The elastic net regularization penalty for Poisson regression, and the weight decay and learning rate for the LSTM resource/lifetime models, are tuned on the corresponding Azure or Huawei Cloud development sets. That is, we do not optimize these parameters for end-to-end generation, but for their stage-specific (and cloud-specific) development data.

The following parameters were tuned on Azure development data, and we use the same values for Huawei Cloud. We use a 2-layer LSTM with 200 hidden units in each layer. Unless otherwise noted, we use 47 bins in the hazard function. The final bin starts at 20 days (virtually all uncensored lifetimes in both clouds terminate within 20 days). For sampling

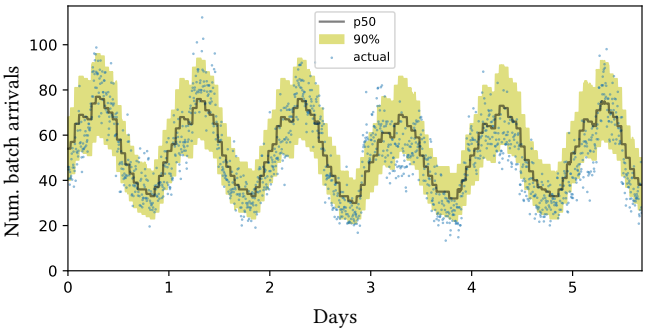


Figure 4. Actual and generated (with median, 90% prediction intervals) *batch arrivals* over Azure test window. 82.5% of true values are captured in the 90% prediction interval.

the *DOH* day (Section 2.1.2), we use the geometric approach (going backward from N), with a success probability of 1/7 (i.e., the expected value is 7 days before N).

We train both our resource and lifetime LSTMs using minibatches, each containing 50 sequences of length 5000. We zero the hidden state before each forward pass. We use long sequences partly to allow the models to capture any momentum in properties that may persist across periods. Our networks are thus uniquely deep both in space (multiple layers transform inputs to outputs) and time (data passes through thousands of layers of computation as we step forward through jobs). Backpropagation remains tractable due to the massive parallelism and available GPU hardware.

5 Prediction Results

In this section, we evaluate the predictions of each stage of our model and compare our approach to traditional methods.

5.1 Batch Arrivals

For batch arrivals (Section 2.1), we sample 500 values from our Poisson distribution on each test period and compute 90% prediction intervals of the sample distribution. We compute the coverage of true arrivals by the prediction intervals.

Azure. On Azure, the batch arrival prediction intervals (Figure 4) capture 82.5% of true values. For the batch model, we sample the *DOH* day using the geometric approach (Section 2.1.2). If we instead always encode the last day of the history (day N), only 56.5% of true values are captured in the 90% prediction interval (not shown). On this dataset, sampling *DOH* days evidently improves coverage significantly.

Huawei Cloud. On Huawei Cloud, the prediction intervals capture 94.5% of arrivals (Figure 5). Coverage is higher here because, with lower counts, the quantile function has fewer steps, so 90% intervals match larger intervals in many periods. Instead of sampling *DOH* days, if we encode the last day of the history, coverage is 95.0% (not shown). Here, sampling *DOH* days is not essential for high coverage.

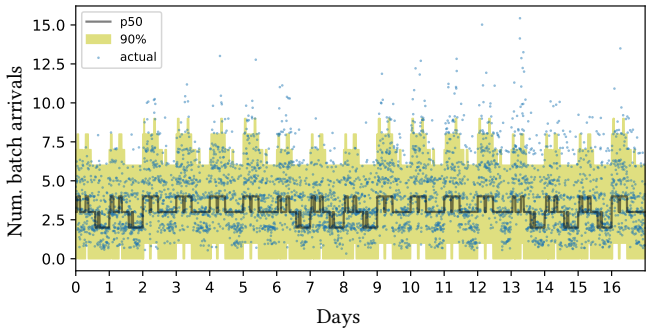


Figure 5. Actual and generated (with median, 90% prediction intervals) *batch arrivals* over Huawei Cloud test window. 94.5% of true values are captured in the 90% prediction interval (counts are jittered slightly to better show the density).

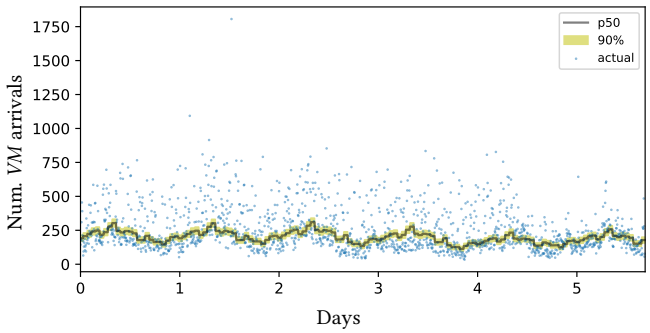


Figure 6. Actual and generated (with median, 90% prediction intervals) *individual VM* arrivals over Azure test window. Modeling raw job arrivals with Poisson greatly underestimates the variance in the arrival process (only 18% of true values are captured in the 90% prediction interval).

Modeling Individual *Job* Arrivals. In prior work, Poisson models are not used to model *batch* arrivals, but rather the arrival of *individual jobs* [12, 21, 26, 47, 48]. We evaluate how well such models capture the arrival process in VM workloads. We use the same Poisson regression model as for batch arrivals, but train on the raw number of VM arrivals in the training periods (and we do not use *DOH* features).

On Azure, modeling individual VM arrivals clearly does not capture the true variation in arrivals (Figure 6). On Huawei Cloud, the results are better, but only 52.9% of true values are captured in the 90% prediction interval (not shown). If we include sampled *DOH* days, 90% coverage increases to 51.4% for Azure data, and 68.2% for Huawei Cloud.

For arrivals in the cloud, Poisson is a better fit for batches than for jobs. In either case, sampling *DOH* days is an effective method for increasing forecast coverage.

	Azure		Huawei Cloud	
System	NLL	1-Best-Err	NLL	1-Best-Err
Uniform	2.83	93.9%	5.55	99.6%
Multinomial	1.58	54.7%	3.34	89.7%
RepeatFlav	N/A	29.7%	N/A	71.3%
LSTM	**0.65**	**25.7%**	**2.10**	**59.2%**

Table 2. Modeling flavor sequences with an LSTM significantly improves NLL and 1-Best-Err in both clouds (N/A indicates undefined results for non-probabilistic baselines). The LSTM's accuracy at predicting next flavors (1 of 259 possible options for Huawei Cloud) is impressive.

5.2 Flavors

We evaluate our flavor LSTM (Section 2.2.1) by comparing its predictions to the predictions of the following baselines:

Uniform Each flavor is equally likely to occur at each step

Multinomial Each flavor's probability is given by the empirical count of that flavor in training data (replicating the traditional, independent-arrival model)

RepeatFlav The next flavor is always predicted to be the same as the previous one (defaulting to multinomial after *EOB*)

We use the following evaluation metrics:

NLL Negative-log-likelihood of next-step probabilities

1-Best-Err Next-step 1-best classification error rate (note, for this metric, the traditional Multinomial approach will output the most frequent flavor)

These metrics (and those for lifetimes below) are computed at each step assuming knowledge of the sequence up to that step. By the chain rule of probability, NLL is thus effectively computing the log likelihood of the entire test set; test set likelihood is the standard evaluation for generative neural networks (when the computation is tractable) [28, 30, 66].

Results are presented in Table 2. Selecting flavors according to their empirical probability is more predictive than selecting uniformly, but significantly worse than repeating the previous flavor. The LSTM works best; for both NLL and 1-Best-Err, it is significantly better than RepeatFlav; the most probable flavor is not always a repeat of the previous one.

5.3 Lifetimes

We now evaluate our lifetime LSTM (Section 2.3). Our first set of results concerns predicting the binned lifetime. We compare the LSTM's predictions to the following baselines:

CoinFlip Hazard in every bin is assumed to be 50%

Overall KM Lifetimes are estimated using the Kaplan-Meier discrete hazard for all flavors pooled into one group

Per-flavor KM Lifetimes are estimated using the Kaplan-Meier discrete hazard for each specific flavor type

System	Azure BCE	Azure 1-Best-Err	Huawei Cloud BCE	Huawei Cloud 1-Best-Err
CoinFlip	0.693	97.1%	0.693	49.5%
Overall KM	0.277	73.8%	0.383	49.5%
Per-flavor KM	0.270	71.5%	0.322	40.1%
RepeatLifetime	N/A	43.4%	N/A	23.9%
LSTM	**0.127**	**27.8%**	**0.098**	**11.2%**

Table 3. Modeling lifetimes with an LSTM significantly improves BCE and 1-Best-Err in both clouds (N/A indicates undefined results for non-probabilistic baselines). The LSTM has exceptional prediction ability, picking the single lifetime bin (from 47 options) with 89% accuracy for Huawei Cloud.

System	Discretization	Interpolation	Survival-MSE
KM	47 bins	None (Stepped)	1.12%
KM	495 bins	None (Stepped)	1.11%
KM	47 bins	CDI	1.11%
KM	495 bins	CDI	1.11%
KM	Continuous	N/A	1.09%
LSTM	47 bins	None (Stepped)	0.52%
LSTM	47 bins	CDI	**0.47%**

Table 4. Evaluation in continuous domain: number of bins and interpolation method has very minor impact on accuracy of KM survival functions. Meanwhile, continuous-density interpolation (CDI) is important for the LSTM. The benefits of using an LSTM far exceed the drawbacks of discretization.

RepeatLifetime Next VM lifetime is always assumed to be same as previous (defaulting to Overall KM after *EOB*).

The following evaluation metrics are used:

BCE Binary cross entropy loss of next-step probabilities

1-Best-Err Next-step error rate where hazard converted to PMF and maximum likelihood bin selected

LSTM predictions are much more accurate than flipping a coin, but also more accurate than Kaplan-Meier estimators (Table 3). Looking at 1-Best-Err, it is remarkable that the LSTM is able to predict the single lifetime bin (out of 47 options) with only 27.8% error on Azure and 11.2% error in Huawei Cloud, in the latter case less than half the error rate achieved when repeating the lifetime of the previous VM.

Next, we consider the question of how well our models perform when their discretized probabilities are converted to the continuous domain. Discretization is a double-edged sword: using a smaller number of bins makes our models less complex and easier to train, but potentially increases the reconstruction error in mapping back to real values. And unlike an LSTM, the Kaplan-Meier baseline applies directly in continuous space. Thus an important question is: do the benefits of discretization exceed potential drawbacks?

To test this, we use the continuous-domain Survival-MSE evaluation [39] on the Azure test data: we convert our discrete hazard outputs to a continuous survival function, then compare the MSE between this survival function and the true survival function for each job. For interpolation, we compare the CDI method (Section 2.4) to a survival "step function". That is, whereas CDI assumes job terminations are spread evenly within a lifetime bin, Stepped assumes that all job terminations happen at the lifetime boundaries.

Having more bins helps Kaplan-Meier only marginally, while having good interpolation seems to help more, especially for the LSTM (Table 4). Having a better model helps even more: the LSTM has half the MSE of other models, regardless of bins or interpolation. We experimented with a 495-bin-LSTM, but results were not nearly as good as with 47 bins. Having more bins hurts the LSTM due to increased complexity in both input features and output dimensionality.

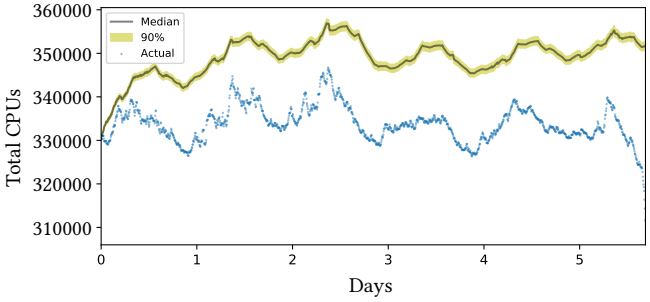
We experimented with alternate versions of the KM estimator that either (1) ignored censored VMs entirely or (2) assumed censored VMs terminated at the end of the observation window. Version (2) consistently worked better than version (1), and, notably, BCE was usually close to the vanilla censoring-aware KM. In typical survival analysis work [39], sometimes 86% of data is censored. But in Azure, only 3.2% of arriving VMs are censored. If we had focused on estimating lifetime solely for VMs that arrived within the past 20 minutes, censoring would play a major role, but in our current evaluation, handling censoring is much less important than modeling inter-job correlations.

6 Use Case Results

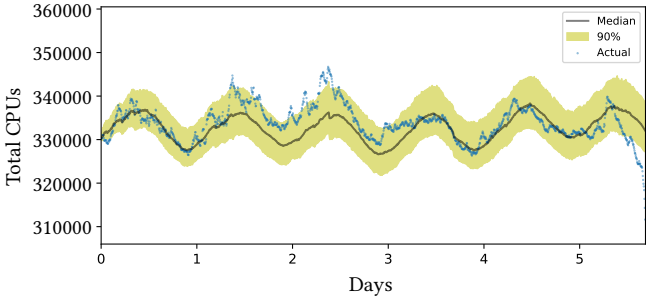
We now evaluate our proposed trace generator end-to-end in the context of two use cases, and we compare to two end-to-end generation baselines: Naive and SimpleBatch. Naive is a three-stage process that ignores inter-job correlations: (1) sample a number of VM arrivals for each period from a Poisson regression (Section 5.1), (2) sample a flavor for each VM from the flavor multinomial (Section 5.2), and (3) sample a lifetime for each VM from the per-flavor Kaplan-Meier (Section 5.3). We also explore whether a novel batch-based baseline (one that does not use RNNs) can be effective: SimpleBatch follows a four-stage process: (1) sample a number of batch arrivals for each period from our proposed Poisson regression, (2) sample a size of each batch from the empirical batch size distribution in training data, (3) sample one flavor for all VMs in a batch from the flavor multinomial (Section 5.2), (4) sample one lifetime for all VMs in a batch from the per-flavor Kaplan-Meier (Section 5.3).

6.1 Capacity Planning

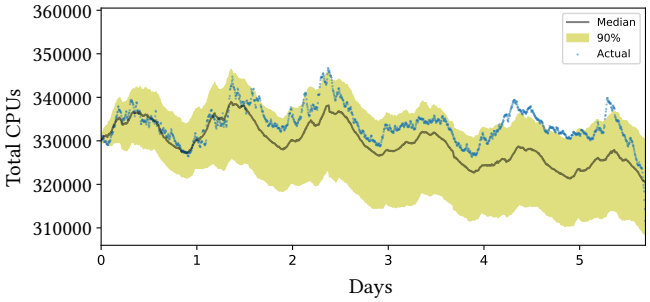
Capacity planning is the task of forecasting future demand in order to ensure sufficient supply. At Huawei Cloud, capacity engineering teams rely on forecasts of future resource usage



(a) NAIVE-generated: 0% captured in 90% prediction interval

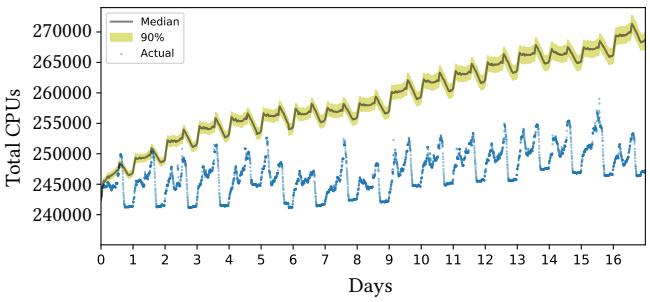


(b) SIMPLEBATCH-generated: 88% captured in 90% prediction interval

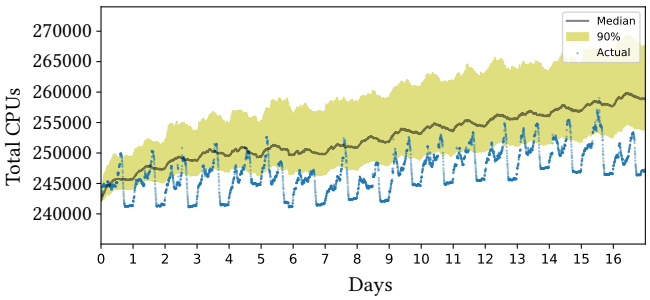


(c) LSTM-generated: 83% captured in 90% prediction interval

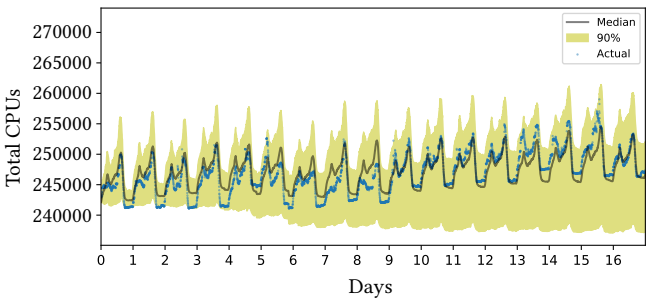
Figure 7. Actual and generated (with median, 90% prediction intervals) total workload over Azure test window for (a) NAIVE, (b) SIMPLEBATCH, and (c) LSTM.



(a) NAIVE-generated: 1% captured in 90% prediction interval



(b) SIMPLEBATCH-generated: 24% captured in 90% prediction interval



(c) LSTM-generated: 93% captured in 90% prediction interval

Figure 8. Actual and generated (with median, 90% prediction intervals) total workload over Huawei Cloud test window for (a) NAIVE, (b) SIMPLEBATCH, and (c) LSTM.

in order to make server purchasing decisions. We evaluate how well our system can support this use case by considering how well it generates forecasts for the total number of CPUs active at each moment of the test window.

We sample a single workload from our model by running our three-stage process for each test period (Section 2.4). To compute a distribution over possible future workloads, we generate 500 such workloads. We calculate total CPU usage over time in each of these samples, and compute the median and 90% prediction intervals over this distribution. To evaluate, we compute the proportion of true data covered by the 90% prediction interval. As a constant across all models, we include in the total workload all VMs already running at the beginning of the test window (using their actual lifetimes).

Azure. On Azure, NAIVE does not capture any of the future workload in its 90% intervals (Figure 7a). This is remarkable considering NAIVE has been the default workload model used by practitioners [34, 38]. In modern cloud workloads, VMs are simply not independent and should not be modeled using independence assumptions. Both SIMPLEBATCH and LSTM work much better (Figures 7b and 7c). SIMPLEBATCH intervals miss the actual workload briefly on day 2 and 6, while LSTM misses for parts of day 5 and 6.

Huawei Cloud. For Huawei Cloud, NAIVE is again a poor workload predictor (Figure 8a). SIMPLEBATCH, though better, generates a total CPU load well above the observed ground truth (Figure 8b). Meanwhile, LSTM covers 92.8% of the true workload in the 90% interval (Figure 8c). We attribute the strong performance of LSTM over SIMPLEBATCH on this

dataset to two factors. First, SimpleBatch uses distributions calculated over the entire training set, but Huawei Cloud training data covers 9 months, over which the workload was growing quickly. Workload growth had leveled-off significantly by the test window. By sampling *DOH* days according to our geometric approach, LSTM generates future workload that resembles the recent past. Indeed, if we remove the *DOH* features, LSTM only covers 61.9% of the true workload. Secondly, on this dataset, LSTM is better able to capture the daily workload pattern. Comparing the median and upper 90% limit in Figures 8b and 8c, we see the LSTM output is significantly more reflective of the true workload pattern.

In Huawei Cloud, the median LSTM workload almost exactly matches the true workload for the final 7 days of the test window (Figure 8c). Since the training window ends almost a month earlier, it is remarkable that future workload can be generated with such high fidelity, so far in advance.

6.2 Workload Scheduling

Workload scheduling is the process of placing VM requests onto specific physical servers. At Huawei Cloud, our VM scheduler [36] handles clusters of up to 10K machines. Like Azure's Protean [31], it has been designed and optimized using both historical and synthetic data. To be useful in evaluating schedulers, synthetic data must reflect key properties of real data. In this section, we describe two properties of traces, *reuse distance* and *packing fragmentation*, that directly affect scheduling performance and quality, respectively. We generate 500 traces for each test period for each of our generators, and evaluate whether measurements of these properties on our generated traces match those on actual test data.

Scheduling Performance. Hadary et al. [31] introduce a workload metric called *reuse distance*, "which for each request of VM type v, measures the number of unique VM types requested since the last time that v was requested." Computed over a large trace, higher frequency of small reuse distances implies subsequent requests tend to be similar to previous ones. Observation of such a pattern in Azure data "motivate[d] the caching of placement evaluation logic, and reuse across multiple requests – this idea is central in [Protean's] design and facilitates scaling to large zones and regions." The size of Protean's cache is tuned based on "memory footprint and hit-rate considerations." Reuse distance therefore affects both the design and tuning of VM schedulers. When tuning on synthetic data, it is imperative that such data reflect the reuse behavior of actual data.

We evaluate our generated data by computing the distribution of reuse distances across the 500 sampled traces from each generator, and compare to the true reuse distance pattern in our actual test period.

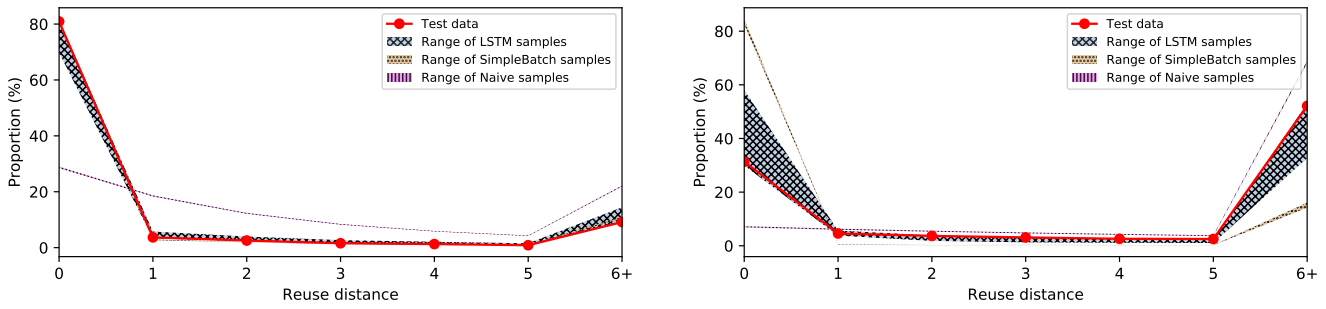
Results are shown in Figure 9. In both clouds, Naive traces exhibit larger reuse distances (i.e., less reuse) than actual data. A scheduler tuned on Naive traces would appear to require a very large cache in order to achieve a desired hit-rate, and thus could be configured with an unnecessarily-large memory footprint. On Azure, SimpleBatch has a similar distribution to test data (but very small variance), whereas on Huawei Cloud, SimpleBatch greatly overestimates flavor reuse. Tuning our scheduler on SimpleBatch traces would lead us to configure too small a cache, resulting in poor performance in production. LSTM traces are the only traces that match the reuse pattern of the actual test data across both clouds.

Scheduling Quality. Due to the heterogeneity of VM workloads, VM requests cannot be packed perfectly onto servers, and some *fragmentation* of resources is inevitable. Since extra capacity must be provisioned in order to cover resources lost this way, cloud providers require good fragmentation estimates. Moreover, schedulers must efficiently place requests in order to minimize fragmentation.

It is common to compare different packing algorithms on traces and choose the one that achieves the lowest fragmentation (e.g. [29, 36, 54]). Packing algorithms may also have parameters that can be tuned in order to minimize fragmentation. Schedulers based on deep reinforcement learning [49, 50] may have thousands or millions of such parameters and they must be optimized on large amounts of synthetic data. On the scale of cloud providers, such tuning is very important; Hadary et al. report that "even 1% in fragmentation reduction can lead to cost savings in the order of $100M per year" [31].

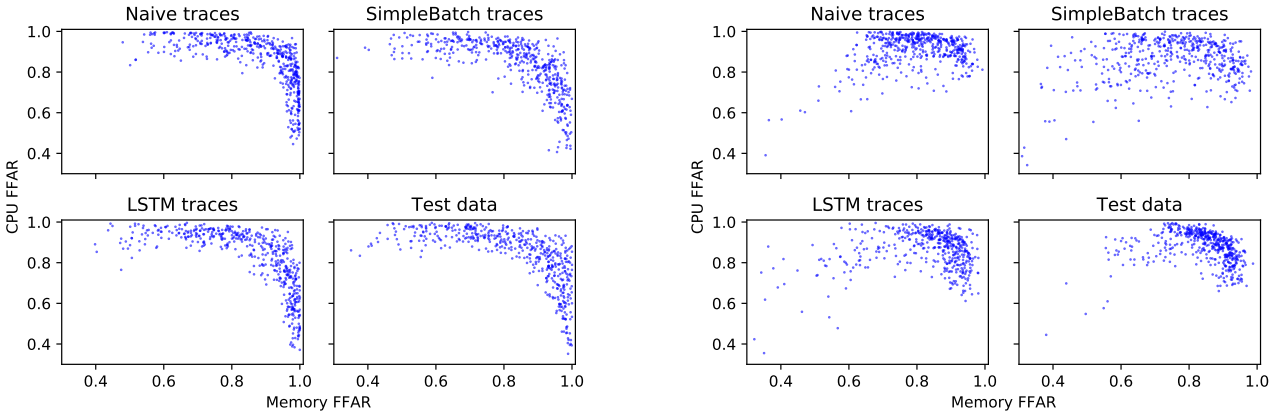
When quantifying the resources lost to fragmentation, or when optimizing packing algorithms in order to achieve lower fragmentation, we need to estimate the true fragmentation of a given packing algorithm as accurately as possible. For synthetic data to be useful, packing of this data must achieve similar fragmentation to packing of actual data.

Here, we measure fragmentation by computing the *first-failure allocation ratio* (FFAR). FFAR computes the proportion of allocated capacity at the point of first scheduling failure (i.e., first point with nowhere to place a request). The full evaluation procedure is as follows: for each of our 500 generated traces (and 500 times on actual test data), we randomly sample a scheduling tuple consisting of a: (1) start point for packing, (2) number of servers, (3) server CPU and Memory capacity, and (4) packing algorithm from one of {Random placement, busiest-fit, cosine similarity [29], delta perp-distance [36]}. We then pack the trace onto the servers using the given algorithm, beginning from the start point and proceeding through arrivals, and departures, until we have a failure, at which point we compute FFAR for CPU and Memory. We selected ranges for server number and capacity such that CPU and Memory were each the limiting resource in roughly 50% of packings on training data. To reduce variance, we use the same random 500 scheduling tuples for each generator.



(a) Azure **(b)** Huawei Cloud

Figure 9. Reuse distance distributions for generated traces and actual data over Azure (9a) and Huawei Cloud (9b) test windows. LSTM traces are the only traces that consistently match the reuse pattern of actual test data across both clouds.



(a) Azure **(b)** Huawei Cloud

Figure 10. Packing results on generated traces and actual test data over Azure (10a) and Huawei Cloud (10b) test windows. Each point is the outcome of a packing experiment, indicating the first-failure allocation ratio for the CPU (y-axis) and Memory (x-axis) resources. In both clouds, LSTM generates traces whose packability best matches that of actual data (see also Table 5).

	Azure		Huawei Cloud	
Generator	Median	>0.95	Median	>0.95
NAIVE	96.7	65.4	93.9	40.6
SIMPLEBATCH	93.5	37.0	91.6	23.4
LSTM	**95.4**	**53.5**	**92.3**	**21.6**
Test data	*94.5*	*47.2*	*92.2*	*18.6*

Table 5. First-failure allocation ratio (FFAR) summary statistics for the *limiting resource* (corresponding to Figure 10 experiments): median value and proportion of times >95%. LSTM-generated traces pack most similarly to real test data.

Table 5 provides the summary metrics for the packing experiments, while full results are plotted in Figure 10. The following key patterns emerge:

(1) *Naively-generated traces are misleadingly easy to pack:* traces from the Naive method result in much higher FFARs, with many more packings reaching >95% FFAR for the limiting resource (e.g. 40.6% for Naive vs. 21.6% for real data in Huawei Cloud). Scheduling failures and fragmentation are driven by runs of many flavors of the same type in a row; such requests suddenly exhaust one resource (e.g. CPU) even while there is ample capacity in the other (e.g. Memory). Naive traces do not have long runs of the same flavor.[4]

(2) *SIMPLEBATCH traces are harder to pack than real traces:* SIMPLEBATCH has the opposite problem: unlike real data, SIMPLEBATCH has no flavor-variation within batches, resulting exclusively in long, hard-to-pack runs of the same flavor.

(3) *LSTM consistently packs the most similarly to real test data:* in both clouds, the packability of LSTM-generated traces best matches that of the actual data.

[4]One motivation for our work in this paper was a prior observation regarding our VM scheduler: we found that traces generated using traditional models were much easier to pack than real historical traces. We therefore needed a method to generate synthetic traces that did not have this problem.

We performed many variations of the above experiments and these patterns held true. For example, we ran the packings using only arrivals and no departures. We also did an arrival-only version with 10X the number of arrivals (by simply scaling our sampled arrival rates); both the reuse and FFAR distributions matched those from the unscaled setting. This is an encouraging result; we often face a scenario where workloads are increasing (either organically or because of resource consolidation) and we are tasked with ensuring the scheduler can handle a 10X higher rate of requests.

7 Related Work

Alternative Modeling Approaches. We have shown how a trace of cloud jobs can be generated using Poisson regression and neural sequence models. We now briefly discuss some alternative modeling approaches.

Rather than a three-stage generative process, it is possible to generate the trace with a single LSTM. The LSTM could control the number of batches in each period, via generation of a special end-of-period (*EOP*) token. Flavors, and durations conditioned on flavors, could be generated in a single output, making use of MADE-style [25] masking in order to ensure proper conditioning, as in PixelRNN [66]. While this would permit greater parallelism in training, generation would still require sequential output of flavors, then durations.

We did not pursue this direction for two main reasons. First, when we experimented with *EOP* tokens, we found the generated workload was exquisitely sensitive to the timely sampling of these tokens, and it was difficult to convey their supreme importance to our network without throwing off our training procedure. Having a separate stage to generate the number of batches was empirically more successful. Secondly, by having an explicit parameter for arrival rate, our approach allowed us to generate 10X workloads by changing a single line of code in our generator.[5]

LSTMs are perhaps the simplest network (in terms of manual tuning) that can reliably model long-term dependencies, but other, more-complex architectures may also be used. For example, a GAN [28] uses paired generator/discriminator networks to enable very realistic output; our work provides the networks that can now be used inside the GAN. Furthermore, Transformers [67] provide a mechanism for superparallelizing training of sequence models (like ours) to enormous data sets, and could be used in place of the LSTMs.

[5]For the single-LSTM approach, it would theoretically be possible to perform what-if experiments by post-processing the LSTM *output probabilities* (e.g., dividing *EOP* token probability by 10). We could use this strategy in our current multi-stage approach as well, e.g., to simulate larger or smaller batches (modifying the *EOB* probability), or greater likelihood of certain flavors. More work is needed in order to determine whether such strategies degrade any desired properties of the generated traces (e.g. reuse distance for the 10X workloads). An interesting alternative to post-processing is the use of autoencoding methods that "disentangle factors of variation" [41]; such methods permit modifications to latent attributes, which are then reflected in the realistic generated data.

It required some domain expertise in order to develop our initial effective generative process. There are approaches that evolve neural architectures [74], either to better fit the data, or because of changes in the environment (which may be particularly common in the cloud domain).

Workload Analysis. For an overview of research in cloud workload analysis (including analysis of arrivals, job behavior, load fluctuations, etc.), see [9]. Much recent work has been enabled by the release of large-scale traces from Facebook [14], Google [58, 65], Alibaba [1], Microsoft Azure [17], and others, including the influential work of Reiss et al. [57]. Chung et al [16] analyze a large trace in order to discover job workflow constraints (e.g., job A consumes job B's output). Whether workflow constraints can be integrated into our generative process merits further research.

Cortez et al. [17] provide the first detailed characterization of *public* cloud provider workload, analyzing Azure's entire VM workload over a three-month window. They introduce a discriminative classifier for predicting VM lifetime, with lifetime binned into 4 groups (essentially a very coarse, non-probabilistic PMF parameterization). However, when they generate data for their scheduler evaluation, they notably rely on a simple univariate (generative) lifetime distribution. Hadary et al. [31] quantify flavor reuse across VM requests. They also note that reuse occurs in other dimensions of VM requests, such as priority and *RequireIsolation*.

Workload Generation. Moreno et al. [53] note that most analysis papers "do not provide a structured model which can be used for conducting simulations." To remedy this, and to take a step toward modeling the diversity of cloud workloads, they introduce a system that models a few user-specific arrival rates, resource requirements, and lifetimes. In contrast, we do not focus on specific users, and can therefore model large-scale future workloads that include entirely new users. Bahga and Madisetti [4] introduce a workload generator for evaluation of cloud *applications*. They model user behavior such as inter-session intervals and session lengths.

A number of papers focus specifically on modeling job arrival rates. Juan et al. [34] introduce a dynamic, hierarchical approach to group arrivals into "bundles". In contrast, we batch based on user ID and arrival time (Section 2) and show this is effective. Koltuk and Schmidt [38] propose an iterative, heuristic approach to generating time-varying arrival rates in cloud workloads. In contrast, we sample from an inhomogeneous Poisson regression. In non-cloud domains requiring high-precision inter-arrival times, there is relevant recent work modeling event streams (a.k.a. asynchronous sequences), including deep point process models [20, 43].

In the domain of networking, packet traces exhibit similar properties to VM workloads, including spatial/temporal heterogeneity [40], time-of-day and day-of-week patterns [7, 63], inhomogeneous arrival rates [71] and inter-packet correlations [3]. Moreover, networking technology is evaluated

using both real and synthetic traffic [13]. Avin et al. [3] decompose packet correlation into temporal and non-temporal structure, and provide an information-theoretic approach to analyze and generate traces. It would be interesting to apply our LSTM approach in this domain (perhaps replacing flavors with packet source/destination pairs, lifetimes with packet sizes, etc.).

Workload Forecasting. While our model generates all individual workload start and stop events, an alternative approach to capacity planning is to leverage techniques from time series forecasting. A number of recent papers have shown that deep learning architectures, from LSTMs [60], to transformers [70], to informers [73], can effectively estimate future behavior from prior history, generating output ranging from point predictions [10, 61], to full probabilistic forecasts [23, 56]. DoppelGANger [44, 45] generates job-specific time series without prior history; it first generates a fixed set of "metadata" categorical attributes, and then generates the job's time series conditional on its metadata. In contrast, we generate both real-valued (durations) and categorical attributes (flavors) over time, and model inter-job correlations.

Forecasting approaches can be used to predict the total CPU, memory, etc. of specific flavors, groups of flavors, regions, etc., by generating specific predictions for any of these dimensions, at specific time granularities (e.g. daily averages). In one sense, our workload model is more powerful: we can generate full probabilistic predictions, for any dimensions of interest, at any time granularity, by simply summing the relevant VMs from our generated traces. However, forecasting methods can predict dimensions that are not part of our generative story (e.g., forecasting workload for specific users).

Survival Analysis. Neural networks have been used previously to parameterize survival functions. Gensheimer and Narasimhan [24] first proposed the hazard-based loss function. They note that this loss "has been well studied for discrete-time survival models in a non-deep learning context" and show how it differs from the similar, "heuristic" loss used previously with neural networks [8]. Kvamme and Borgan [39] provide an excellent synthesis of recent neural network survival research, connecting prior work to parameterizations of either the discrete hazard function [24], the discrete PMF [22, 42], or continuous-time methods based on the Cox proportional hazards assumption [15]. Our work can be viewed as a novel sequential, inter-case extension of this line of research.

Recurrent neural networks have also been used in survival, but not to model *inter-case* relationships. Rather, RNNs have been used to represent the sequence of survival states for one patient [27] and the sequence of hazard functions over time [59]. Although we focused on cloud workloads, our approach to handling inter-case relationships should be useful for other survival scenarios. Indeed, in any system where there is congestion, there may be inter-case correlations in the waiting or service times [62]. Our recurrent lifetime model is a new tool for modeling such correlations.

Privacy and Synthetic Data. In the same way that companies have released large pre-trained neural networks for text (e.g. [18]), cloud providers could potentially leverage our work to release large pre-trained models of systems data, rather than the proprietary traces. Preserving privacy is an active area of ML [2, 6, 33]. For systems data, preserving privacy generally requires protecting both business secrets and user information [45]. Since our model does not generate user-specific information, the primary concern of cloud providers would be "leaking information about the types of resources available and in use at the enterprise" [45]. Aspects of our model (such as arrival rates or flavor trends) can be altered for confidentiality reasons. Technically, this is the same problem as performing what-if experiments (see Footnote 5). Progress in what-if generation can provide mechanisms for providers to release slightly *fictitious*, but still highly useful, trace generators.

8 Conclusion

We have designed, implemented, and evaluated an approach to cloud workload modeling that successfully models the real production VM workload of two large-scale cloud providers.

The main conceptual contribution of the paper is an extension of recent advances in neural survival analysis in order to model inter-job lifetime correlations. Our approach draws on the ability of recurrent neural networks to model complex, long-range dependencies, and to generate realistic output with similar characteristics to training sequences. This is the first model of large-scale cloud workloads that captures long-range inter-job correlations in arrival rates, flavors, and lifetimes, and it is able to create synthetic workload of a quality not previously seen from cloud workload generators. Compared to commonly-used approaches, the model is highly effective at estimating the probability of flavors and lifetimes in actual data. Moreover, it shows promise as a tool for medium-term capacity planning, and its generated traces have key properties necessary for optimizing the performance and quality of VM schedulers.

As RNNs are easier to apply than ever, and more and more data becomes available in cloud environments, we believe there is great potential for both practitioners and researchers to leverage and extend this work. In particular, designing and evaluating extensions of the resource model to DAG-of-task workloads [14, 57] is an important area for future research.

Acknowledgments

We thank our shepherd, Roxana Geambasu, and the SOSP reviewers for their helpful comments.

References

[1] Alibaba cluster trace program. https://github.com/alibaba/clusterdata, 2017.

[2] Abadi, M., Chu, A., Goodfellow, I., McMahan, H. B., Mironov, I., Talwar, K., and Zhang, L. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (2016), pp. 308–318.

[3] Avin, C., Ghobadi, M., Griner, C., and Schmid, S. On the complexity of traffic traces and implications. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 4*, 1 (2020), 1–29.

[4] Bahga, A., and Madisetti, V. K. Synthetic workload generation for cloud computing applications. *Journal of Software Engineering and Applications 4*, 07 (2011), 396.

[5] Barroso, L. A., and Hölzle, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture 4*, 1 (2009), 1–108.

[6] Beaulieu-Jones, B. K., Wu, Z. S., Williams, C., Lee, R., Bhavnani, S. P., Byrd, J. B., and Greene, C. S. Privacy-preserving generative deep neural networks support clinical data sharing. *Circulation: Cardiovascular Quality and Outcomes 12*, 7 (2019).

[7] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (2010), pp. 267–280.

[8] Brown, S. F., Branford, A. J., and Moran, W. On the use of artificial neural networks for the analysis of survival data. *IEEE transactions on neural networks 8*, 5 (1997), 1071–1077.

[9] Calzarossa, M. C., Massari, L., and Tessera, D. Workload characterization: A survey revisited. *ACM Computing Surveys (CSUR) 48*, 3 (2016), 1–43.

[10] Cao, D., Wang, Y., Duan, J., Zhang, C., Zhu, X., Huang, C., Tong, Y., Xu, B., Bai, J., Tong, J., and Zhang, Q. Spectral temporal graph neural network for multivariate time-series forecasting. *Advances in Neural Information Processing Systems 33* (2020).

[11] Cawley, S. L., and Pachter, L. HMM sampling and applications to gene finding and alternative splicing. *Bioinformatics 19*, suppl_2 (2003), ii36–ii41.

[12] Chen, J., Wang, C., Zhou, B. B., Sun, L., Lee, Y. C., and Zomaya, A. Y. Tradeoffs between profit and customer satisfaction for service provisioning in the cloud. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing* (2011), HPDC '11, pp. 229–238.

[13] Chen, K., Singla, A., Singh, A., Ramachandran, K., Xu, L., Zhang, Y., Wen, X., and Chen, Y. OSA: An optical switching architecture for data center networks with unprecedented flexibility. *IEEE/ACM Transactions on Networking 22*, 2 (2013), 498–511.

[14] Chen, Y., Ganapathi, A., Griffith, R., and Katz, R. The case for evaluating MapReduce performance using workload suites. In *2011 IEEE 19th annual international symposium on modelling, analysis, and simulation of computer and telecommunication systems* (2011), IEEE, pp. 390–399.

[15] Ching, T., Zhu, X., and Garmire, L. X. Cox-nnet: an artificial neural network method for prognosis prediction of high-throughput omics data. *PLoS computational biology 14*, 4 (2018), e1006076.

[16] Chung, A., Krishnan, S., Karanasos, K., Curino, C., and Ganger, G. R. Unearthing inter-job dependencies for better cluster scheduling. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)* (2020), pp. 1205–1223.

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

[18] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[19] Downey, A. B., and Feitelson, D. G. The elusive goal of workload characterization. *ACM SIGMETRICS Performance Evaluation Review 26*, 4 (1999), 14–29.

[20] Du, N., Dai, H., Trivedi, R., Upadhyay, U., Gomez-Rodriguez, M., and Song, L. Recurrent marked temporal point processes: Embedding event history to vector. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016), pp. 1555–1564.

[21] Feng, G., Garg, S., Buyya, R., and Li, W. Revenue maximization using adaptive resource provisioning in cloud computing environments. In *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing* (2012), GRID '12, pp. 192–200.

[22] Fotso, S. Deep neural networks for survival analysis based on a multi-task framework. *arXiv preprint arXiv:1801.05512* (2018).

[23] Gasthaus, J., Benidis, K., Wang, Y., Rangapuram, S. S., Salinas, D., Flunkert, V., and Januschowski, T. Probabilistic forecasting with spline quantile function RNNs. In *The 22nd international conference on artificial intelligence and statistics* (2019), pp. 1901–1910.

[24] Gensheimer, M. F., and Narasimhan, B. A scalable discrete-time survival model for neural networks. *PeerJ 7* (2019), e6257.

[25] Germain, M., Gregor, K., Murray, I., and Larochelle, H. MADE: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning* (2015), PMLR, pp. 881–889.

[26] Ghaderi, J. Randomized algorithms for scheduling VMs in the cloud. In *Proceedings of IEEE Infocom* (2016), pp. 1–9.

[27] Giunchiglia, E., Nemchenko, A., and van der Schaar, M. RNN-SURV: A deep recurrent model for survival analysis. In *International Conference on Artificial Neural Networks* (2018), pp. 23–32.

[28] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. *Advances in neural information processing systems* (2014), 2672–2680.

[29] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), ACM New York, NY, USA, pp. 455–466.

[30] Graves, A. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* (2013).

[31] Hadary, O., Marshall, L., Menache, I., Pan, A., Greeff, E. E., Dion, D., Dorminey, S., Joshi, S., Chen, Y., Russinovich, M., and Moscibroda, T. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 845–861.

[32] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[33] Jordon, J., Yoon, J., and Van Der Schaar, M. PATE-GAN: Generating synthetic data with differential privacy guarantees. In *International conference on learning representations* (2018).

[34] Juan, D.-C., Li, L., Peng, H.-K., Marculescu, D., and Faloutsos, C. Beyond Poisson: Modeling inter-arrival time of requests in a datacenter. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (2014), pp. 198–209.

[35] Kaplan, E. L., and Meier, P. Nonparametric estimation from incomplete observations. *Journal of the American statistical association 53*, 282 (1958), 457–481.

[36] Ke, X., Guo, C., Ji, S., Bergsma, S., Hu, Z., and Guo, L. Fundy: A scalable and extensible resource manager for cloud resources. In *IEEE Cloud* (2021).

[37] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[38] Koltuk, F., and Schmidt, E. G. A novel method for the synthetic generation of non-IID workloads for cloud data centers. In *2020 IEEE Symposium on Computers and Communications (ISCC)* (2020), IEEE,

pp. 1–6.

[39] Kvamme, H., and Borgan, Ø. Continuous and discrete-time survival prediction with neural networks. *arXiv preprint arXiv:1910.06724* (2019).

[40] LaCurts, K., Mogul, J. C., Balakrishnan, H., and Turner, Y. Cicada: Introducing predictive guarantees for cloud networks. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)* (2014).

[41] Larsen, A. B. L., Sønderby, S. K., Larochelle, H., and Winther, O. Autoencoding beyond pixels using a learned similarity metric. In *International conference on machine learning* (2016), PMLR, pp. 1558–1566.

[42] Lee, C., Zame, W., Yoon, J., and van der Schaar, M. Deephit: A deep learning approach to survival analysis with competing risks. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2018), vol. 32.

[43] Li, L., Zhang, J., Yan, J., Jin, Y., Zhang, Y., Duan, Y., and Tian, G. Synergetic learning of heterogeneous temporal sequences for multi-horizon probabilistic forecasting. *arXiv preprint arXiv:2102.00431* (2021).

[44] Lin, Z., Jain, A., Wang, C., Fanti, G., and Sekar, V. Generating high-fidelity, synthetic time series datasets with DoppelGANger. *arXiv preprint arXiv:1909.13403* (2019).

[45] Lin, Z., Jain, A., Wang, C., Fanti, G., and Sekar, V. Using GANs for sharing networked time series data: Challenges, initial promise, and open questions. In *Proceedings of the ACM Internet Measurement Conference* (2020), pp. 464–483.

[46] Lu, C., Ye, K., Xu, G., Xu, C.-Z., and Bai, T. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)* (2017), pp. 2884–2892.

[47] Maguluri, S. T., and Srikant, R. Scheduling jobs with unknown duration in clouds. *IEEE/ACM Transactions On Networking 22*, 6 (2013), 1938–1951.

[48] Maguluri, S. T., Srikant, R., and Ying, L. Stochastic models of load balancing and scheduling in cloud computing clusters. In *Proceedings of IEEE Infocom* (2012), pp. 702–710.

[49] Mao, H., Alizadeh, M., Menache, I., and Kandula, S. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (2016), pp. 50–56.

[50] Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., and Alizadeh, M. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 270–288.

[51] Meier, P., Karrison, T., Chappell, R., and Xie, H. The price of Kaplan–Meier. *Journal of the American Statistical Association 99*, 467 (2004), 890–896.

[52] Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning* (2017), PMLR, pp. 2430–2439.

[53] Moreno, I. S., Garraghan, P., Townend, P., and Xu, J. Analysis, modeling and simulation of workload patterns in a large-scale utility cloud. *IEEE Transactions on Cloud Computing 2*, 2 (2014), 208–221.

[54] Panigrahy, R., Talwar, K., Uyeda, L., and Wieder, U. Heuristics for Vector Bin Packing. *research. microsoft. com* (2011).

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

[56] Rasul, K., Sheikh, A.-S., Schuster, I., Bergmann, U., and Vollgraf, R. Multi-variate probabilistic time series forecasting via conditioned normalizing flows. *arXiv preprint arXiv:2002.06103* (2020).

[57] 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 third ACM symposium on cloud computing* (2012), pp. 1–13.

[58] Reiss, C., and Wilkes, J. Google cluster-usage traces: format + schema. *Google Inc., White Paper* (2011), 1–14.

[59] Ren, K., Qin, J., Zheng, L., Yang, Z., Zhang, W., Qiu, L., and Yu, Y. Deep recurrent survival analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 4798–4805.

[60] Salinas, D., Flunkert, V., Gasthaus, J., and Januschowski, T. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting 36*, 3 (2020), 1181–1191.

[61] Sen, R., Yu, H.-F., and Dhillon, I. Think globally, act locally: A deep neural network approach to high-dimensional time series forecasting. *arXiv preprint arXiv:1905.03806* (2019).

[62] Senderovich, A., Beck, J. C., Gal, A., and Weidlich, M. Congestion graphs for automated time predictions. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 4854–4861.

[63] Sommers, J., and Barford, P. Self-configuring network traffic generation. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement* (2004), pp. 68–81.

[64] Taylor, S. J., and Letham, B. Forecasting at scale. *The American Statistician 72*, 1 (2018), 37–45.

[65] Tirmazi, M., Barker, A., Deng, N., Haque, M. E., Qin, Z. G., Hand, S., Harchol-Balter, M., and Wilkes, J. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–14.

[66] Van Oord, A., Kalchbrenner, N., and Kavukcuoglu, K. Pixel recurrent neural networks. In *International Conference on Machine Learning* (2016), PMLR, pp. 1747–1756.

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

[68] Verma, A., Korupolu, M., and Wilkes, J. Evaluating job packing in warehouse-scale computing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)* (2014), pp. 48–56.

[69] Wamba, G. M., Li, Y., Orgerie, A.-C., Beldiceanu, N., and Menaud, J.-M. Cloud workload prediction and generation models. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (2017), IEEE, pp. 89–96.

[70] Wu, S., Xiao, X., Ding, Q., Zhao, P., Wei, Y., and Huang, J. Adversarial sparse transformer for time series forecasting. *Advances in Neural Information Processing Systems 33* (2020).

[71] Zhang, Q., Liu, V., Zeng, H., and Krishnamurthy, A. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference* (2017), pp. 78–85.

[72] Zhang, Q., Yang, L. T., Yan, Z., Chen, Z., and Li, P. An efficient deep learning model to predict cloud workload for industry informatics. *IEEE transactions on industrial informatics 14*, 7 (2018), 3170–3178.

[73] Zhou, H., Zhang, S., Peng, J., Zhang, S., Li, J., Xiong, H., and Zhang, W. Informer: Beyond efficient transformer for long sequence time-series forecasting. *arXiv preprint arXiv:2012.07436* (2020).

[74] Zoph, B., and Le, Q. V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

[75] Zou, H., and Hastie, T. Regularization and variable selection via the elastic net. *Journal of the royal statistical society: series B (statistical methodology) 67*, 2 (2005), 301–320.