# Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation

Olga Poppe, Tayo Amuneke, Dalitso Banda, Aritra De, Ari Green, Manon Knoertzer, Ehi Nosakhare, Karthik Rajendran, Deepak Shankargouda, Meina Wang, Alan Au, Carlo Curino, Qun Guo, Alekh Jindal, Ajay Kalhan, Morgan Oslake, Sonia Parchani, Vijay Ramani, Raj Sellappan, Saikat Sen, Sheetal Shrotri, Soundararajan Srinivasan, Ping Xia, Shize Xu, Alicia Yang, Yiwen Zhu
Microsoft, One Microsoft Way, Redmond, WA 98052
firstname.lastname@microsoft.com

## ABSTRACT

Microsoft Azure is dedicated to guarantee high quality of service to its customers, in particular, during periods of high customer activity, while controlling cost. We employ a Data Science (DS) driven solution to predict user load and leverage these predictions to optimize resource allocation. To this end, we built the Seagull infrastructure that processes per-server telemetry, validates the data, trains and deploys ML models. The models are used to predict customer load per server (24h into the future), and optimize service operations. Seagull continually re-evaluates accuracy of predictions, fallback to previously known good models and triggers alerts as appropriate. We deployed this infrastructure in production for PostgreSQL and MySQL servers across all Azure regions, and applied it to the problem of scheduling server backups during low-load time. This minimizes interference with user-induced load and improves customer experience.

**PVLDB Reference Format:**
Poppe et al. Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation. PVLDB, 14(2): 154 - 162, 2021.
doi:10.14778/3425879.3425886

## 1 INTRODUCTION

Microsoft Azure, Google Cloud Platform, Amazon Web Services, and Rackspace Cloud Servers are the leading cloud service providers that aim to guarantee high quality of service to their customers, while controlling operating costs [27, 38]. Achieving these conflicting goals manually is labor-intensive, time-consuming, error-prone, neither scalable, nor durable. Thus, these providers shift towards automatically managed services. To this end, Data Science (DS) techniques are deployed to predict resource demand and leverage these predictions to optimize resource allocation [14].

**Motivation**. Backups of databases are currently scheduled by an automated workflow that does not take typical customer activity patterns into account. Thus, backups often collide with peaks of customer activity resulting in inevitable competition for resources and poor quality of service during backup windows. To solve this problem currently, an engineer plots the customer load per database per week and manually sets the backup window during low customer activity. However, this solution is neither scalable to millions of customers, nor durable since customer activity varies over time. More recently, customers can select a backup window themselves. However, they may not know the best time to run a backup. Instead of these manual solutions, DS techniques could be deployed to predict customer load. These predictions could then be leveraged to schedule backups during expected low customer activity.

An infrastructure that analyses historical load per system component, predicts its future load, and leverages these predictions to optimize resource allocation is valuable for many products and use cases. Over the last two years we have built such an infrastructure, called Seagull, and applied it to two scenarios: (1) Backup scheduling of PostgreSQL and MySQL servers and (2) Preemptive auto-scale of SQL databases. These scenarios required us to battle-test the infrastructure across all Azure regions and gave us confidence on the high impact and generality of the Seagull approach.

**Challenges**. While building the Seagull infrastructure, we tackled the following open challenges.

• *Design of an end-to-end infrastructure* that predicts resource utilization and leverages these predictions to optimize resource allocation. This infrastructure must be: (a) Reusable for various scenarios and (b) Scalable to millions of customers worldwide.

• *Implementation and deployment of this infrastructure to production* to predict customer activity and schedule backups such that they do not interfere with customer load.

• *Accurate yet efficient customer low load prediction* for optimized backup scheduling. This challenge includes choice of an ML model that finds the middle ground between accuracy and scalability. In addition, prediction accuracy must be redefined to focus on predicting the lowest valley in customer CPU load that is long enough to fit a full backup of a server of its backup day. Accurate load prediction for the whole day is less critical for backup scheduling.

**State-of-the-Art Approaches**. Most of existing systems for ML lack easy integration with Azure compute [8, 11, 15, 16, 23, 28]. Thus, we built our solution based on Azure ML [4].

While time series forecast in general and load prediction in particular are well studied topics, none of the state-of-the-art approaches focused on predicting the lowest valley in customer CPU load for optimized backup scheduling. Instead, existing approaches focus on, for example, idle time detection for predictive resource provisioning [26, 38], VM workload prediction for dynamic VM allocation [13, 14], and demand-driven auto-scale [18–22, 35, 36]. Thus, these approaches do not tackle the unique challenges of low


Proceedings of the VLDB Endowment, Vol. 14, No. 2 ISSN 2150-8097.
doi:10.14778/3425879.3425886

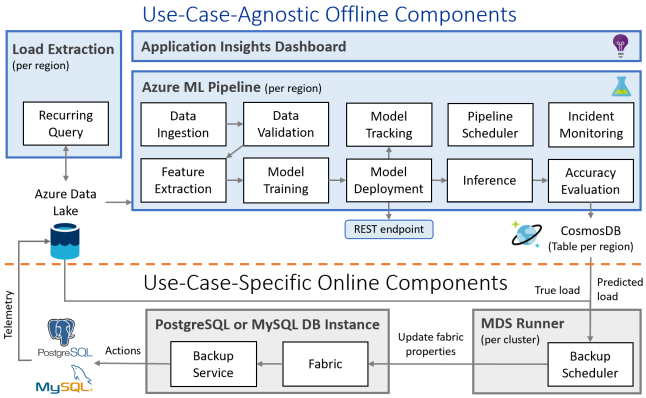



**Figure 1: Seagull infrastructure**

load prediction for optimized backup scheduling described above. In particular, they neither define the accuracy of low load prediction, nor compare several ML models with respect to low load prediction.

**Proposed Solution**. We built the Seagull infrastructure (Figure 1) that deploys DS techniques to predict resource utilization and leverages these predictions to optimize resource allocation. This infrastructure consumes prior load, validates this data, extracts features, trains an ML model, deploys this model to a REST endpoint, tracks the versions of all deployed models, predicts future load, and evaluates the accuracy of these predictions. We deployed Seagull to production worldwide to schedule backups of PostgreSQL and MySQL servers during time intervals of expected low customer activity. We achieved several hundred hours of improved customer experience across all regions per month.

**Contributions**. Seagull features the following innovations.

• We designed and implemented an end-to-end Seagull infrastructure and deployed it in all Azure regions to optimize backup scheduling of PostgreSQL and MySQL servers. We describe our design principles and lessons learned. We present our optimization techniques that reduce runtime and ensure scalability. We explain how to reuse the infrastructure for other scenarios. We evaluate the impact of the Seagull infrastructure on both improving customer experience and reducing engineering effort.

• We conducted comprehensive analysis to classify the servers into homogeneous groups based on their typical customer activity patterns. Majority of servers are either stable or follow a daily or a weekly pattern. Thus, the load per server on previous (equivalent) day is a strong predictor of the load per server today. This heuristic, called persistent forecast, correctly predicted the lowest load window per server on its backup day in 96% of cases.

• We defined the accuracy of low load window prediction as the combination of two metrics. One, the lowest load window is chosen correctly if there is no other window that is long enough to fit a full backup and has significantly lower average user CPU load. Two, the load during a lowest load window is predicted accurately if majority of predicted data points are within a tight acceptable error bound of their respective true data points.

• We applied several ML models commonly used for time series prediction (NimbusML [9], GluonTS [7], and Prophet [10]) to predict low load of unstable servers that do not follow a pattern that can be recognized by persistent forecast. We compared these models with respect to accuracy and scalability on real production data during one month in four Azure regions. Surprisingly, the accuracy of ML models is not significantly higher than the accuracy of persistent forecast. Thus, we deployed persistent forecast based on previous day to predict low load for all servers.

**Outline**. We present the Seagull infrastructure in Section 2. We classify the servers in Section 3. Section 4 defines low load prediction accuracy. Section 5 compares the ML models. We evaluate Seagull in Sections 6, summarize lessons learned in Section 7 and, review related work in Section 8. Section 9 concludes the paper.

## 2 SEAGULL INFRASTRUCTURE

In this section, we summarize our design principles, give an overview of the Seagull infrastructure, and describe how to reuse it.

### 2.1 Design Principles

**Modularity**. With the goal to reuse the Seagull infrastructure for various products and scenarios at Microsoft, we had to design it in a modular way. At the same time, we were determined to solve a specific task of optimized backup scheduling. To achieve both goals, we grouped the use-case-agnostic and use-case-specific components together (Figure 1). The use-case-agnostic components can be reused in several scenarios (Section 2.4). For example, any ML model can be plugged in. Nevertheless, the use-case-agnostic components often have to be adjusted to a particular data set, product, and scenario. For example, if the load of most servers is stable or conforms to a business pattern (Section 3), a simple heuristic can be used to predict the load. Complex ML models may not be needed (Section 5). However, the usage patterns may change over time. This observation justifies the need for a robust infrastructure that automatically detects these changes, notifies about them, and allows to easily replace the model.

**Scalability**. With the goal to deploy the Seagull infrastructure in all Azure regions, we had to ensure that it scales well for production data. Thus, we broke the input data down by region and ran a DS pipeline per region. Since the size of regions varies, the size of input files ranges from hundreds of kilobytes to a few gigabytes. Consequently, the runtime of a pipeline ranges from few minutes to few hours (Figures 7(a) and 8). We used Dask [6] to run time-consuming computations in parallel and achieved up to 4X speed-up compared to single-threaded execution (Figure 8(b)).

The choice of an ML model is determined not only by its accuracy but also by its scalability. For example, ARIMA [2] is computationally intensive since it searches the optimal values of six parameters to make an accurate load prediction per server. Parameter sharing among servers caused worsening of accuracy. While inference time is within a few seconds, fitting may take up to 3 hours per server. Hence, executing ARIMA in parallel for each server does not make runtime of ARIMA comparable to other models (Figure 7(a)). Thus, we excluded ARIMA from further consideration.

### 2.2 Use-Case-Agnostic Offline Components

The use-case-agnostic components consume the load per system component (e.g., database, server, VM) and apply ML models to predict future load of this component.

**Load Extraction Module** is implemented as a recurring query that extracts relevant data from raw production telemetry and stores this data in Azure Data Lake Store (ADLS) [3]. These files are input to the Azure ML pipeline.

For the backup scheduling scenario, we have selected the average customer CPU load percentage per five minutes as an indicator of customer activity. Other signals (memory, I/O, number of active connections, etc.) can be added to improve accuracy. In this paper, customer CPU load percentage per server is referred to as load per server for readability. Servers are due for full backup at least once a week. Thus, the load extraction query runs once a week per region.

**Azure ML Pipeline** is the core component of the Seagull infrastructure. It is built using the functionality of Azure Machine Learning (AML) [4] that facilitates end-to-end machine learning life cycle. This pipeline consumes the load, validates it, extracts features, trains a model, deploys the model, and makes it accessible through a REST endpoint. The pipeline tracks the versions of deployed models, performs inference, and evaluates the accuracy of predictions. Results are stored in Cosmos DB [5], globally distributed and highly available database service. Based on predicted load, resource allocation can be optimized in various ways.

In our case, the predictions are input to the backup scheduling algorithm. A run of the AML pipeline is scheduled once a week per region since servers are due for full backup at least once a week. Due to space limitations, we focus on the following five most interesting modules of the pipeline:

• ***Data Validation Module***. Since data validation is a well-studied topic [12], we implemented existing rules such as detection of schema and bound anomalies.

• ***Feature Extraction Module***. Lifespan and typical resource usage patterns are features that are useful for load prediction. In particular, we differentiate between short-lived and long-lived servers, stable and unstable servers, servers that follow a daily or a weekly pattern and servers that do not conform to such a pattern, predictable and unpredictable servers in Sections 3 and 4.2. We will extend this module by other features [32] to improve accuracy.

• ***Model Training and Inference Modules***. While many ML models can be plugged into the Seagull infrastructure, we compared NimbusML [9], GluonTS [7], and Prophet [10] with respect to accuracy and scalability. We applied these models to servers that cannot be accurately predicted by persistent forecast in Section 5.

• ***Accuracy Evaluation Module***. For backup scheduling, accurate load prediction for the whole day per server is less important than accurate prediction of lowest load window per day and server. Thus, we tailor prediction accuracy to our use case. We measure if the lowest load window is chosen correctly and if the load during this window is predicted accurately in Sections 3.1 and 4.

**Application Insights Dashboard** [1] provides summarized view of the pipeline runs to facilitate real-time monitoring and incident management. Examples of incidents include missing or invalid input data, errors or exceptions in any step of the pipeline.

## 2.3 Use-Case-Specific Online Components

The use-case-specific components leverage predicted load to optimize backup scheduling. The backup scheduler runs within Master Data Service (MDS) runner per day and cluster. The Runner Service deploys executables which probe their respective services resulting in measurement of availability and quality of service. The runner service is deployed in each Azure region.

For those servers that are due for full backups the next day, the backup scheduling algorithm verifies if these servers were predicted correctly for the last three weeks. In this way, we verify that the servers were predictable for several weeks and we do not reschedule a backup at a worse time based on predictions we are not confident in. Three weeks of history is a compromise between prediction confidence and relevance of this rule to the majority of servers (58% of servers survive beyond three weeks, Figure 3). For such predictable servers, the algorithm extracts the predicted load for the next day and selects a time window during which customer activity is expected to be the lowest. The algorithm stores the start time of this window as a service fabric property of respective PostgreSQL and MySQL database instances. This property is used by the backup service to schedule backups. Servers that did not exist or were unpredictable for the last three weeks are scheduled for backup at default time.

## 2.4 Reuse of Seagull for Other Scenarios

So far, we applied the Seagull infrastructure to two different scenarios: (1) Backup scheduling of PostgreSQL and MySQL servers and (2) Preemptive auto-scale of SQL databases. Based on this experience, we now summarize how to reuse the use-case-agnostic components of Seagull.

**No Changes**. All interfaces between the use-case-agnostic components, Model Deployment and Tracking are designed independently from any scenario and require no changes.

**Parameter Updates**. Data Ingestion and Validation, storage of results to CosmosDB, Pipeline Scheduler, Incident Management, and Application Insights Dashboard are parameterized to facilitate easy adjustment to a new scenario. For example, to account for changes of input data, we automatically deduce schema and other data properties (e.g., range of numeric attribute values) from the input data. The schema and data properties are stored in a file. After the file has been verified by a domain expert, it is used to detect schema and bound anomalies.

Other components require similar parameter updates. For example, Data Ingestion requires update of the location of input data in ADLS and access rights to this data. Also, the schema of CosmosDB tables, frequency of pipeline runs, and gathered statistics may be different for other scenarios.

**Adjustments**. Load Extraction, Feature Extraction, Model Training, Inference, and Accuracy Evaluation may require non-trivial customization. For example, other forecast signals (CPU, memory, disk, I/O, etc.) and features (subscriber identifier, number of active connections, etc.) may be needed for other scenarios. Accuracy and scalability of ML models heavily depend on the input data and scenario (Sections 3 and 5). Accuracy Evaluation may have to be tailored to the use case requirements (Section 4).

# 3 POSTGRESQL AND MYSQL SERVERS

In this section, we first define load prediction accuracy metric and then use this metric to measure if a server has stable load or follows a daily or a weekly pattern.

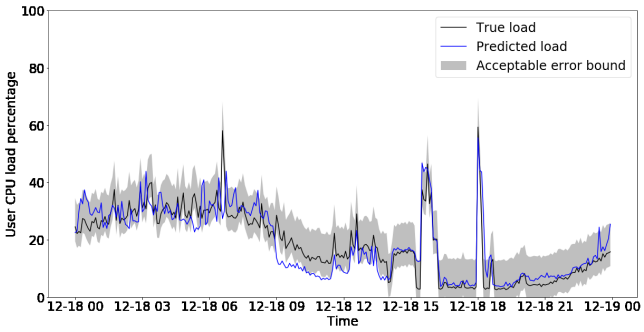



**Figure 2: Acceptable error bound**

## 3.1 Load Prediction Accuracy Metric

While there are several established statistical measures of prediction error (e.g., mean absolute scaled error and mean normalized root mean squared error), we found them unintuitive and cumbersome to use in our case. They produce a number representing prediction error per server per day. They give no insights into whether the lowest load window was chosen correctly per server per day nor whether the load was predicted accurately during this window. Thus, Definitions 3.2 and 4.2 below define these two metrics.

*Definition 3.1.* (**Acceptable Error Bound, Bucket Ratio Metric**) Given predicted and true load for a server $s$ during a time interval $t$, we define the *bucket ratio metric* for the server $s$ and the interval $t$ as the percentage of predicted data points that are within the *acceptable error bound* of +10/−5 of their respective true data points during $t$.

Definition 3.1 specifies an asymmetric error bound that tolerates up to 10% over-predicted load but only at most 5% under-predicted load because a slight overestimation of low load periods is less critical for our use case than a slight underestimation that may result in interference with high customer load. In Definitions 3.1–4.3, we plug in constants that were empirically chosen by domain experts and are now used in production for the backup scheduling use case. Other constants can be plugged in for other scenarios.

*Definition 3.2.* (**Accurate Load Prediction**) Prediction of the load of a server $s$ during a time interval $t$ is *accurate* if the bucket ratio of the server $s$ during the time interval $t$ is at least 90%. Otherwise, a prediction is *inaccurate*.

In Figure 2, we depict predicted load as blue line, true load as back line, and acceptable error bound as gray area. Intuitively, a prediction is accurate if 90% of the blue line is in the gray area. Even though the prediction looks close enough, the bucket ratio is only 75%. Thus, this prediction is inaccurate. This example illustrates that Definitions 3.1 and 3.2 impose strict constraints on accuracy.

## 3.2 Server Classification

We classify the servers with respect to typical customer behavior in Figure 3. The classification provides valuable insights about load predictability. We will leverage these insights while choosing the ML model in Section 5. Given a random sample of several tens of

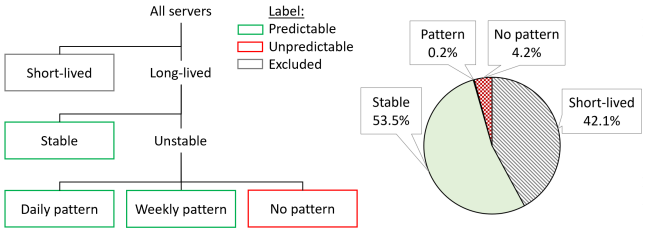



**Figure 3: Classification of servers**

thousands of servers from four regions during one month in 2019, Figure 3 summarizes the percentage of servers per class.

*Definition 3.3.* (**Short-Lived Server**) A server is called *long-lived* if it existed more than three weeks. Otherwise, a server is called *short-lived*.

As shown in Figure 3, 58% of servers survive for more than three weeks creating enough history to make a reliable conclusion whether they are predictable (Section 4.2). Remaining 42% of servers are short-lived. We exclude them from further consideration.

*Definition 3.4.* (**Stable Server**) A long-lived server is called *stable* during a time interval $t$ if its load is accurately predicted by its average load during the time interval $t$ (Definition 3.2). Otherwise, a server is called *unstable*.

53.5% of servers are long-lived and stable and thus easily predictable (Figure 3). 4.4% of long-lived unstable servers require a more detailed analysis. They are further classified into those that follow a daily or a weekly pattern and those that do not conform to such a pattern.

*Definition 3.5.* (**Server with Daily Pattern**) Given the load of a server $s$ on two consecutive days $d-1$ and $d$, the server $s$ has a *daily pattern* on day $d$ if its load on day $d$ is accurately predicted by its load on the previous day $d-1$. A server has a *daily pattern* during a time interval $t$ if its load conforms to this daily pattern on each day during the whole time period $t$.

Figure 4 shows an example of a server with a strong daily pattern. We plot the load on this day in black and on the previous day in blue. These lines overlap almost perfectly. The bucket ratio is 95%. Such a precise daily pattern could be the result of an automated recurring workload.

*Definition 3.6.* (**Server with Weekly Pattern**) Given the load of a server $s$ on two consecutive equivalent days of the week $d-7$ and $d$, the server $s$ has a *weekly pattern* on day $d$ if its load on day $d$ is accurately predicted by its load on the previous equivalent day of the week $d-7$. A server has a *weekly pattern* during a time interval $t$ if it does not have a daily pattern during the time period $t$ and its load conforms to a weekly pattern on each day during the whole time interval $t$.

Figure 5 shows an example of a server that follows a weekly pattern. Similarly to previous Sunday (December 1), the load on this Sunday (December 8) is medium before noon and high after noon. The bucket ratio is over 90%. In contrast, the load on previous day (December 7) is low before noon and medium after noon. The

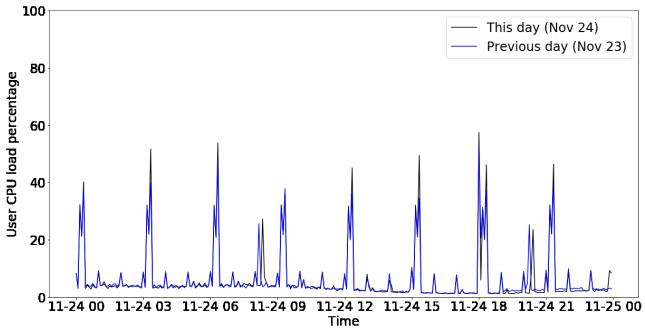



**Figure 4: Server with daily pattern**

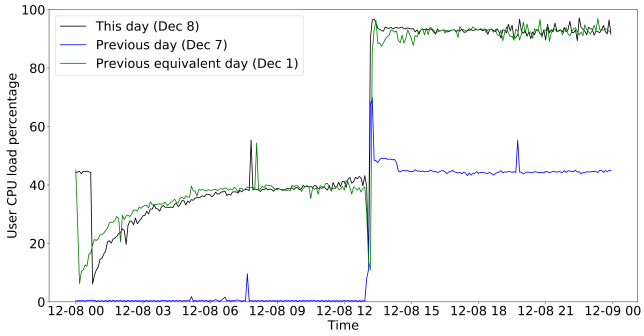



**Figure 5: Server with weekly pattern**

bucket ratio is only 1%. Thus, we conclude that this server follows a weekly pattern but does not conform to a daily pattern.

0.2% of servers conform to a daily or a weekly pattern and thus are easy to predict (Figure 3). Even though this percentage is relatively low, hundreds of top-revenue customers fall into this class of servers and cannot be disregarded.

**Summary**. Figure 3 illustrates that 53.7% of servers is expected to be predictable because their load is either stable or conforms to a pattern. 4.2% of the servers are neither stable nor follow a pattern. They are likely to be unpredictable. 42.1% are short-lived and thus excluded. These insights will be used while choosing the ML model to predict low load per server in Section 5.

# 4 LOW LOAD PREDICTION ACCURACY

In addition to the load prediction accuracy metric in Section 3.1, we define the lowest load window metric. Based on these metrics, we formulate the backup scheduling problem.

## 4.1 Lowest Load Window Metric

For each server on its backup day, our goal is to predict the lowest valley in the user load that is long enough to run a full backup of this server. The time interval of this valley is called the lowest load window. We measure if this window is chosen correctly and if the load during this window is predicted accurately. Accurate load prediction during the whole day is less critical in our case.

*Definition 4.1.* (**Lowest Load (LL) Window**) Let $s$ be a server which is due for full backup on day $d$. Let $b$ be the expected duration

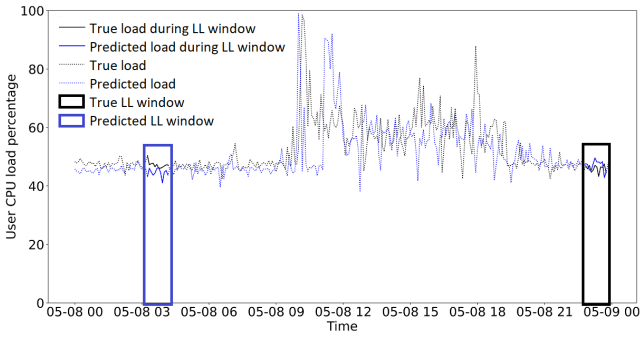



**Figure 6: Correctly chosen LL window**

of full backup of the server $s$. *True LL window* for the server $s$ on the day $d$ is the time interval of length $b$ during which the average true load of the server $s$ on the day $d$ is minimal across all other time intervals of length $b$ on the day $d$. *Predicted LL window* is defined analogously based on predicted load of the server $s$ on day $d$.

*Definition 4.2.* (**Correctly Chosen LL Window**) Let $w_t$ and $w_p$ be the true and predicted LL windows for a server $s$ on day $d$. If the average true load during the predicted LL window $w_p$ is within an acceptable error bound of the average true load during the true LL window $w_t$, then the predicted LL window $w_p$ is *chosen correctly*.

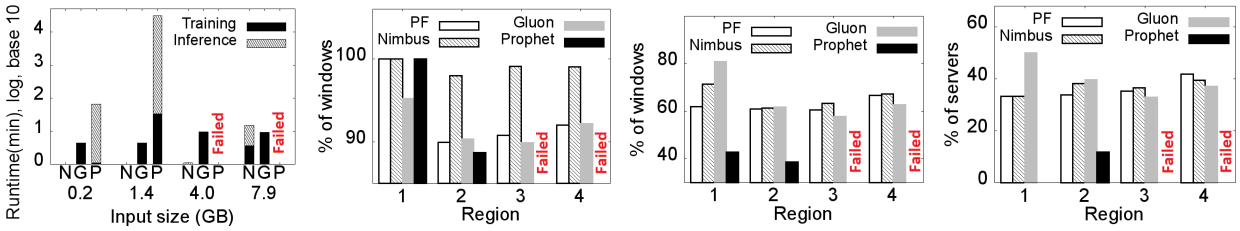
In Figure 6, the true and predicted LL windows do not overlap. However, the average true load during true LL window is only slightly lower than the average true load during predicted LL window. Thus, the true LL window would not be a significantly better time interval to run a backup than the predicted LL window. Hence, we conclude that the predicted LL window is chosen correctly.

## 4.2 Backup Scheduling Problem Statement

For each server $s$ that is due for full backup on day $d$, we aim to: (1) Correctly choose the LL window on day $d$ to schedule a backup during this LL window on day $d$ and (2) Accurately predict the load during this LL window to move a backup from default backup day to another day of the week if the load is lower on another day. These two metrics are orthogonal. Indeed, the true and predicted LL windows may coincide. However, the true load may be significantly higher than the predicted load [33]. The opposite case is also possible. Namely, the load may be predicted accurately during predicted LL window but the true load during the true LL window may be much lower than during the predicted LL window [33]. Based on these observations, we conclude that only both metrics combined give us reliable insights about low load prediction accuracy.

*Definition 4.3.* (**Predictable Server**) A long-lived server is called *predictable* if for the last three weeks its LL windows were chosen correctly and the load during these windows was predicted accurately (Definitions 3.2 and 4.2).

As explained in Section 2, we change backup window for predictable servers only. Servers that did not exist or were not predictable for three weeks, default to current backup time that is chosen independently from customer activity.




(a) Training and inference (b) LL windows (c) Load during LL windows (d) Predictable servers

**Figure 7: Low load prediction using Persistent Forecast (PF), NimbusML (N), GluonTS (G), and Prophet (P)**

# 5 LOW LOAD PREDICTION

In this section, we describe the ML models commonly used for time series forecast, choose a model per each class of servers, and compare the models with respect to their accuracy and scalability.

## 5.1 ML Models for Time Series Forecast

We now summarize the key ideas of the ML models that we considered to predict low customer activity.

**Persistent Forecast** refers to replicating previously seen load per server as the forecast of the load for this server. We compared three variations of persistent forecast:

• *Previous day* takes the load per server on the previous day and utilizes it as predicted load on the next day.

• *Previous equivalent day* forecasts the load of a server by replicating its load on previous equivalent day of the week.

• *Previous week average* uses the average load per server during previous week as predicted load per server.

**NimbusML** [9] is a Python module that provides Python bindings for ML.NET. NimbusML aims to enable data science teams that are more familiar with Python to take advantage of ML.NET's functionality and performance. It provides battle-tested, state-of-the-art ML algorithms, transforms, and components. Specifically, we use Singular Spectrum Analysis to transform forecasts.

**GluonTS** [7] is a toolkit for probabilistic time series modeling, focusing on deep learning-based models. We train a simple feed forward estimator. We tried several other estimators but this model achieved highest accuracy.

**Prophet** [10] is open source software released by Facebook. It forecasts a time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works well for time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend. It handles outliers well. We fit Prophet with daily seasonality enabled.

## 5.2 ML Model per Class of Servers

In this section, we discuss the applicability of each model in Section 5.1 to each class of servers in Section 3.2. We differentiate between two cases:

• *Stable servers and servers that follow business patterns that can be recognized by persistent forecast.* Obviously, such servers can be accurately predicted by persistent forecast and no complex ML models are needed. Indeed, the previous week average can predict the load of stable servers (Definition 3.4); 53.5% of servers are stable (Figure 3). Previous equivalent day is more powerful than previous week average because it captures a weekly pattern (Definition 3.6), including stable load which covers 53.6% of servers. Previous day is also more powerful than previous week average, since it captures a daily pattern (Definition 3.5), including stable load. 53.7% of servers can be predicted by the previous day's pattern. Since previous day is suitable for the largest subset of servers, we focus on this variant in the following.

• *Unstable servers that do not conform to a pattern that can be recognized by persistent forecast.* 4.2% of servers fall into this category. In Section 5.3, we apply ML models to such servers to find out if these models can detect a predictable load pattern for these servers.

## 5.3 Experimental Comparison of ML Models

*5.3.1 Experimental Setup.* We conducted all experiments on a VM running Ubuntu 18.04. This VM has 16 CPUs and 64GB of RAM.

**Input Data**. As described in Section 2, the pipeline runs per Azure region once a week. Thus, our input data is partitioned by region and week. Since the size of regions varies, the size of input files ranges from hundreds of kilobytes to a few gigabytes. Below, we randomly selected four input files with different sizes to demonstrate the scalability of ML models and determine if there are differences in accuracy of predictions between models and regions. The input files are in csv format. They contain server identifier, timestamp in minutes, average user CPU load percentage per five minutes, default backup start and end timestamps.

To identify predictable servers, we have to consider three weeks (Definition 4.3). To infer the load per server on its backup day, ML models are trained on one week of data prior to backup day per server. Thus, each input data set contains four weeks in one region, unless stated otherwise. We consider servers have at least three days of history prior to their backup days to train the ML models.

**Methodology**. We implemented the Seagull pipeline in Python. Our base-line implementation is *single-threaded*. Our *multi-threaded* Dask-based [6] implementation partitions the data per server and processes servers in parallel.

**Metrics**. For each ML model, we measure the percentage of correctly chosen LL windows, the percentage of LL windows with accurately predicted load, and the percentage of predictable servers among servers that existed at least three weeks (Definitions 3.2, 4.2, and 4.3). We measure the runtime of training, inference, and accuracy evaluation in minutes.

*5.3.2 Stable Servers and Servers with Pattern.* As explained in Section 5.2, majority of long-lived servers have stable load or follow daily or weekly patterns that can be recognized by persistent forecast. Therefore, we use persistent forecast to predict the load of such servers. For our sample data set, this heuristic correctly selected 99.83% of LL windows, accurately predicted the load during 99.06% of all windows, and classified 96.92% of servers as predictable.

*5.3.3 Unstable Servers Without Pattern.* We now apply ML models described in Section 5.1 to unstable servers that do not follow business patterns that can be recognized by persistent forecast.

**Training and Inference**. *Persistent forecast* does not require training because it uses the load per server on the previous day as predicted load per server on the next day.

*NimbusML* scales well (Figure 7(a)). Runtime for training and inference increases linearly from 2.5 seconds to 4 minutes as the number of servers grows from 10 to 700. Some of these measurements are not visible due to log scale.

*GluonTS* also scales well. Training time ranges from 4 to 10 minutes, while inference time ranges from 0.2 to 16 seconds as the number of servers grows from 10 to 700.

*Prophet* is not scalable. Its training time increases from 1 to 34 minutes, while inference time grows from 1 to 15 hours as the number of servers increases from 10 to 100. Thus, we implemented Prophet on Dask and achieved up to 60X speedup compared to single-threaded execution. However, when the number of servers exceeds 200, Prophet runs out of memory on Dask for any number of workers, while single-threaded execution does not terminate.

**Low Load Prediction Accuracy**. NimbusML correctly chooses the highest percentage of LL windows compared to other tools (Figure 7(b)). There is slight variance in accuracy of load prediction during LL windows and the percentage of predictable servers across regions and models (Figures 7(c) and 7(d)). Accuracy of persistent forecast, NimbusML, and GluonTS is comparable with respect to these two metrics. Prophet has similar or lower accuracy compared to the other models.

## 5.4 Choice of Model for Final Deployment

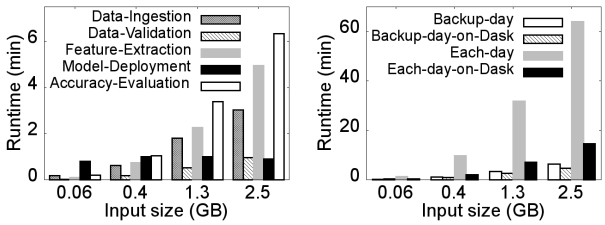
To find the middle ground between the accuracy of low load prediction and the overhead of model training and inference, we deployed persistent forecast based on previous day to production. This heuristic correctly selected 99% of low load windows, accurately predicted the load during 96% of all windows, and classified 75% of long-lived servers as predictable. The accuracy of other models is not significantly higher than the accuracy of persistent forecast. Persistent forecast does not introduce any computational delay due to training and thus scales better than other models. Lastly, it is easier to maintain a single model for the entire fleet of servers than a different model per each class of servers.

# 6 INFRASTRUCTURE EVALUATION

We now evaluate runtime, scalability, and impact of Seagull.

## 6.1 Runtime and Scalability

In Figure 8(a), we measure the runtime of the use-case-agnostic components per Azure region. These components are: Data Ingestion, Data Validation, Feature Extraction, Model Deployment, and




(a) All components (b) Accuracy evaluation

**Figure 8: Runtime and scalability evaluation**

Accuracy Evaluation. Runtime of Model Training and Inference per ML model are evaluated in Figure 7(a). Model Tracking, Pipeline Scheduler, and Incident Management run concurrently with other components and do not block the flow of the data through the AML pipeline. Thus, they are omitted in Figure 8(a). Since Load Extraction runs outside of the pipeline for all regions at once, it is also omitted. Load Extraction takes 30 minutes given the petabyte scale and complexity of raw telemetry.

In Figure 8, we measure the runtime for the same four regions of as in Figure 7(a). While Figure 7(a) considers four weeks to train ML models and infer future load, Figure 8 considers only one week which corresponds to our production settings (Section 2.2).

Model Deployment takes about one minute independently from deployed model. Runtime of other components increases linearly with growing input size. When input size exceeds 1GB, Accuracy Evaluation becomes a bottleneck. Thus, we partitioned input data per server and ran Accuracy Evaluation in parallel per server using Dask [6]. Figure 8(b) compares single-threaded and multi-threaded Accuracy Evaluation per server on its backup day. While Dask is 5 seconds slower than the single-threaded execution for 60MB, Dask consistently wins for input sizes over 400MB. For 2.5GB, Dask is 26% faster than single-threaded execution.

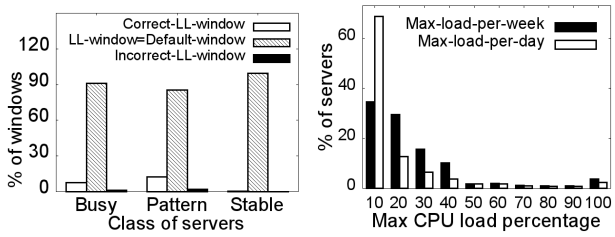
To further optimize backup scheduling, we will move a backup of a server from its default backup day to other day of the week if the load is lower and/or prediction is more accurate on another day. In Figure 8(b), we also measure the runtime of accuracy evaluation on each day one week ahead per server. Dask consistently achieves 3-4.6X speedup compared to the single-threaded implementation for all input sizes. For 2.5GB, the single-threaded implementation runs for over 1 hour, while Dask terminates after 15 minutes which is an acceptable computational delay for a large Azure region.

## 6.2 Impact and Future Work

The impact of the Seagull infrastructure is two-fold, namely, it improves customer experience and reduces engineering effort.

**Improved Customer Experience**. Seagull is deployed for tens of thousands of PostgreSQL and MySQL servers in tens of Azure regions to optimize backup scheduling. In Figure 9(a), we compare predicted LL windows (Definition 4.1) to default backup windows for all servers in all regions during one month in 2020.

For busy servers with customer load over 60% of capacity, 7.7% of backup collisions with peaks of customer activity are now avoided which corresponds to several hundred hours of improved customer




(a) Backup scheduling (b) Auto-scale

**Figure 9: Impact evaluation per use case**

experience. 91.1% of default windows are as good as LL windows on respective backup days. This happens by chance when default windows do not collide with high customer load. Only 1.2% of LL windows were not chosen correctly. This can be explained by unexpected change of customer behavior compared to the previous three weeks (Definition 4.3). For servers with predictable daily patterns (Definition 3.5), the results are similar. In particular, 12.5% of backups were moved from default windows that coincided with customer activity to correctly chosen LL windows (Definition 4.2). This percentage corresponds to several hundred hours of improved customer experience. As expected, for stable servers (Definition 3.4), 99.5% of default windows correspond to LL windows.

We also use the LL window metric (Section 4) to measure if backup windows selected by customers correspond to predictable LL windows and suggest windows with expected lower load instead.

While analysing the load, we concluded that many servers are not only predictable but also do not use the full capacity most of the time. Figure 9(b) illustrates the percentage of servers per maximal CPU load percentage of capacity per time unit. Only 3.7% of servers reach their CPU capacity per week, i.e., for 96.3% of servers resources could be saved. This observation opens up opportunities to overbook or auto-scale resources [26, 34]. We will explore these optimization techniques in follow-up projects to amplify impact.

**Reduced Engineering Effort**. Thanks to the automated load analysis enabled by Seagull, the engineers do not have to study customer behavior to select backup windows (Section 1). This manual approach was labor-intensive, time-consuming, neither scalable to millions of customers, nor durable since load patterns vary.

Based on the Seagull infrastructure, the time to setup a load prediction pipeline for other use cases came down from months to weeks. As described in Section 2.4, we applied Seagull to two scenarios so far. It took several months for a dedicated team of three software engineers, two data scientists, and a project manager to build, optimize, test, and deploy the Seagull infrastructure to production worldwide. However, it took only a few weeks to adjust this infrastructure to a new use case. In particular, we updated parameters of the use-case-agnostic components, re-implemented Load Extraction and Accuracy Evaluation, and hooked the predictions with the backup scheduling service.

## 7 LESSONS LEARNED

**Keep Version One Simple**. We originally started building the Seagull infrastructure to predict the load of several millions of SQL databases and enable preemptive auto-scale of resources. Since this was a complex and risky endeavor, we first tested the infrastructure on a *smaller fleet* of tens of thousands of PostgreSQL and MySQL servers and a *less risky scenario* of backup scheduling. We closely monitored, optimized, and adjusted the infrastructure since its deployment. Next, we will apply this matured system to more ambitious and risky scenarios and at higher scale.

**Verify Assumptions**. Building a reliable and scalable infrastructure like Seagull is challenging. Thus, it is important to verify all assumptions that such a project relies upon. For example, when we started building Seagull, the mechanisms that scale resources of SQL databases were slow. Thus, reactive auto-scale was unreliable and preemptive policies were needed. However, in the meantime, these mechanisms were optimized making even reactive auto-scale suitable for many databases. This example illustrates the need for a generic infrastructure to minimize loss of effort and amplify impact.

## 8 RELATED WORK

**Systems for ML** were proposed in the past. However, most of them lack easy integration with Azure compute [8, 11, 15, 16, 23, 28]. In particular, security, privacy, license, compatibility, and interfaces would have to be done from scratch. Thus, we built the Seagull infrastructure upon Azure products [1, 3–5]. They offer all features we needed to build Seagull. We also considered leveraging the model-serving infrastructure Resource Central [14]. However, at that time, AML [4] provided support and integration for a broader set of modeling and tracking tools.

**Load Prediction** for optimized resource allocation on a cluster has become a popular research direction in the recent years. Existing approaches focus on predicting survivability of databases for optimized resource provisioning [32], idle time detection for database quiescing and overbooking [26, 38], database workload prediction for database consolidation [17], VM workload prediction [24] for oversubscribing servers [14], dynamic VM provisioning [13], and reducing performance interference between VMs co-located on the same physical machine [30], workload classification for capacity planning and task scheduling [29], cost- and QoS-aware application placement in virtualized server clusters [37, 39], and preemptive auto-scale of resources [18–22, 25, 31, 34–36]. None of these approaches focused on predicting low load windows for optimized scheduling of system maintenance tasks. Thus, these approaches neither define low load prediction accuracy, nor compare ML models from the perspective of low load prediction.

## 9 CONCLUSIONS

We built the Seagull infrastructure for load prediction and optimized resource allocation on the cloud. While the infrastructure is applicable to a wide range of use cases, we illustrated it by the backup scheduling scenario.

## ACKNOWLEDGMENTS

The authors thank Akshaya Annavajhala, Purnesh Dixit, Larry Franks, Chris Lauren, and Santhosh Pillai for fruitful discussions about AML. We are grateful to Ashvin Agrawal and Hiren Patel for their help with large-scale telemetry analysis. We thank Matteo Interlandi, Siqi Liu, and Markus Weimer for their feedback.

## REFERENCES


[1] 2020. Application Insights. https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview.

[2] 2020. ARIMA. https://pypi.org/project/pmdarima/.

[3] 2020. Azure Data Lake Analytics. https://azure.microsoft.com/en-us/services/data-lake-analytics.

[4] 2020. Azure ML. https://azure.microsoft.com/en-us/services/machine-learning/.

[5] 2020. Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction.

[6] 2020. Dask. https://dask.org/.

[7] 2020. GluonTS. https://gluon-ts.mxnet.io/.

[8] 2020. MLflow. https://mlflow.org/.

[9] 2020. NimbusML. https://docs.microsoft.com/en-us/python/api/nimbusml/nimbusml.timeseries.ssaforecaster.

[10] 2020. Prophet. https://facebook.github.io/prophet/.

[11] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI*. 265–283.

[12] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *SysML*.

[13] Rodrigo Calheiros, Enayat Masoumi, R. Ranjan, and Rajkumar Buyya. 2014. Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications' QoS. *IEEE Transactions on Cloud Computing* 3 (08 2014), 449–458.

[14] 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 *SOSP*. 153–167.

[15] Daniel Crankshaw, Peter Bailis, Joseph E. Gonzalez, Haoyuan Li, Zhao Zhang, Michael J. Franklin, Ali Ghodsi, and Michael I. Jordan. 2015. The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox. In *CIDR*.

[16] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI*. 613–627.

[17] Carlo Curino, Evan Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-Aware Database Monitoring and Consolidation. In *SIGMOD*. 313–324.

[18] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. 2016. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In *SIGMOD*. 1923–1924.

[19] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. *SIGPLAN Not.* 49, 4 (Feb. 2014), 127–144.

[20] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-Regulating Stream Processing in Heron. In *Proc. VLDB Endow.* 1825–1836.

[21] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. 2010. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *TNSM*. 9–16.

[22] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. 2012. Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud. *Future Generation Comp. Syst.* 28 (01 2012), 155–162.

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *MM*. Association for Computing Machinery, 675–678.

[24] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. 2012. Workload Characterization and Prediction in the Cloud: A Multiple Time Series Approach. In *IEEE Network Operations and Management Symposium*. 1287–1294.

[25] Cinar Kilcioglu, Justin M. Rao, Aadharsh Kannan, and R. Preston McAfee. 2017. Usage Patterns and the Economics of the Public Cloud. In *WWW*. 83–91.

[26] Willis Lang, Karthik Ramachandra, David J. DeWitt, Shize Xu, Qun Guo, Ajay Kalhan, and Peter Carlin. 2016. Not for the Timid: On the Impact of Aggressive over-Booking in the Cloud. *Proc. VLDB Endow.* 9, 13 (2016), 1245–1256.

[27] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: Comparing Public Cloud Providers. In *IMC*. 1–14.

[28] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7.

[29] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. 2010. Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters. *SIGMETRICS Perform. Eval. Rev.* 37, 4 (March 2010), 34–41.

[30] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *USENIX ATC*. 219–230.

[31] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. 2009. Automated Control of Multiple Virtualized Resources. In *EuroSys*. 13–26.

[32] Jose Picado, Willis Lang, and Edward C. Thayer. 2018. Survivability of Cloud Databases - Factors and Prediction. In *SIGMOD*. 811–823.

[33] Olga Poppe, Tayo Amuneke, Dalitso Banda, Aritra De, Ari Green, Manon Knoertzer, Ehi Nosakhare, Karthik Rajendran, Deepak Shankargouda, Meina Wang, Alan Au, Carlo Curino, Qun Guo, Alekh Jindal, Ajay Kalhan, Morgan Oslake, Sonia Parchani, Vijay Ramani, Raj Sellappan, Saikat Sen, Sheetal Shrotri, Soundararajan Srinivasan, Ping Xia, Shize Xu, Alicia Yang, and Yiwen Zhu. 2020. Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation. https://arxiv.org/abs/2009.12922. Technical report.

[34] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *CLOUD*. 500–507.

[35] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *SOCC*. 1–14.

[36] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. 2018. P-Store: An Elastic Database System with Predictive Provisioning. In *SIGMOD*. 205–219.

[37] Akshat Verma, Puneet Ahuja, editor="Issarny Valérie Neogi, Anindya", and Richard Schantz. 2008. pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems. In *Middleware*. 243–264.

[38] Lalitha Viswanathan, Bikash Chandra, Willis Lang, Karthik Ramachandra, Jignesh M. Patel, Ajay Kalhan, David J. DeWitt, and Alan Halverson. 2017. Predictive Provisioning: Efficiently Anticipating Usage in Azure SQL Database. In *ICDE*. 1111–1116.

[39] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *ISCA*. 607–618.