Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless

Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan
Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA
firstname.lastname@microsoft.com

ABSTRACT

Microsoft Azure SQL Database is among the leading relational database service providers in the cloud. Serverless compute automatically scales resources based on workload demand. When a database becomes idle its resources are reclaimed. When activity returns, resources are resumed. Customers pay only for resources they used. However, scaling is currently merely reactive, not proactive, according to customers' workloads. Therefore, resources may not be immediately available when a customer comes back online after a prolonged idle period. In this work, we focus on reducing this delay in resource availability by predicting the pause/resume patterns and proactively resuming resources for each database. Furthermore, we avoid taking away resources for short idle periods to relieve the back-end from ineffective pause/resume workflows. Results of this study are currently being used worldwide to find the middle ground between quality of service and cost of operation.

PVLDB Reference Format:
Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless. PVLDB, 15(6): 1279-1287, 2022.
doi:10.14778/3514061.3514073

1 INTRODUCTION

Microsoft Azure SQL Databases [5], Google Cloud SQL Databases [13], and Amazon RDS for SQL Server [3] are the leading relational database service providers in the cloud. They deploy automatic, fully managed databases to guarantee high Quality of Service (QoS) to their customers, while controlling Cost of Goods Sold (COGS).

Azure SQL Database serverless automatically scales resources based on demand and bills for the amount of resources used per second [7]. However, resumes and pauses are currently merely reactive, meaning that they do not take typical resource usage patterns into account. Therefore, serverless compute can introduce delays in resource availability after idle periods. Consequently, serverless compute may be less suitable for time-critical applications than provisioned compute that allocates a fixed amount of resources [6].

In this work, we aim to overcome the reactive nature of serverless compute by proactively resuming resources based on historical resume patterns. Furthermore, if pauses are short, the availability time of resources is too fragmented for effective reuse. Thus, we aim to relieve the back-end from short pauses.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514073

Challenges. Optimizing Azure SQL Database serverless tier is a challenging endeavour for the following reasons.

(1) *Large search space of tunable parameters.* Proactive auto-scale of resources depends on many tunable parameters. The search space is prohibitively expensive to be explored exhaustively. Therefore, we identify trends of how these parameters influence the results and choose a reasonable set of parameters. The choice of parameters usually involves a trade-off between QoS and COGS [46]. For example, resuming resources in advance will guarantee high QoS, but waste COGS until these resources are used. We will discuss the trade-offs, while exploring the search space of parameters.

(2) *Opposing optimization objectives.* We aim to enable proactive resumes and avoid short pauses. However, these are opposing goals. Indeed, increasing the number of proactive resumes will also increase the number of wrong resumes, i.e., the customer did not come online as expected. Wrong resumes will in turn increase the number of pauses, some of which will be short. Also, reducing the number of short pauses will reduce the number of resumes, making the task of correct proactive resume harder because of fewer historical resumes. Solving this catch-22 is the goal of this work.

(3) *Changed resource usage patterns.* Resource usage patterns on serverless compute changed compared to provisioned compute. For example, provisioned databases are typically short-lived and often underutilized [33, 36, 39, 46]. In contrast to that, half of serverless databases existed over three weeks and half of idle periods are within a few hours (Figures 6 and 10(a)). At the same time, we observe certain similarities. For example, only a negligible percentage of provisioned or serverless databases follow a strict daily or weekly pattern. Therefore, we need to determine which lessons learned on provisioned compute can be transferred to serverless compute.

State-of-the-Art Techniques. While there are approaches to demand-driven auto-scale of resources in the cloud [27–30, 32, 37, 43–45], to the best of our knowledge, none of them addresses all challenges described above. In particular, they do not focus on achieving the contradictory goals of enabling proactive resume to guarantee high QoS, while reducing the number of short pauses to keep the operational costs low. Some of the existing approaches studied resource allocation on provisioned compute [26, 33, 36, 39, 40, 46] and we will transfer lessons learned from provisioned compute to serverless compute, when possible.

Our Proposed Solution. To address the challenges above, we propose the Moneyball approach that finds the middle ground between the contradictory goals of enabling proactive resume, while reducing the number of short pauses.[1]

[1] Similarly to the book and the movie "Moneyball" [1, 34], we apply statistical methods to achieve good results, while minimizing costs. However, we do not borrow statistical methods from this book.

To guarantee high QoS, we reduce delays in resource availability on login. To this end, we predict the pause/resume patterns and make recommendations when to proactively resume resources for each database. We compare probabilistic and predictive approaches to proactive resume and tune the key parameters to increase the number of correct proactive resumes, while reducing the operational costs due to wrong proactive resumes and the wait time intervals until the proactively resumed resources are used.

To reduce the back-end workload, we avoid short pauses. We compare two alternative solutions. One, we restrict the number of pauses per database and day (called a budget-based solution). Two, we introduce a wait time interval (called logical pause) before we scale resources down (called physical pause). We consider greedy and predictive approaches and compare their results to the optimal result. We tune the main parameters to reduce the number of short pauses and the costs due to idle resources during avoided pauses.

Contributions. Cloud service providers have recently evolved from provisioned to serverless compute [2, 7–9, 11, 14, 15, 17, 22, 23]. All of them face the challenge of provisioning resources only when needed. We believe that Moneyball generalizes to the cloud model in any company. Its key contributions are the following.

(1) We define the two-dimensional Moneyball problem space. We propose a visual way to compare our proposed solutions to the optimum and their impact on QoS and COGS.

(2) We summarize the main lessons learned during a decade of analysis of provisioned SQL databases. We transfer this learning to serverless compute, while solving the Moneyball problem. In particular, we select features and compare ML models to heuristics with respect to accuracy and maintenance overhead.

(3) We analyze production telemetry of serverless SQL databases with respect to their lifespan and typical resource usage patterns during half a year in tens of Azure regions where tens of thousands of serverless databases are currently deployed. Given the size and scope of this analysis, we believe that the usage patterns we observed represent the behaviors of any serverless databases.

(4) We enable proactive resume based on historical resume patterns per database. Up to 80% of resumes are proactive and correct within several hours for long-lived databases that existed at least 3 weeks. 99% of long-lived databases benefit from proactive resumes.

(5) We avoid short pauses by logically pausing a database that becomes idle before scaling its resources down. Logical pause is a simple, effective, and flexible technique that avoids up to half of pauses. 49% of databases benefit from this workload reduction.

2 MONEYBALL PROBLEM

Provisioned vs Serverless Compute. Resources of Azure SQL Databases are currently allocated in two ways.

Provisioned compute allocates a fixed amount of resources that does not change over time unless the customer explicitly requests a different amount [6]. Resources of a database s are resumed during the entire life time of s (Figure 1(a) and Table 1). However, rigorous telemetry analysis reveals that these resources are often underutilized [25, 26, 33, 36, 40, 46]. There are extensive idle periods during which resources are wasted unless customers manually scale resources down. This manual resource scaling is labor-intensive, time-consuming, error-prone, neither scalable, nor durable.

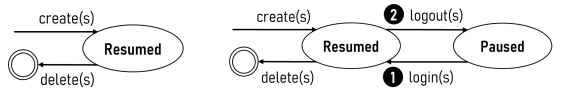
Table 1: Table of notations

Notation	Description
S	Set of databases, $s \in S$
d	Weekday (e.g., Wednesday)
W	Set of time windows within a day, $w \in W$
θ	Threshold
k	Budget
l	Duration of logical pause in hours
$H(s)$	Historical data of s
$h(s,d)$	Number of d's in $H(s)$
$r(s,d,w)$	Number of d's on which s was resumed during w in $H(s)$
$p(s,d,w)$	Probability of resume of s on d during w
$H(s,d,w)$	Historical data of s on d during w
$P(s,d)$	Predicted pause/resume pattern of s on d
$Predict(s,d)$	Time complexity of predicting the pause/resume pattern of s on d
$cost$	COGS per vCore per hour in dollars
$vcores(s)$	Maximum vCores of s
$pauses(s)$	Total duration of all pauses of s in hours without proactive resume
$wait(s)$	Total wait time in hours until proactively resumed resources of s are used
$avoided(s)$	Total duration of avoided pauses of s in hours
$allowed(s)$	Number of pauses of s that are longer than l
$idle(s,l)$	s is idle during l
$create(s)$	s is created
$delete(s)$	s is deleted
$login(s)$	Customer logs in to s
$logout(s)$	Customer logs out of s
$login(s).time$	Time stamp of $login(s)$
$p.start$	Start time stamp of a pause p

To overcome these limitations, *serverless compute* was recently introduced [7]. The resources of a database s are automatically scaled based on demand. If the serverless database is online, then scaling generally occurs with low latency since some resources remain allocated which helps mitigate the performance impact from compute warmup. However, if the serverless database is paused, then more or even all compute resources may be deallocated which elongates the latency to subsequently resume the database when workload activity returns. This paper focuses on minimizing the latency to resume a database since that has the greater performance impact between these two auto-scaling scenarios.

When the customer logs in, resources are resumed (❶ in Figure 1(b)). When the customer logs out, resources are paused for this database, reclaimed, and possibly assigned to other active databases (❷). Customers are billed per second only when resources are resumed for their databases. Thus, serverless compute minimizes both the waste of resources and the costs for the customers. However, serverless compute can be further optimized as described below.

Reactive vs Proactive Resume. Transitions between paused and resumed states are not instantaneous (Figure 1(b)). A resume workflow assigns resources to a database that becomes active, while



(a) Provisioned compute (b) Reactive auto-scale on serverless compute

Figure 1: Life cycle of a database

a pause workflow takes resources away from a database that becomes idle (Figure 2). Currently, resumes are merely reactive, not proactive. Delays in resource availability may occur after long idle periods. These delays make serverless compute less suitable for time-critical applications than provisioned compute [6].

In this work, we aim to reduce these delays by proactively resuming resources based on historical resume patterns per database. For example, if a database is usually resumed during a window w, we can proactively resume resources at the beginning of w.

Definition 2.1. (**Correct Proactive Resume**) A proactive resume of a database s within a window w is *correct* if the resources of s are used within w. Otherwise, a proactive resume is *wrong*.

However, if we proactively resume too far in advance, resources will stay idle until the customer logs in and uses them. COGS are wasted during these idle intervals. COGS are also wasted due to wrong proactive resumes. We aim to enable proactive resume, while keeping its operational cost low (Definition 4.6).

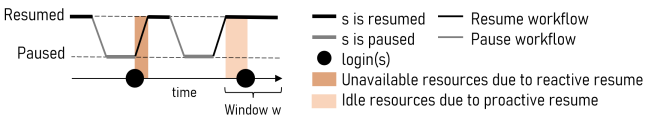


Figure 2: Reactive vs proactive resume

Effective vs Ineffective Pause. A pause is ineffective for short idle periods because the availability time of resources is too fragmented for effective reuse (Figure 3). We aim to relieve the back-end from frequent pause/resume workflows.

Definition 2.2. (**Ineffective Pause**) Given a threshold l, a pause is called *ineffective* if its duration is within l.

However, if we do not pause for long idle intervals, resources and COGS will be wasted. We aim to avoid up to half of pauses, while reducing the operational costs (Definitions 5.2 and 5.4).

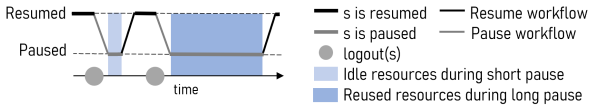


Figure 3: Effective vs ineffective pause

Moneyball Problem Statement. In this work, we aim to achieve the following three goals: (1) Maximize the number of correct proactive resumes, (2) Minimize the number of short pauses, and (3) Minimize the operational costs of these two optimization techniques.

3 TRANSFER LEARNING FROM PROVISIONED TO SERVERLESS COMPUTE

The resource usage patterns of provisioned Azure SQL Databases have been rigorously studied for over a decade [26, 33, 36, 39, 40, 46]. In this section, we briefly summarize the main lessons learned that can be helpful to solve the Moneyball problem and transfer this learning to serverless compute, when possible.

3.1 Features

Provisioned Compute. Historical load is an indicator of the future load per database. Some databases have stable load. Others follow a business pattern. At least three weeks of historical data are required to make a reliable load prediction [36, 40, 46]. Resource usage patterns may be different for databases with different editions (e.g., premium, standard), performance levels (i.e., number of Database Transaction Units (DTU) [6]), and Azure regions [33, 36, 39]. Furthermore, resource usage patterns may change over time.

Serverless Compute. To capture result variation between different regions and weeks, we analyzed half a year of production telemetry from tens of Azure regions where tens of thousands serverless databases are currently deployed. We included all features that can be useful for the prediction of pause/resume behavior in our analysis. They are: timestamp in seconds, database identifier, database state (1 means resumed, -1 means paused), duration of time intervals during which this database was resumed or paused, database compute capacity in maximum vCores, database creation and deletion timestamps, and Azure region.

3.2 ML Models

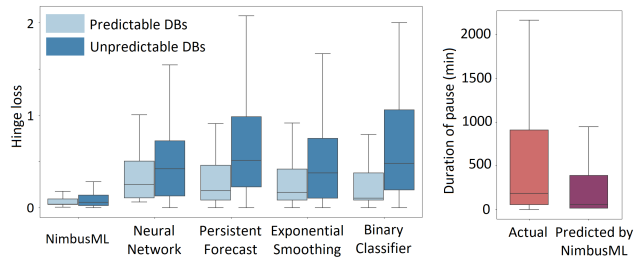
Provisioned Compute. In our prior research [40, 41], we predicted the load of provisioned databases using ARIMA [4], Prophet [21], NimbusML [18], Neural Network [12], Exponential Smoothing [10] and the Persistent Forecast heuristic that uses the load on a given day as the prediction of the load on next day per database. ARIMA and Prophet do not scale to tens of thousands of databases in large Azure regions. While NimbusML is the most accurate model, the gain in accuracy is not significant compared to Persistent Forecast because provisioned databases fall into one of the following extremes: (1) Most databases are *easily predictable* even by Persistent Forecast because their load is stable or follows a pattern (Definitions 3.1 and 3.2). (2) Remaining databases are *hard to predict* even by advanced ML models because their load tends to be random.

Serverless Compute. To verify that this conclusion holds for serverless databases, we classified them by their typical pause/resume patterns into the following groups.

Definition 3.1. (**Stable Database**) Given historical data $H(s)$ of a database s and a threshold θ, s is called *stable* if s is either resumed or paused at least $\theta\%$ of the time in $H(s)$. Otherwise, s is *unstable*.

Definition 3.2. (**Pattern**) Let s be an unstable database, $H(s)$ be the historical data of s, d be a weekday, w be a window, and θ be a threshold. s follows a *pattern* if at least $\theta\%$ of its resumes and pauses happen within the window w on each weekday d in $H(s)$.

Definition 3.3. (**Predictable Database**) A database s is called *predictable* if s is stable or follows a pattern. Otherwise, s is called *unpredictable*.



(a) Accuracy of ML models (b) Pause duration

Figure 4: Results of ML models

We randomly sampled several thousands of serverless databases in one Azure region. Similarly to provisioned databases, most serverless databases are predictable even under strict constraints. Indeed, 74% of serverless databases are stable at least 90% of the time. 3% follow a pattern within 15 minutes at least 90% of the time. Remaining 23% of databases are unpredictable.

We trained ML models on three weeks of historical data and predicted the load on the following day per database. Given that our input data is binary (Section 3.1), we measure hinge loss of NimbusML [18], Neural Network [12], Exponential Smoothing [10], ML.NET Binary Classifier [16], and Persistent Forecast for each class of databases in Figure 4(a). As expected, all models are more accurate for predictable databases than for unpredictable databases. Similarly to provisioned compute, NimbusML is the most accurate among these models for both classes of databases. Thus, we include NimbusML in our detailed analysis below.

4 PROACTIVE RESUME

In this section, we analyze resume patterns per database over time, make recommendations when to proactively resume a database, and evaluate the effectiveness of these recommendations.

4.1 Proactive Resume Algorithms

Example 4.1. The database in Figure 5 is unpredictable by Definition 3.3. However, a closer look reveals that this database is usually resumed between 5:40AM and 9:20AM on Wednesdays. These resumes are highlighted by red arrows. Only one expected resume is missing on 2/17. Next, we describe how to detect such recurring resumes and make them proactive.

Definition 4.2. (**Probability of Resume**) Let $H(s)$ be the historical data of a database s, $h(s,d)$ be the number of weekdays d in $H(s)$, and $r(s,d,w)$ be the number of d's on which s was resumed during a window w in $H(s)$ (Table 1). The *probability of resume* of s on d during w is computed as $p(s,d,w) = \frac{r(s,d,w)}{h(s,d)}$ [20].

Example 4.3. Given eight weeks of history in Figure 5, the probability of resume of this database s on Wednesday between 5:40 and 9:20 is $p(s, \textit{Wednesday}, [5{:}40, 9{:}20]) = \frac{7}{8} = 0.875$.

Definition 4.4. (**Probabilistic Resume Recommendation**) Given a threshold θ, we recommend to *proactively resume* a database s on a weekday d at the beginning of a window w if $p(s,d,w) \geq \theta$.

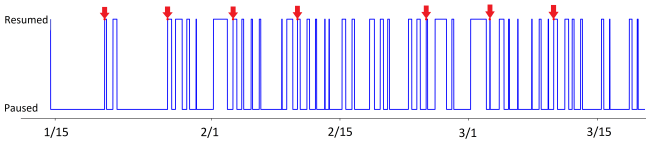


Figure 5: Recurring resumes

Probabilistic Resume computes resume recommendations R on a weekday d based on historical data of databases S. For each database $s \in S$ and window $w \in W$, Algorithm 1 adds a recommendation $[s,d,w]$ to proactively resume a database s on a weekday d at the beginning of a window w to the set of results R if the probability of resume $p(s,d,w)$ satisfies the threshold θ.

Complexity. Let $|S|$ be the number of databases in S, $|W|$ be the number of windows in W, and $|H(s,d,w)|$ be the number of tuples in historical data per database, weekday, and window. The time complexity of Algorithm 1 is $O(|S| \times |W| \times |H(s,d,w)|)$. Its space complexity is determined by the number of databases $|S|$, the number of tuples in historical data per database $|H(s)|$, and the number of recommendations per database in R. The number of results per database is in tern determined by the number of windows $|W|$. In summary, the space complexity is $O(|S| \times (|H(s)| + |W|))$.

Algorithm 1 Probabilistic proactive resume

Input: Historical data of databases S, set of windows W within one day, probability threshold θ
Output: Set of resume recommendations R on a weekday d

```
1: for each s ∈ S do
2:     for each w ∈ W do
3:         if p(s,d,w) ≥ θ then R ← R ∪ [s,d,w]
4: return R
```

Definition 4.5. (**Predictive Resume Recommendation**) Given the predicted pause/resume pattern $P(s,d,w)$ for a database s on a weekday d during a window w, we recommend to *proactively resume* s on d at the beginning of w if $\exists \textit{resume} \in P(s,d,w)$.

Predictive Resume algorithm is analogous to Algorithm 1 except that it consumes predicted pause/resume patterns and detects predicted resumes per Definition 4.5. While any ML model can be plugged into this algorithm to predict pause/resume patterns, we chose NimbusML since it is the most accurate model in Figure 4(a).

Complexity. Predictive resume introduces the overhead of predicting the pause/resume pattern per database and day, denoted $\textit{Predict}(s,d)$. Thus, the time complexity is $O(|S| \times (\textit{Predict}(s,d) + |W| \times |P(s,d,w)|))$. Predictive resume also stores the predicted pause/resume pattern per database and day, denoted $P(s,d)$. The space complexity is $O(|S| \times (|H(s)| + |P(s,d)| + |W|))$.

4.2 Middle Ground between QoS and COGS

While proactive resumes improve QoS, they also shorten pauses during which resources can be reused and COGS can be saved. Also, some proactive resumes will be wrong unavoidably. COGS are wasted due to wrong resumes as well. Definition 4.6 quantifies the operational cost of proactive resume.

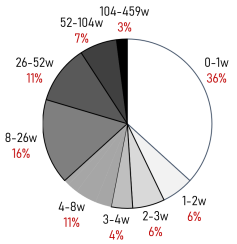
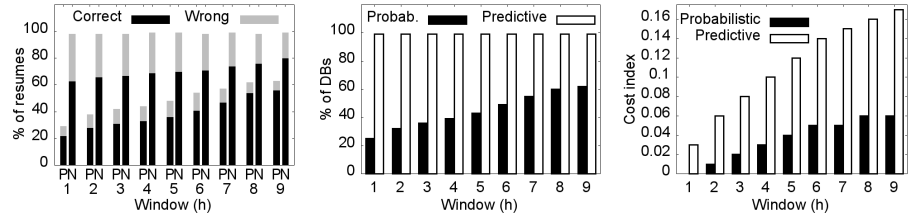


Figure 6: Lifespan



(a) Proactive resumes: Probabilistic resume (P) vs predictive resume (N)

(b) Benefited databases

(c) Resume cost index

Figure 7: Varying size of time window

Definition 4.6. (**Resume Cost Index**) Let $pauses(s)$ be the total duration of all pauses of a database s in hours without proactive resume. Let $vcores(s)$ be the maximum vCores of s and $cost$ be COGS per vCore per hour in dollars. The total cost savings are:

$$Total\ cost\ savings = \sum_{s \in S} pauses(s) \times vcores(s) \times cost \quad (1)$$

Let $wait(s)$ be the total wait time in hours until proactively resumed resources of a database s are used. The wasted cost is:

$$Wasted\ cost = \sum_{s \in S} wait(s) \times vcores(s) \times cost \quad (2)$$

Resume cost index corresponds to the ratio of the wasted cost to the total cost savings.

The cost index depends on several tunable parameters such as the size of the window and the length of historical data. We now experimentally find the middle ground between QoS and COGS, while enabling proactive resume.

In Figure 6, we measure the percentage of databases per their lifetime in weeks. Half of databases existed at least 3 weeks and thus have enough history to make a reliable prediction (Section 3).

Definition 4.7. (**Long-Lived Database**) A database is *long-lived* if it exists at least three weeks. Otherwise, it is *short-lived*.

Setup. In Figures 7–10 and 12, results are shown for several thousands of randomly sampled serverless long-lived databases in one Azure region. Unless stated otherwise, the length of historical (training) data is 3 weeks. The length of validation time interval is 1 day. Default size of the window is 5 hours. The window slides every 10 minutes. Default probability threshold is 0.9.

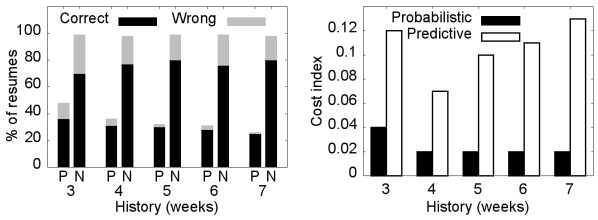
Size of the Window. In Figure 7, we vary the size of the window from 1 to 9 hours and measure the percentage of correct and wrong proactive resumes among all resumes (Definition 2.1), the percentage of database that have correct proactive resumes, and the resume cost index (Definition 4.6)

The percentages of correct resumes increase from 22 to 56 for probabilistic resume as the window grows in Figure 7(a). These percentages increase from 63 to 80 for predictive resume. Probabilistic resume benefits 25 to 62% of databases as the window grows in Figure 7(b). Independently from the size of the window, predictive resume benefits 99% of databases. The percentages of correct resumes and benefited databases is up to 3X higher for predictive resume than for probabilistic resume.

Unfortunately, the cost index also grows with the window since proactively resumed resources stay idle longer. Probabilistic resume has up to 5X fewer wrong resumes. Therefore, its cost index is up to 5X lower than the cost index of predictive resume in Figure 7(c).

Length of Historical Data. In Figure 8, we vary the length of historical data from 3 to 7 weeks. Given 3 weeks of history, 36% of resumes are proactive and correct, 43% of databases have correct proactive resumes, 12% of resumes are proactive and wrong, and the cost index is 4% for probabilistic resume. These percentages decrease as the length of historical data grows.

There is are no clear trends for the results of predictive resume across weeks. 70 to 80% of resumes are proactive and correct. 99% of databases have correct resumes. 18 to 29% of resumes are wrong. The cost index ranges from 7 to 12%. Similarly to Figure 7, predictive resume has up to 3X more correct resumes and benefited databases than probabilistic resume. Predictive resume also has up to 18X more wrong resumes. Thus, its cost index is up to 10X higher than the cost index of probabilistic resume.



(a) Proactive resumes: Probabilistic resume (P) vs predictive resume (N)

(b) Resume cost index

Figure 8: Varying length of historical data

Summary. Most resumes are proactive and correct within a few hours for long-lived databases. Most long-lived databases benefit from this QoS optimization. Cost index is low for short windows.

5 AVOIDING INEFFECTIVE PAUSES

Pauses are ineffective for short idle periods. Indeed, no COGS are saved and unnecessary pause/resume workloads are introduced. To alleviate these workloads from the back-end, we avoid ineffective pauses by restricting the number of pauses (called *budget*) and delaying pauses (called *logical pause*).

5.1 Budgeting Algorithms

One straightforward idea that comes to mind is to restrict the number of pauses per database and day and prioritize long pauses.

Definition 5.1. (**Budget**) *Budget* k is the number of allowed pauses per database s and window w.

A given budget can be spent in different ways as defined below.

Greedy Budget allows the first k pauses and avoids all following pauses per database and day. Let $logout_i(s).time$ and $login_{i+1}(s).time$ be the time stamps of two consecutive logout and login events for a database s. For each database s, Algorithm 2 stores the beginning $logout_i(s).time$ and the end $login_{i+1}(s).time$ of each avoided pause in the set of results R.

Complexity. Given the number of logouts $|logout(s,d)|$ for a database s on a weekday d, the time complexity of Algorithm 2 is $O(|S| \times |logout(s,d)|)$. Since avoided pauses are stored in the set of results R, the space complexity is $O(|S| \times (|logout(s,d)| - k))$.

Algorithm 2 Greedy budget

Input: Login/logout events of databases S on weekday d, budget k
Output: Set of avoided pauses R on a weekday d

```
1: for each s ∈ S do n ← 1
2:     for each logout_i(s) do
3:         if n > k then
4:             R ← R ∪ [s, logout_i(s).time, login_{i+1}(s).time]
5:         else n ← n + 1
6: return R
```

Algorithm 2 disregards the duration of avoided pauses. In the worst case, it spends the budget on early short pauses and avoids later long pauses which makes greedy budget expensive.

Definition 5.2. (**Pause Cost Index**) Let $avoided(s)$ be the duration of avoided pauses in hours for a database s. Other notations are summarized in Table 1. Then, the wasted cost is computed as:

$$Wasted\ cost = \sum_{s \in S} avoided(s) \times vcores(s) \times cost \quad (3)$$

The *pause cost index* is defined as the ratio of the wasted cost (Equation 3) to the total cost savings (Equation 1).

Predictive Budget prioritizes predicted long pauses over predicted short pauses, while spending the budget. For each database $s \in S$, Algorithm 3 consumes the predicted pause/resume pattern $P(s,d)$, extracts the longest k predicted pauses, and avoids all actual pauses that do not start within a given time delta δ of a long predicted pause.

Complexity. Predictive budget introduces the overhead of predicting the pause/resume pattern per database and day $Predict(s,d)$, sorting the predicted pauses by duration in $O(|P(s,d)| \log |P(s,d)|)$ time, and comparing the beginnings of k longest predicted pauses to the timestamps of actual logouts in $O(|logout(s,d)| \times k)$ time. The time complexity is $O(|S| \times (Predict(s,d) + |P(s,d)| \log |P(s,d)| + |logout(s,d)| \times k))$. Algorithm 3 stores the historical data, the predicted pause/resume pattern, and the avoided pauses per database. Its time complexity is $O(|S| \times (|H(s)| + |P(s,d)| + |logout(s,d)| - k))$.

Algorithm 3 Predictive budget

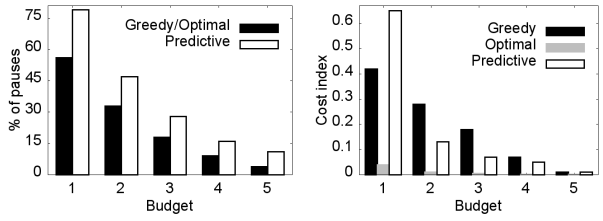
Input: Login/logout events of databases S on weekday d, predicted pause/resume patterns of S on d, budget k, window $w = 2 \times \delta$
Output: Set of avoided pauses R on a weekday d

```
1: for each s ∈ S do pauses ← getLongPauses(P(s,d), k), n ← 1
2:     for each logout_i(s) do
3:         if n > k or ∄p ∈ pauses such that
4:         logout_i(s).time − δ ≤ p.start ≤ logout_i(s).time + δ then
5:             R ← R ∪ [s, logout_i(s).time, login_{i+1}(s).time]
6:         else n ← n + 1
7: return R
```

Optimal Budget. To evaluate the effectiveness of the greedy and predictive budgets, we compare them to the optimal budget that avoids the top k shortest pauses per database.

Value of Budget. The percentages of avoided pauses and the cost index depend on the value of k which we vary in Figure 9. Greedy and optimal budget avoid 56 to 4% of pauses as budget grows from 1 to 5 in Figure 9(a). 56 to 2% of databases have avoided pauses for budget 1 to 5. While greedy budget disregards the duration of avoided pauses, optimal budget avoids the shortest k pauses per database and day. Thus, the cost of optimal budget is one order of magnitude lower than the cost of greedy budget in Figure 9(b).

Predictive budget delays pausing a database until long predicted pauses to reduce the time intervals during which resources are idle and COGS are wasted. However, if the start or duration of the longest k pauses per database and day are predicted wrong, then the predictive algorithm does not spend the available budget and avoids up to 5X more pauses than the greedy algorithm in Figure 9(a). Nevertheless, the cost of the predictive algorithm is up to 3X lower than the cost of the greedy algorithm for budget 2 to 4 because the predictive algorithm prioritizes long pauses while spending the budget (Figure 9(b)). Unfortunately, predictive budget does not guarantee lower cost compared to greedy budget. In fact, the cost of the predictive algorithm is 55% higher than the cost of the greedy algorithm for budget 1 in Figure 9(b).



(a) Avoided pauses (b) Pause cost index

Figure 9: Budget

Budget can be defined at different system granularities (e.g., per database, per tenant ring, per cluster) and for different windows (e.g., daily, weekly, monthly). However, we observed similar results to Figure 9. We do not consider global budget because resources are not shared across clusters in Azure.

Summary. Greedy budget disregards the duration of avoided pauses. Thus, its cost index is one order of magnitude higher than

the cost index of optimal budget. Predictive budget does not always spend the available budget and thus does not guarantee lower cost compared to greedy budget.

5.2 Logical Pause-Based Algorithms

Another simple idea is to wait for the customer to come back online before taking resources away from her database.

Definition 5.3. (**Logical Pause, Physical Pause**) Let $logout_i(s).time$ and $login_{i+1}(s).time$ be the time stamps of two consecutive logout and login events for s. Let t be the time stamp when resources are taken away from s such that $logout_i(s).time < t < login_{i+1}(s).time$. The time interval $(logout_i(s).time, t)$ is called *logical pause*. The time interval $[t, login_{i+1}(s).time)$ is called *physical pause*.

Greedy Logical Pause logically pauses a database s for the time interval l when the customer logs out.

Complexity. The time complexity of Algorithm 4 is the same as for Algorithm 2. Its space complexity is $O(|S| \times |logout(s,d)|)$.

Algorithm 4 Greedy logical pause

Input: Login/logout events of databases S on weekday d, duration of logical pause l
Output: Set of avoided pauses R on a weekday d

```
1: for each s ∈ S do
2:     for each logout_i(s) do
3:         if logout_i(s).time + l ≤ login_{i+1}(s).time then
4:             R ← R ∪ [s, logout_i(s).time, logout_i(s).time + l]
5:         else R ← R ∪ [s, logout_i(s).time, login_{i+1}(s).time]
6: return R
```

During logical pause, resources are still available in case the customer comes back online. In this way, we reduce delays in resource availability, while avoiding short pauses. However, resources are idle during avoided pauses (Line 5). In addition, pauses that are longer than l are shortened by greedy logical pauses (Line 4). This makes greedy pause expensive.

Definition 5.4. (**Greedy Pause Cost Index**) Let l be the duration of logical pause in hours, $avoided(s)$ be the duration of avoided pauses in hours for a database s, and $allowed(s)$ be the number of pauses of s that are longer than l. Other notations are summarized in Table 1. Then, the wasted cost is computed as follows:

$$Wasted\ cost = \sum_{s \in S} (avoided(s) + l \times allowed(s)) \times vcores(s) \times cost$$

The *greedy pause cost index* is defined as the ratio of the wasted cost to the total cost savings (Equation 1).

Predictive Logical Pause avoids predicted short pauses without reducing the duration of predicted long pauses. For each database $s \in S$, Algorithm 5 consumes the predicted pause/resume pattern $P(s,d)$ and computes the maximal duration of predicted pauses within the time delta δ of each logout. If this maximal duration is shorter than logical pause l, then this pause is avoided.

Complexity. Algorithm 5 predicts the pause/resume pattern per database and day and computes the maximal duration of predicted pauses that start within the time delta δ of each logout. Its time complexity is $O(|S| \times (Predict(s,d) + |logout(s,d)| \times |P(s,d)|))$. Algorithm 5 stores the historical data, the predicted pause/resume pattern, and the avoided pauses per database. Its space complexity is $O(|S| \times (|H(s)| + |P(s,d)| + |logout(s,d)|))$.

Algorithm 5 Predictive logical pause

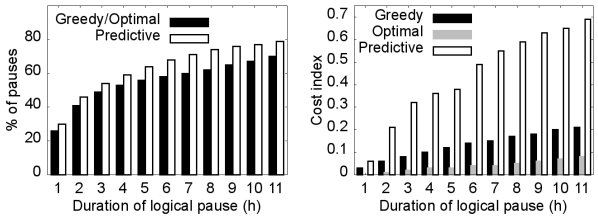
Input: Login/logout events of databases S on weekday d, predicted pause/resume patterns of S on d, duration of logical pause l, window $w = 2 \times \delta$
Output: Set of avoided pauses R on a weekday d

```
1: for each s ∈ S do
2:     for each logout_i(s) do
3:         w_i ← [logout_i(s).time − δ, logout_i(s).time + δ]
4:         maxDuraiton ← getMaxPauseDuration(P(s,d), w_i)
5:         if maxDuration < l then
6:             R ← R ∪ [s, logout_i(s).time, login_{i+1}(s).time]
7: return R
```

Optimal Logical Pause. To evaluate the effectiveness of the greedy and predictive logical pause, we compare them to the optimal logical pause that avoids all pauses that are shorter than l.

Duration of Logical Pause. The number of avoided pauses and the cost index depend on the duration of logical pause l that we vary in Figure 10. The greedy and optimal algorithms avoid 26 to 70% of pauses in Figure 10(a) and benefit 33 to 58% of databases as the duration of logical pause increases from 1 to 11 hours. The cost index of the greedy algorithm is up to 6X higher than the cost index of the optimal algorithm in Figure 10(b).

Since predicted pauses tend to be shorter than the actual pauses (Figure 4(b)), the predictive algorithm avoids up to 19% more pauses than the greedy algorithm in Figure 10(a). Thus, the cost index of the predictive algorithm up to 4X higher than the cost index of the greedy algorithm in Figure 10(b).



(a) Avoided pauses (b) Pause cost index

Figure 10: Logical pause

Summary. Greedy logical pause is a simple, flexible, and effective technique to avoid short pauses. Most databases benefit from this optimization technique at relatively low cost.

6 PUTTING IT ALL TOGETHER

In this section, we summarize how proactive resume and logical pause work together to proactively scale serverless databases. We also evaluate the impact of these optimization techniques.

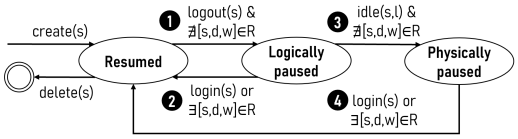


Figure 11: Proactive auto-scale on serverless compute

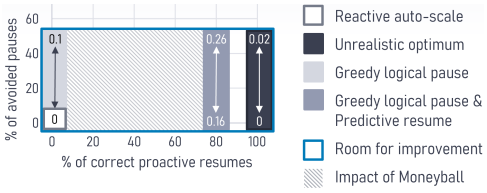


Figure 12: Moneyball problem space

Proactive Auto-Scale on Serverless Compute. Figure 11 formalizes the life cycle of a serverless proactively scaled database (compare to Figure 1). Let d be the current weekday and w be the current window. Other notations are summarized in Table 1.

The database s stays resumed as long as s is active or there is a recommendation to keep s proactively resumed on d during w, denoted $\exists[s, d, w] \in R$. If there is no such recommendation, then s is logically paused once the customer logs out (❶ in Figure 11). s stays logically paused for at most the time interval l. During logical pause, if the customer logs in or there is a recommendation to proactively resume s on d during w, then s is resumed (❷). If s stays idle during logical pause l and no resume is expected during on d during w, then s is physically paused (❸). s stays physically paused until s is resumed once the customer logs in or there is a recommendation to proactively resume s on d during w (❹).

Impact of Moneyball. Figure 12 illustrates the two-dimensional problem space where each dimension corresponds to the optimization technique enabled by Moneyball. X-axis represents the percentage of correct proactive resumes, while Y-axis depicts the percentage of avoided pauses. Rectangles represent alternative solutions and numbers correspond to their respective cost indexes.

In reactive approach, no resumes are proactive, no pauses are avoided, and thus no COGS are wasted (i.e., the cost index is 0). This case is shown as a white rectangle in Figure 12.

Ideally, all resumes are proactive and correct. In addition, up to half of pauses are avoided and these avoided pauses are the shortest to reduce resource idleness and wasted COGS. The cost index of the optimal solution is up to 0.02 (Definition 5.2). The range of these unrealistic optimal solutions is shown as a black rectangle. The area between the reactive approach and the optimal solution, highlighted by blue frame, is the potential room for improvement.

To avoid ineffective pauses, we introduce a wait time interval, called logical pause, before scaling resources down. Given that resources are idle during logical pauses, this solution wastes COGS. The number of avoided pauses and the cost index depend on the duration of logical pauses (Figure 10). For example, if logical pause is 4 hours, 53% of pauses are avoided and the cost index is 0.1 (Definition 5.4). The spectrum of logical-pause-based solutions is shown as a light gray rectangle.

Up to 80% of all resumes are proactive and correct within several hours for long-lived databases. Due to wrong resumes and wait time until the proactively resumed resources are used, the cost index is 0.16 (Definition 4.6). Combining proactive resume with logical pause makes up to 80% of resumes proactive and correct for long-lived databases, while still avoiding up to half of pauses. This combined Moneyball approach is shown as a dark gray rectangle. Its cost index is 0.26 which we consider to be reasonable cost for these optimization techniques. The striped area between the reactive approach and Moneyball represents the impact of this work.

7 RELATED WORK

Self-driving databases [19, 38] in general and demand-driven auto-scale of resources [26–30, 32, 37, 43–45] in particular have become popular research directions in the recent years. However, some of these state-of-the-art approaches are merely reactive [26–28]. In contrast, our Moneyball approach is proactive based on typical resource usage patterns per database (Section 4).

Some approaches avoid and mitigate under-estimation errors [44], reconfigure databases based on predicted load [45], or benchmark the efficiency of a cloud service [36]. These mechanisms are orthogonal to the Moneyball problem (Section 2).

Other approaches focus on load analysis [25, 32, 35, 42], load prediction using machine learning and other techniques [24, 29–31, 33, 36, 39, 40, 43, 46], or learning a relationship between available resources and performance [37]. We transferred learning from these approaches to Azure SQL Database serverless to solve the Moneyball problem (Sections 3–5).

While several state-of-the-art approaches focus on solving the trade-off between QoS and COGS in the cloud [27, 29, 33, 36, 39, 43, 44, 46], none of them achieves the contradictory goals of enabling proactive resume to guarantee high QoS and avoiding short pauses to alleviate this workload, while controlling operational costs at the same time. This is the key contribution of our Moneyball approach.

8 CONCLUSIONS

The Moneyball approach introduces two optimization techniques of Microsoft Azure SQL Database serverless. (1) To reduce delays in resource availability, we predict resume patterns per database over time and proactively resume resources. (2) To reduce the back-end workload, we avoid short pauses by logically pausing a database that becomes idle before scaling its resources down. We compared several algorithms and tuned their key parameters to keep the operational cost of these optimization techniques low. Results of this study are used in production in all Azure regions.

ACKNOWLEDGMENTS

The authors thank Ehi Nosakhare and Karthik Rajendran for their hard work predicting the load of provisioned SQL databases. Their findings guided our approach on serverless compute. We also thank Carlo Curino, Yiwen Zhu, and VLDB reviewers for their feedback.

REFERENCES

[1] 2011. *Moneyball (film).* Retrieved December 15, 2021 from https://en.wikipedia.org/wiki/Moneyball_(film)

[2] 2021. *Alibaba Cloud Function Compute.* Retrieved December 15, 2021 from https://www.alibabacloud.com/product/function-compute

[3] 2021. *Amazon RDS for SQL Server.* Retrieved December 15, 2021 from https://aws.amazon.com/rds/sqlserver

[4] 2021. *ARIMA.* Retrieved December 15, 2021 from https://pypi.org/project/pmdarima

[5] 2021. *Azure SQL Database.* Retrieved December 15, 2021 from https://azure.microsoft.com/en-us/products/azure-sql/database

[6] 2021. *Azure SQL Database pricing.* Retrieved December 15, 2021 from https://azure.microsoft.com/en-us/pricing/details/azure-sql-database

[7] 2021. *Azure SQL Database serverless.* Retrieved December 15, 2021 from https://docs.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview

[8] 2021. *CockroachDB Serverless.* Retrieved December 15, 2021 from https://www.cockroachlabs.com/lp/serverless/

[9] 2021. *Databricks Serverless SQL.* Retrieved December 15, 2021 from https://databricks.com/blog/2021/08/30/announcing-databricks-serverless-sql.html

[10] 2021. *Exponential Smoothing.* Retrieved December 15, 2021 from https://www.statsmodels.org/stable/generated/statsmodels.tsa.holtwinters.ExponentialSmoothing.html

[11] 2021. *Fauna Serverless.* Retrieved December 15, 2021 from https://fauna.com/serverless

[12] 2021. *GluonTS.* Retrieved December 15, 2021 from https://gluon-ts.mxnet.io/

[13] 2021. *Google Cloud SQL.* Retrieved December 15, 2021 from https://cloud.google.com/sql

[14] 2021. *Google Serverless Computing.* Retrieved December 15, 2021 from https://cloud.google.com/serverless

[15] 2021. *IBM Cloud Functions.* Retrieved December 15, 2021 from https://www.ibm.com/cloud/functions

[16] 2021. *ML.NET Binary Trainer.* Retrieved December 15, 2021 from https://docs.microsoft.com/en-us/dotnet/api/microsoft.ml.trainers.fasttree.fastforestbinarytrainer

[17] 2021. *MongoDB Serverless.* Retrieved December 15, 2021 from https://www.mongodb.com/cloud/atlas/serverless

[18] 2021. *NimbusML.* Retrieved December 15, 2021 from https://docs.microsoft.com/en-us/python/api/nimbusml/nimbusml.timeseries.ssaforecaster

[19] 2021. *Oracle Autonomous Database.* Retrieved December 15, 2021 from https://www.oracle.com/autonomous-database/

[20] 2021. *Probability theory.* Retrieved December 15, 2021 from https://en.wikipedia.org/wiki/Event_(probability_theory)

[21] 2021. *Prophet.* Retrieved December 15, 2021 from https://facebook.github.io/prophet

[22] 2021. *Serverless on AWS.* Retrieved December 15, 2021 from https://aws.amazon.com/serverless/

[23] 2021. *Snowflake Serverless.* Retrieved December 15, 2021 from https://docs.snowflake.com/en/user-guide/admin-serverless-billing.html

[24] Rodrigo Calheiros, Enayat Masoumi, Rajiv 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.

[25] 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.

[26] 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.

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

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] Michael Lewis. 2003. *Moneyball: The Art of Winning an Unfair Game.* W.W. Norton and Company.

[35] 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.

[36] Justin Moeller, Zi Ye, Katherine Lin, and Willis Lang. 2021. Toto - Benchmarking the Efficiency of a Cloud Service. In *SIGMOD*. 2543–2556.

[37] 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.

[38] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*.

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

[40] 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. *Proc. VLDB Endow.* 14, 2 (2020), 154–162.

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

[42] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *SOCC*. 1–13.

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

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

[45] 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.

[46] 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.