Moneyball: Proactive Auto-Scale of Microsoft Azure SQL Database Serverless at Low Operational Cost

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. It implements a win-win solution, namely, when a database becomes idle its resources are reclaimed and 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 analyzing typical resource usage patterns per database and proactively resuming resources. Furthermore, we avoid taking away resources for short idle periods to relieve the system 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.

1 INTRODUCTION

Microsoft Azure SQL Databases [4], Google Cloud SQL Databases [9], and Amazon RDS for SQL Server [2] 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 tier automatically scales resources based on workload demand and bills for the amount of resources used per second. The serverless compute tier automatically pauses resources during idle periods and automatically resumes resources when activity returns [6]. In this way, serverless compute implements a win-win solution in which customers are billed only for resources they used and idle resources are reclaimed.

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, the serverless compute may be less suitable for time-critical applications than provisioned compute that allocates a fixed amount of resources [5]. 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 system from ineffective pause/resume workflows.

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

(1) *Conflicting optimization objectives.* In this work we aim to maximize the number of proactive resumes, while at the same time avoiding ineffective pauses. However, these goals are in conflict with each other. Indeed, increasing the number of proactive resumes will unavoidably lead to higher number of wrong proactive 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 ineffective. Vice versa holds as well. Namely, reducing the number of short pauses will reduce the number of resumes, making correct proactive resume harder because of fewer historical resumes. Solving this catch-22 is the main target of this work.

(2) *Middle ground between QoS and COGS.* Many tunable parameters influence the results [30]. These parameters include length of historical data, probability thresholds, and size of time window. The search space is prohibitively expensive to be explored exhaustively. Thus, we identify trends how these parameters influence the results and choose a reasonable set of parameters. The choice of these parameters usually involves a trade-off between QoS and COGS [41]. For example, resuming resources in advance will guarantee high QoS but waste COGS until these resources are used. We will discuss these trade-offs, while exploring the search space of parameters.

(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 [27, 30, 33, 41]. In contrast to that, half of serverless databases existed over three weeks and half of idle periods are within seven hours (Figures 6 and 11(a)). At the same time, we observe certain similarities. For example, less than 1% of provisioned or serverless databases follow a strict daily or weekly pattern (Figure 5(a)). Therefore, we need to determine which lessons learnt 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 [21–24, 26, 31, 38–40], 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 ineffective pause/resume workflows to minimize COGS. Some of the existing approaches studied resource allocation on provisioned compute [20, 27, 30, 33, 34, 41] 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 ineffective pause/resume workflows.[1]

To guarantee high QoS, we reduce delays in resource availability as follows. First, we analyze typical resource usage patterns per database and compute probabilities of resume per database, time window, and usage pattern. Based on these probabilities, we make

[1] Similarly to the book and the movie "Moneyball" [12, 28], we apply statistical methods to achieve good results, while minimizing costs. However, we do not borrow statistical methods from this book.

recommendations when to proactively resume resources per database. One third of resumes are proactive and correct within a few hours. Two thirds of serverless databases have proactive resumes.

To reduce the system workload, we avoid ineffective pauses for short idle periods. To this end, we analyze the number and duration of pauses per database. Based on this analysis, 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 take away resources (called physical pause). Logical pause effectively avoids short pauses at reasonable additional cost.

Contributions. Several cloud service providers have recently evolved from provisioned to serverless compute [1, 10, 17]. They also face the challenge of provisioning resources only when they are 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. We vary features, the length of historical data, the size of time window, and the probability threshold to maximize the number of correct proactive resumes, while minimizing the impact on COGS due to wrong proactive resumes and wait time until the resumed resources are used.

(5) We reduce the back-end workload by avoiding pause/resume workflows for short idle periods. We analyze the number and duration of pauses per database. Based on this analysis, tune the duration of wait time interval before the resources are scaled down and the number of pauses per database and day to maximize the number of avoided short pauses, while minimizing COGS.

Outline. Section 2 defines the Moneyball problem. We transfer learning from provisioned to serverless compute in Section 3. We enable proactive resume in Sections 4 and avoid ineffective pauses in Section 5. Our approach is summarized in Section 6. We review related work in Section 7 and conclude the paper in Section 8.

2 MONEYBALL PROBLEM

Serverless vs Provisioned Compute. Resources of Azure SQL Databases are currently allocated in two ways. Table 1 summarizes the key differences between serverless and provisioned compute [6].

Provisioned compute allocates a fixed amount of resources that does not change over time unless the customer explicitly requests a different amount [5]. All provisioned resources are always available to the customer. However, rigorous telemetry analysis reveals that these resources are not fully utilized all the time [19, 20, 27, 30, 34, 41]. There are extensive idle periods and periods of low utilization 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.

To overcome these limitations, *serverless compute* was recently introduced [6]. It automatically scales resources based on demand. When the customer logs in, resources are resumed. When the database becomes idle, resources are paused for this database, reclaimed, and possibly assigned to other active databases to minimize waste of resources. Customers pay only during the time intervals when resources are resumed for their databases. Therefore, serverless compute implements a win-win solution that minimizes both waste of resources and costs for customers. However, serverless compute can be further optimized as described below.

Criteria	Serverless compute	Provisioned compute
Compute scaling	Demand-driven reactive auto-scale	Manual customer request
Compute responsiveness	Lower after inactive periods	Immediate
Idle resources	Paused	Wasted
Billing	Per second for used resources only	Per hour for all provisioned resources

Table 1: Serverless vs provisioned compute

Proactive vs Reactive Resume. Resources of a serverless database are either resumed or paused (Figure 1). A resume workflow assigns resources to a database that becomes active, while a pause workflow takes resources away from a database that becomes idle. Currently, resumes are merely reactive, not proactive. Unfortunately, resources cannot be reclaimed instantaneously when a customer logs in after a long idle period. Delays in resource availability may occur. These delays make serverless compute less suitable for time-critical applications than provisioned compute [5].

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 time window w every day, we can proactively resume resources at the beginning of this window w. Some of these proactive resumes will be correct and some may be wrong.

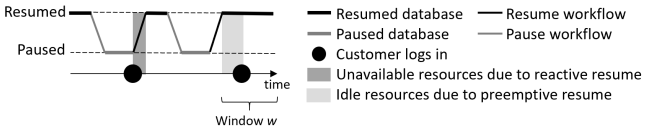
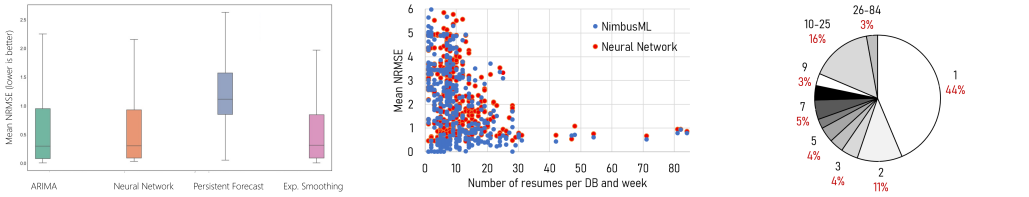


Figure 1: Proactive vs reactive resume

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

Resources are idle during the window w if a resume is wrong. Even for correct resume, resources are idle until the customer uses them. COGS are wasted during these idle intervals. We aim to enable proactive resume, while minimizing COGS (Definition 4.5).



(a) Provisioned compute [35] (b) Number of resumes vs Mean NRMSE (c) Number of resumes per DB & week

Figure 2: Accuracy of ML models

Effective vs Ineffective Pause. A pause is ineffective for short idle periods (Figure 3). Indeed, it is unlikely that another database on the same node will use the released resources for exactly the same short time interval. Resources will probably remain idle. Even if resources are used by another database, it is advantageous to relieve the system from frequent pause/resume workflows that reassign resources from one database to another and back again.

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 minimizing COGS (Definitions 5.2 and 5.5).

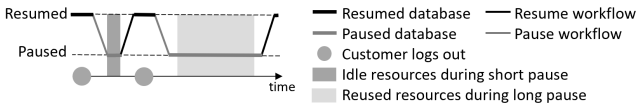


Figure 3: Effective vs ineffective pause

Moneyball Problem Statement. The goal of Moneyball is to maximize the number of correct proactive resumes and the number of avoided ineffective pauses, while minimizing COGS.

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 [20, 27, 30, 33, 34, 41]. 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 per database is an indicator of the future load per database. Some databases have stable load or follow an hourly, daily, or weekly pattern [30, 34, 41]. Their usage patterns on weekdays are often different from those on weekends [30]. At least three weeks of historical load are usually required to make a reliable load prediction [34, 41]. Unfortunately, short-lived databases may not have enough history. Interestingly, databases that belong to the same subscriber are often used in a similar way. Therefore, grouping short-lived databases by subscriber may allow predicting the load per database and subscriber [33].

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) [5]), and Azure regions [27, 30, 33]. Furthermore, resource usage patterns may change over time. Thus, solutions must be tailored to each edition, performance level, and region and adjusted over time.

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

3.2 ML Models

Provisioned Compute. ML models often do not achieve significantly higher accuracy compared to simple heuristics [20, 30, 34]. Indeed, ML models are prone to over-fitting. That is, while their accuracy is high for workloads they were trained on, the accuracy degrades significantly for unseen workloads [20]. In addition, some of the ML models (e.g., ARIMA [3] and Prophet [16]) are computationally expensive. They do not scale to tens of thousands of databases in large Azure regions. Parallel and distributed training and inference per database does not make the runtime of these ML models comparable to simple heuristics [34]. In summary, simple heuristics have easy-to-understand semantics and do not rely on external libraries. Thus, they are easy to explain, implement, debug, extend, and maintain long term in production [20, 30].

Serverless Compute. To verify that this conclusion holds for serverless databases, we compared the commonly used ML models for time series prediction (NimbusML [13], Neural Network [8], Exponential Smoothing [7], and LSTM [11]) to the persistent forecast heuristic that uses the load on a given day as the prediction of the load on next day for each database. To make results on serverless compute comparable to our previous study on provisioned compute (Figure 2(a)), we used a similar setup. Namely, we sampled thousands of serverless databases in the same region. For each database, we used one week of historical data to train the ML models and predicted the load on the following day. We measured the Mean Normalized Root Mean Square Error (Mean NRMSE).

As Figure **??** illustrates, the accuracy of ML models is not significantly higher than the accuracy of persistent forecast. Pause/resume patterns are hard to predict because the data is sparse. Indeed, Mean NRMSE tends to be low if the number of resumes exceeds 25 per database and week (Figure 2(b)). Unfortunately, only 3% of databases

have more than 25 resumes per week, while 55% of databases have only one or two resumes per week (Figure 2(c)). It is worth noting that low Mean NRMSE is often deceiving on such sparse data. For example, the database in Figure 4 is mostly paused. Thus, its Mean NRMSE is relatively low (3.09). But its resumes are not predictable.

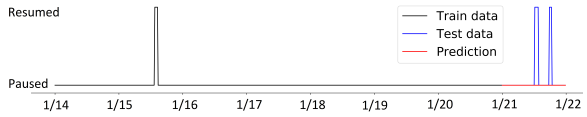


Figure 4: Load prediction using Neural Network [8]

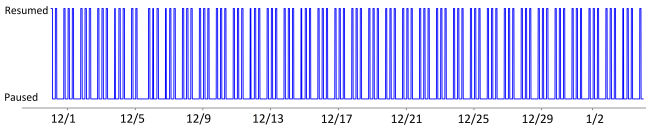
To make the training data less sparse, we increased its duration to several weeks but the difference in accuracy between ML models and persistent forecast stayed similar to Figure **??** [36]. Such low gain in accuracy does not justify the overhead of deployment and maintenance of ML models in production worldwide.

4 PROACTIVE RESUME

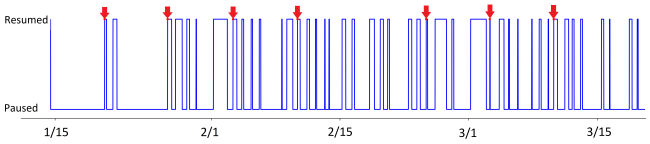
In this section, we analyze resume patterns per database over time, make recommendations when to proactively resume a database, and evaluate the effectiveness of these recommendations.

4.1 Resume Patterns

Even though the data is sparse and the percentage of databases that strictly follow predictable usage patterns is negligible (less than 1%), many databases have recurring resumes within several hours per day or per week.



(a) Strict daily pattern



(b) Loose weekly pattern

Figure 5: Resume patterns

Example 4.1. Figures 5(a) and 5(b) depict the pause/resume patterns of two databases. The database in Figure 5(a) has 103 pause/resume workflows during 5 weeks. It strictly follows a daily pattern since 95% of resumes and pauses happened within 5 minutes per day. Only one expected pause/resume workflow is missing on 12/5. Such precise pattern could be the result of an automated workload.

In contrast, the database in Figure 5(b) seems to be completely unpredictable at first sight. 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. Thus, we conclude that this database loosely follows a weekly resume pattern. Next, we describe how to detect such recurring resumes and make them proactive.

Definition 4.2. (**Probability of Resume**). Given historical data of a database d, let $h(d, days)$ be the number of all days, while be the number of days on which the database d was resumed during a time window w. Then, the *probability of resume* of the database d during the window w on the given days, denoted $p(d, w, days)$, is computed as the ratio of $r(d, w, days)$ to $h(d, days)$ [15].

To maximize the percentage of proactive resumes and their confidence, we differentiate between four resume patterns per database and time window in Definition 4.3.

Definition 4.3. (**Resume Patterns**). Let θ be a probability threshold. Other notation is summarized in Table 2. We say that a database d follows a *daily resume pattern* during a time window w if $p(d, w, Monday\ to\ Sunday) \geq \theta$. Analogously, a database d follows a *business resume pattern* during a time window w if $p(d, w, Monday\ to\ Friday) \geq \theta$. Similarly, a database d follows a *weekend resume pattern* during a time window w if $p(d, w, Saturday\ to\ Sunday) \geq \theta$. Lastly, a database d follows a *weekly resume pattern* during a time window w on a weekday if $p(d, w, weekday) \geq \theta$.

Example 4.4. Given 8 weeks of history and assuming the probability threshold $\theta = 0.85$, the database d in Figure 5(b) follows a weekly resume pattern since its probability of resume $p(d,$ [5:40, 9:20], *Wednesday*$) = \frac{7}{8} = 0.875$ exceeds the threshold θ.

Notation	Description
D	Set of serverless databases
d	Serverless database, $d \in D$
w	Time window (e.g., 8AM to 9AM)
θ	Probability threshold
l	Duration of logical pause in hours
$cost$	COGS per vCore per hour
$days$	Set of weekdays (e.g., Monday to Friday)
$h(d, days)$	Number of $days$ in historical data of d
$r(d, w, days)$	Number of $days$ on which d was resumed during w, $r(d, w, days) \leq h(d, days)$
$p(d, w, days)$	Probability of resume of d during w on $days$
$vcores(d)$	Maximum vCores of d
$pauses(d)$	Duration of pauses of d in hours
$unused(d)$	Wait time in hours until proactively resumed resources of d are used
$avoided(d)$	Duration of avoided pauses of d in hours
$allowed(d)$	Number of pauses of d that are longer than l

Table 2: Table of notations

4.2 Resume Recommendations

Proactive resumes take time away from pauses during which resources and COGS are saved. That is, time intervals during which a database is resumed become longer, while pauses become shorter.

Definition 4.5. (**Proactive Cost Index**). Let $pauses(d)$ be the duration of all pauses in hours of a database d without proactive resume. Let $vcores(d)$ be the maximum vCores of d and $cost$ be the maintenance cost per vCore per hour. The total cost savings are computed as follows:

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

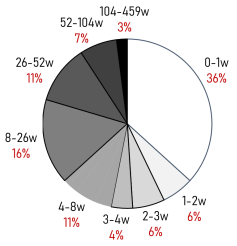
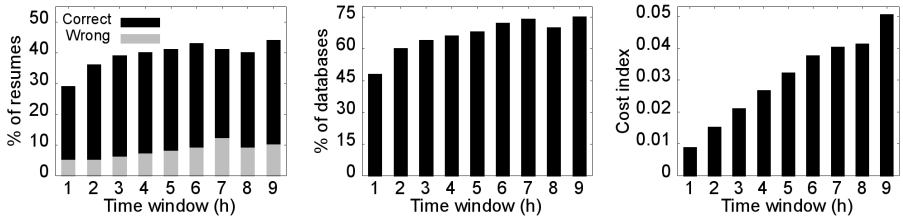


Figure 6: Lifespan



(a) Proactive resumes (b) Affected databases (c) Proactive cost index

Figure 7: Varying size of time window

Let $unused(d)$ be the wait time in hours until proactively resumed resources of a database d are used. The wasted cost is computed as follows:

$$Wasted\ cost = \sum_{d \in D} unused(d) \times vcores(d) \times cost \qquad (2)$$

The *proactive cost index* corresponds to the ratio of the wasted cost to the total cost savings.

In the following, we compute resume probabilities per resume pattern, time window, and database. Based on these probabilities, we then make recommendations when to proactively resume each database, while minimizing the proactive cost index.

Example 4.6. Consider the resume probabilities for a database and a window in Table 3. They reveal that this database was resumed during this window on all days, except Thursdays. The probability of resume on Thursday is only 0.66 which contributes to 0.93 resume probability on business days and 0.95 on all days of the week.

Resume pattern	Resume probability
Mo, Tu, We, Fr, Sa, Su, weekend	1
Th	0.66
business	0.93
daily	0.95

Table 3: Resume probabilities per resume pattern

Safe vs Aggressive Proactive Resume. Given a probability threshold and resume probabilities, we can make recommendations based on the most general resume pattern that satisfies the threshold and enable *aggressive proactive resume*. If the probability threshold is 0.95 in Example 4.6, we would recommend to resume this database daily during this window. In this way, we maximize the number of proactive resumes, while taking into account that some of these resumes may be wrong, e.g., on Thursdays (Definition 2.1).

Alternatively, we can make recommendations based on resume patterns with highest confidence and enable *safe proactive resumes*. In Example 4.6, we would recommend to resume this database on all days during this window, except Thursdays. In this way, we minimize the number of wrong resumes and the proactive cost index, while taking into account that some resumes will be reactive, rather than proactive, e.g., on Thursdays.

We experimentally compared these strategies and found that the numbers of correct and wrong proactive resumes are higher for aggressive proactive resume than for safe proactive resume. Unfortunately, additional wrong resumes negate additional correct resumes for aggressive proactive resume compared to safe proactive resume. Furthermore, aggressive proactive resume introduces wrong resumes and therefore increases the number of ineffective pause/resume workflows and the proactive cost index. Thus, we now focus on fine tuning safe proactive resume.

4.3 Middle Ground between QoS and COGS

In this section, we experimentally find the middle ground between QoS and COGS, while enabling safe proactive resume. Since at least 3 weeks of history are required to make a reliable load predication (Section 3), the length of historical data is 3 weeks and the length of validation time interval is 1 week. Thus, results are shown for several thousands of serverless databases that existed at least 4 weeks in one region, unless stated otherwise. By default, we show results for safe proactive resume with probability threshold 0.9 within a time window of 5 hours that slides every 10 minutes.

Lifespan. Figure 6 illustrates 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*.

Varying the Length of Historical Data. In Figure 8, we vary the length of historical data from 3 to 7 weeks and measure the percentage of correct and wrong proactive resumes among all resumes (Definition 2.1), the percentage of databases with proactive resumes, and the proactive cost index.

For 3 to 5 weeks of history, one third of resumes are proactive and correct. When the history length exceeds 5 weeks, the numbers of correct proactive resumes and affected databases significantly drop. Thus, we conclude that prioritizing recent history, improves the results. To enable proactive resume for majority of serverless databases (Figure 6), we require 3 weeks of history per database.

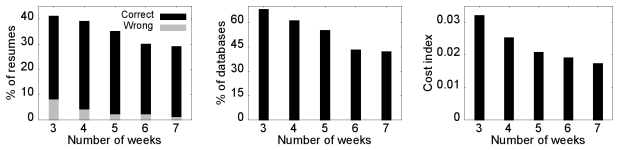
Varying the Size of Time Window. In Figure 7, we vary the window size from 1 to 9 hours. Larger window can increase the probability of resume and hence the numbers of proactive resumes and affected databases. However, the proactive cost index (Definition 4.5) is also expected to be higher since we may wait longer for the proactively resumed resources to be used.

As the window increases from 1 to 6 hours, the percentages of correct and wrong proactive resumes grow in Figure 7(a). One third of resumes are proactive and correct for the windows 3 to 6 hours which is the best trade-off between correct and wrong resumes. If the window is longer than 6 hours, the percentage of wrong

resumes increases, while the percentage of correct resumes stays about the same as for the windows 3 to 6 hours.

Similarly, the percentage of databases with proactive resumes grows from 48% to 74% as the window size increases from 1 to 7 hours in Figure 7(b). If the window is longer than 7 hours, the percentage of affected database drops to 70%.

The proactive cost index increases from 0.009 to 0.05 as the window grows from 1 to 9 hours in Figure 7(c). For 5-hour-long window, the proactive cost index is 0.03. We consider this cost acceptable for the positive impact of proactive resume on QoS.



(a) Proactive resumes (b) Affected databases (c) Proactive cost index

Figure 8: Varying length of historical data

Varying the Features. Given that half of serverless databases exist less than 3 weeks (Figure 6), we grouped short-lived serverless databases per subscriber and applied the Moneyball approach to each group since databases that belong to the same subscriber are often used in a similar way on provisioned compute (Section 3). Unfortunately, for short-lived serverless databases grouped by subscriber, the percentages of correct proactive resumes and affected databases are negligible compared to long-lived serverless databases in Figures 7 and 8. Most short-lived databases existed less than one week (Figure 6). Therefore, only few resume recommendations per subscriber are applicable to short-lived databases.

Varying the Probability Threshold. While Figures 7 and 8 focus on high-confidence proactive resumes with probability threshold 0.9, we also evaluated low-confidence proactive resumes with threshold 0.6 and concluded that wrong resumes negate additional correct resumes and almost double the proactive cost index [36]. Thus, we make high-confidence resume recommendations.

Varying the Azure Regions. Since results may vary across regions (Section 3), we compute the percentages of correct proactive resumes and affected databases for three randomly selected regions. However, we observed almost identical results [36]. Also, the proactive cost index is 0.03 for these regions. We draw on these results to apply the method and parameters across all regions.

Summary. One third of resumes are proactive and correct within 5 hours for long-lived databases. Two thirds of long-lived databases profit from this QoS optimization. Additional costs and the percentage of wrong proactive resumes are acceptable.

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 system, we avoid short pauses by restricting the number of pauses (called *budget*) and delaying pauses (called *logical pause*) in Sections 5.1 and 5.2, respectively. Unless stated otherwise, we show results for tens of thousands of serverless databases in one Azure region during two months.

5.1 Budget

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**). Let g be a system granularity (database, tenant ring, or cluster) and w be a time window. Then, *budget* is the number of allowed pauses per granularity g and window w.

Budgeting Algorithms. A given budget can be spent in different ways defined by the following budgeting algorithms.

Greedy Budget is our base-line solution that spends the budget on first come, first served basis. On the up side, this algorithm is simple. It guarantees that the given budget is always spent when possible. On the down side, it neglects the expected duration of pauses. Consequently, it can spend the budget on early short pauses and avoid later long pauses which makes greedy budget expensive.

Definition 5.2. (**Budget Cost Index**). Let $avoided(d)$ be the duration of avoided pauses in hours for a database d. Other notations are summarized in Table 2. Then, the wasted cost is computed as:

$$Wasted\ cost = \sum_{d \in D} avoided(d) \times vcores(d) \times cost \qquad (3)$$

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

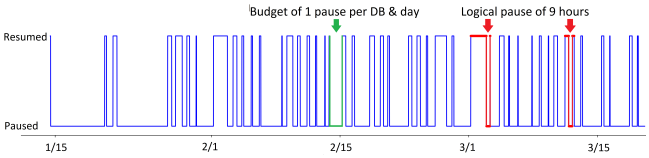


Figure 9: Avoiding short pauses

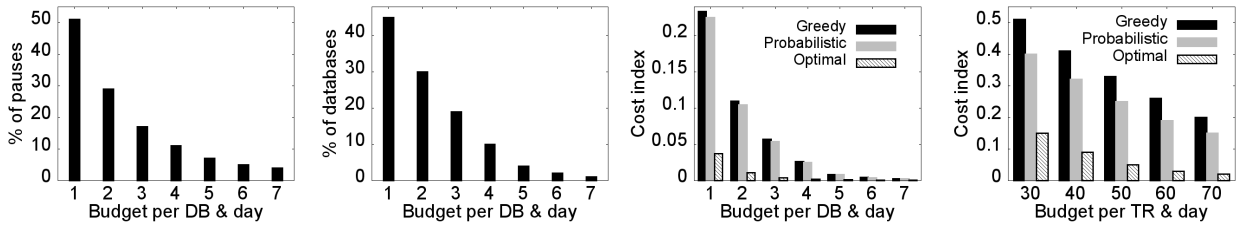
Example 5.3. The database in Figure 9 has 45 pause/resume workflows during 8 weeks. The total duration of all pauses is 50 days, while the median pause duration is 17 hours. If one pause per database and day is allowed, then one pause on 2/13 is avoided by the greedy budget. The avoided pause is highlighted by green arrow. Its duration is 33 hours. On the up side, the cost index of greedy budget is only 0.03 for this database since only one pause is avoided. On the down side, 34 pauses are shorter than the avoided pause. Short pauses should be avoided to reduce the budget cost index.

Probabilistic Budget. We propose to prioritize expected long pauses over expected short pauses, while spending the budget. To this end, we compute probabilities of long pauses per database and time window and detect pause patterns analogously to resume patterns per Definitions 4.2 and 4.3.

Optimal budget. To assess the quality of probabilistic budget, we compare it to optimal budget that knows the duration of pauses in advance and avoids the shortest pauses only.

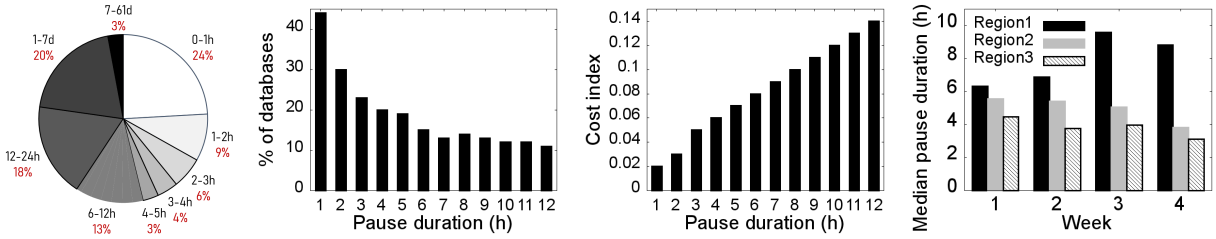
Fine-Grained Budget. If budget is defined per database and day, we compute the percentages of avoided pauses and databases with avoided pauses in Figures 10(a) and 10(b), respectively. For example, if only one pause is allowed per database and day, 51% of pauses are avoided and 45% of databases have avoided pauses. These percentages rapidly decline as budget increases.

In Figure 10(c), we compare three budgeting algorithms with respect to their costs. Unfortunately, probabilistic budget is only



(a) Avoided pauses (b) Databases with avoided pauses (c) Budget cost index (d) Budget cost index

Figure 10: Budget



(a) Pause duration (b) Databases with logical pauses (c) Logical cost index (d) Median pause duration

Figure 11: Logical pause

up to 1% cheaper than greedy and up to 19% more expensive than optimal. Predicting long pauses is challenging for two reasons. One, majority of pauses are short, within 7 hours (Figure 11(a)). Two, 3 weeks of history per database are required to predict pauses but only half of serverless databases existed that long (Figure 6).

Coarse-Grained Budget. To reduce the cost of probabilistic budget compared to greedy, we defined budget at coarser granularity. Figure 10(d) illustrates the cost comparison when budget is defined per tenant ring (TR) and day. Now, probabilistic budget has more flexibility to choose long pauses. Indeed, it can pause the same database more often if its pauses tend to be long and disallow all pauses of another database if its pauses are usually short. Thanks to this flexibility, probabilistic budget is now up to 12% cheaper than greedy but also up to 26% more expensive than optimal. Lastly, we defined even coarser-grained budget per cluster and/or week and observed similar results to Figure 10(d).

Summary. While half of pauses are avoided and almost half of databases profit from this workload reduction, greedy budget does not guarantee that only short pauses are avoided. Thus, its cost is not negligible. Probabilistic budget does not reduce this cost much. Therefore, we explore alternative solutions in Section 5.2.

5.2 Logical Pause

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

Definition 5.4. (**Logical Pause**). Let t_1 be the time point when a database becomes idle and t_2 be the time point when the resources are taken away from this database. The time interval (t_1, t_2) is called *logical pause*, while the time point t_2 is called *physical pause*.

During logical pause, resources are available and can be used immediately if the customer comes back online. In this way, we reduce delays in resource availability, while avoiding short pauses. Unfortunately, this optimization technique does not come for free since resources are idle during logical pauses. Not only short pauses are avoided but also long pauses are shortened by logical pauses.

Definition 5.5. (**Logical Cost Index**). Let l be the duration of logical pause in hours, $avoided(d)$ be the duration of avoided pauses in hours for a database d, and $allowed(d)$ be the number of pauses of d that are longer than l. Other notations are summarized in Table 2. Then, the wasted cost is computed as follows:

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

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

Example 5.6. Assume the duration of logical pause is set to 9 hours. On the up side, two shortest pauses are avoided for the database in Figure 9. Avoided pauses are highlighted by red arrows. On the down side, the remaining 43 pauses are 9 hours shorter. Therefore, the logical cost index is high (0.34) for this database.

Logical Pause Duration. The number of avoided pauses and the logical cost index depend on the duration of logical pause that we now tune. In Figure 11(a), we compute the percentage of pauses per pause duration in hours. As this figure illustrates, most pauses are short. Indeed, 24% of pauses do not exceed an hour, 51% are within 7 hours, 77% are within a day, and 97% are within a week. Based on these results, we conclude that logical pause of 7 hours avoids half of pauses. Vast majority of serverless databases (82%)

have pauses that are within 7 hours (Figure 11(b)). These databases benefit from this workload reduction.

As Figure 11(c) illustrates, the logical cost index grows from 0.02 to 0.14 as the duration of logical pause increases from 1 to 12 hours. In particular, the logical cost index is 0.09 for 7-hours-long logical pause which is a reasonable cost for reducing the system workload by half. Indeed, the cost index of probabilistic budget is 0.23 and 0.4 in Figures 10(c) and 10(d) for the same workload reduction.

Median Pause Duration. Given that the results may vary by region and week (Section 3), we measured median pause duration during 4 weeks in 3 regions and observed significant differences in Figure 11(d). Logical pause of 7 hours is a good choice only for Week 2 in Region 1. Regions 2 and 3 have much shorter median pause duration than Region 1 for all weeks. Region 1 has median pause duration between 9 and 10 hours during Weeks 3 and 4. Therefore, we conclude that logical pause duration must be tailored for each Azure region and adjusted over time.

Summary. Logical pause is a simple, yet effective way to avoid short pauses and reduce the system workload. Majority of serverless database profit from this optimization at relatively low cost.

6 PUTTING IT ALL TOGETHER

The problem space is two-dimensional where each dimension corresponds to the goal of Moneyball (Figure 12). 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. This analysis covers one randomly selected week and region where tens of thousands of serverless databases were sampled.

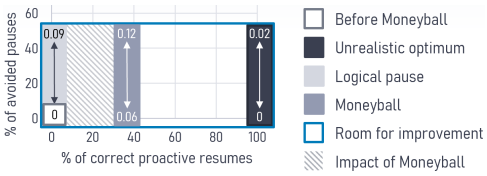


Figure 12: Moneyball problem space

Before Moneyball, all resumes were reactive, all pauses were allowed, and no COGS were wasted (i.e., cost index was 0). This case is shown as a white rectangle in Figure 12.

In ideal case, all resumes are proactive and correct. In addition, up to half of pauses are avoided and these avoided pauses are the shortest to minimize costs. Given that resources are idle during avoided pauses, the cost index of the optimal solution is 0.02 (Definition 5.2). The range of these unrealistic optimal solutions is shown as a black rectangle. The area between the solution before Moneyball 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. For example, if logical pause is 1 hour, 24% of pauses are avoided and the cost index is only 0.02 (Definition 5.5). Increasing logical pause to 7 hours reduces the number of pauses by 50% but also increases the cost index to 0.09. The spectrum of logical-pause-based solutions is shown as a light gray rectangle.

37% of all resumes are proactive and correct for long-lived databases. Due to wait time until the proactively resumed resources are used, the cost index is 0.06 (Definition 4.5). Combining proactive resume with logical pause of 7 hours makes one thirds of resumes proactive and correct for long-lived databases, while still avoiding half of pauses. This combined Moneyball approach is shown as a dark gray rectangle. Its cost index is 0.12 which we consider to be reasonable cost for these optimization techniques. The striped area between the solution before Moneyball and our proposed solution represents the impact of this work.

7 RELATED WORK

Self-driving databases [14, 32] in general and demand-driven autoscale of resources [20–24, 26, 31, 38–40] in particular have become popular research directions in the recent years. However, some of these state-of-the-art approaches are merely reactive [20–22]. 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 [39], reconfigure databases based on predicted load [40], or benchmark the efficiency of a cloud service [30]. These mechanisms are orthogonal to the Moneyball problem (Section 2).

Other approaches focus on load analysis [19, 26, 29, 37], load prediction using machine learning and other techniques [18, 23–25, 27, 30, 33, 34, 38, 41], or learning a relationship between available resources and performance [31]. 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 [21, 23, 27, 30, 33, 38, 39, 41], none of them achieves the contradictory goals of enabling proactive resume to guarantee high QoS and avoiding ineffective pause/resume workflows to alleviate this system workload, while controlling operational costs at the same time. This is the key contribution of our Moneyball approach.

8 CONCLUSIONS

Moneyball approach introduces the following optimization techniques of Microsoft Azure SQL Database serverless, while minimizing costs of operation. (1) To reduce delays in resource availability, we analyze resume patterns per database over time and proactively resume resources. One third of resumes are proactive and correct for long-lived databases. Two thirds of long-lived databases have proactive resumes. (2) To reduce the system workload, we avoid ineffective pauses for short idle periods. Instead, we logically pause a database when it becomes idle, i.e., we delay scaling resources down. Logical pause is a simple, flexible, and effective technique to reduce the number of short pauses. Results of this study are currently being used in production in all Azure regions.

REFERENCES

[1] Alibaba Function Compute. https://www.alibabacloud.com/product/function-compute.
[2] Amazon RDS for SQL Server. https://aws.amazon.com/rds/sqlserver.
[3] ARIMA. https://pypi.org/project/pmdarima.
[4] Azure SQL Database. https://azure.microsoft.com/en-us/products/azure-sql/database.
[5] Azure SQL Database pricing. https://azure.microsoft.com/en-us/pricing/details/azure-sql-database.
[6] Azure SQL Database serverless. https://docs.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview.
[7] Exponential Smoothing. https://machinelearningmastery.com/exponential-smoothing-for-time-series-forecasting-in-python/.
[8] GluonTS. https://gluon-ts.mxnet.io/.
[9] Google Cloud SQL. https://cloud.google.com/sql.
[10] Google Serverless Computing. https://cloud.google.com/serverless.
[11] LSTM. https://keras.io/api/layers/recurrent_layers/lstm/.
[12] Moneyball (film). https://en.wikipedia.org/wiki/Moneyball_(film).
[13] NimbusML. https://docs.microsoft.com/en-us/python/api/nimbusml/nimbusml.timeseries.ssaforecaster.
[14] Oracle Autonomous Database. https://www.oracle.com/autonomous-database/.
[15] Probability theory. https://en.wikipedia.org/wiki/Event_(probability_theory).
[16] Prophet. https://facebook.github.io/prophet.
[17] Serverless on AWS. https://aws.amazon.com/serverless/.
[18] R. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya. Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications' QoS. *IEEE Transactions on Cloud Computing*, 3:449–458, 08 2014.
[19] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*, page 153–167, 2017.
[20] S. Das, F. Li, V. R. Narasayya, and A. C. König. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In *SIGMOD*, pages 1923–1924, 2016.
[21] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. *SIGPLAN Not.*, 49(4):127–144, 2014.
[22] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-Regulating Stream Processing in Heron. In *Proc. VLDB Endow.*, pages 1825–1836, 2017.
[23] Z. Gong, X. Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *TNSM*, pages 9–16, 2010.
[24] S. Islam, J. Keung, K. Lee, and A. Liu. Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud. *Future Generation Comp. Syst.*, 28:155–162, 01 2012.
[25] A. Khan, X. Yan, S. Tao, and N. Anerousis. Workload Characterization and Prediction in the Cloud: A Multiple Time Series Approach. In *IEEE Network Operations and Management Symposium*, pages 1287–1294, 2012.
[26] C. Kilcioglu, J. M. Rao, A. Kannan, and R. P. McAfee. Usage Patterns and the Economics of the Public Cloud. In *WWW*, page 83–91, 2017.
[27] W. Lang, K. Ramachandra, D. J. DeWitt, S. Xu, Q. Guo, A. Kalhan, and P. Carlin. Not for the Timid: On the Impact of Aggressive over-Booking in the Cloud. *Proc. VLDB Endow.*, 9(13):1245–1256, 2016.
[28] M. Lewis. *Moneyball: The Art of Winning an Unfair Game.* W.W. Norton and Company, 2003.
[29] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters. *SIGMETRICS Perform. Eval. Rev.*, 37(4):34–41, Mar. 2010.
[30] J. Moeller, Z. Ye, K. Lin, and W. Lang. Toto - Benchmarking the Efficiency of a Cloud Service. In *SIGMOD*, 2021. To appear.
[31] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated Control of Multiple Virtualized Resources. In *EuroSys*, page 13–26, 2009.
[32] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-Driving Database Management Systems. In *CIDR*, 2017.
[33] J. Picado, W. Lang, and E. C. Thayer. Survivability of Cloud Databases - Factors and Prediction. In *SIGMOD*, page 811–823, 2018.
[34] O. Poppe, T. Amuneke, D. Banda, A. De, A. Green, M. Knoertzer, E. Nosakhare, K. Rajendran, D. Shankargouda, M. Wang, A. Au, C. Curino, Q. Guo, A. Jindal, A. Kalhan, M. Oslake, S. Parchani, V. Ramani, R. Sellappan, S. Sen, S. Shrotri, S. Srinivasan, P. Xia, S. Xu, A. Yang, and Y. Zhu. Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation. *Proc. VLDB Endow.*, 14(2):154–162, 2020.
[35] O. Poppe, A. Au, A. De, R. Sellappan, S. Sen, D. Shankargouda, M. Wang, T. Amuneke, D. Banda, A. Green, M. Knoertzer, E. Nosakhare, K. Rajendran, V. Ramani, S. Srinivasan, C. Curino, A. Jindal, Y. Zhu, Q. Guo, A. Kalhan, M. Oslake, S. Xu, S. Parchani, S. Shrotri, and P. Xia. Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation. https://arxiv.org/abs/2009.12922, 2020. Extended version.
[36] O. Poppe, Q. Guo, W. Lang, P. Arora, M. Oslake, S. Xu, and A. Kalhan. Moneyball: Optimizing Microsoft Azure SQL Database Serverless, While Minimizing Costs. https://www.dropbox.com/s/i7guk9ngz98y9c3/Moneyball.pdf?dl=0, 2021. Extended version.
[37] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *SOCC*, pages 1–13, 2012.
[38] N. Roy, A. Dubey, and A. Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *CLOUD*, pages 500–507, 2011.
[39] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *SOCC*, pages 1–14, 2011.
[40] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. Andrade. P-Store: An Elastic Database System with Predictive Provisioning. In *SIGMOD*, page 205–219, 2018.
[41] L. Viswanathan, B. Chandra, W. Lang, K. Ramachandra, J. M. Patel, A. Kalhan, D. J. DeWitt, and A. Halverson. Predictive Provisioning: Efficiently Anticipating Usage in Azure SQL Database. In *ICDE*, pages 1111–1116, 2017.