Proactive Resume and Pause of Resources for Microsoft Azure SQL Database Serverless

Olga Poppe
Microsoft Corporation
Redmond, WA, USA
olpoppe@microsoft.com

Pankaj Arora
Microsoft Corporation
Redmond, WA, USA
paarora@microsoft.com

Sakshi Sharma
Microsoft Corporation
Bengaluru, India
sakssharma@microsoft.com

Jie Chen
Microsoft Corporation
Sunnyvale, CA, USA
jiechen2@microsoft.com

Sachin Pandit
Microsoft Corporation
Sunnyvale, CA, USA
sapandi@microsoft.com

Rahul Sawhney
Microsoft Corporation
Bengaluru, India
rahulsawhney@microsoft.com

Vaishali Jhalani
Microsoft Corporation
Bengaluru, India
vjhalani@microsoft.com

Willis Lang
Microsoft Corporation
Edina, MN, USA
wilang@microsoft.com

Qun Guo
Microsoft Corporation
Redmond, WA, USA
qunguo@microsoft.com

Anupriya Inumella
Microsoft Corporation
Sunnyvale, CA, USA
aninumel@microsoft.com

Sanjana Dulipeta Sridhar
Microsoft Corporation
Bengaluru, India
sanjanadu@microsoft.com

Dheren Gala
Microsoft Corporation
Bengaluru, India
dherengala@microsoft.com

Nilesh Rathi
Microsoft Corporation
Bengaluru, India
nileshrathi@microsoft.com

Morgan Oslake
Microsoft Corporation
Redmond, WA, USA
moslake@microsoft.com

Alexandru Chirica
Microsoft Corporation
Atlanta, GA, USA
achirica@microsoft.com

Sarika Iyer
Microsoft Corporation
Sunnyvale, CA, USA
saiyer@microsoft.com

Prateek Goel
Microsoft Corporation
Bengaluru, India
prateekgoel@microsoft.com

Ajay Kalhan
Microsoft Corporation
Redmond, WA, USA
ajayk@microsoft.com

ABSTRACT

Demand-driven resource allocation for cloud databases has become a popular research direction. Recent approaches have evolved from reactive policies to proactive decision making. These approaches leverage not only the current resource demand but also the predicted demand to make more informed resource allocation decisions for each database and thus improve the quality of service and reduce the operational costs. We present an infrastructure that enables proactive resource allocation capabilities for millions of serverless Azure SQL databases. Our solution finds near-optimal middle ground between high availability of resources, low operational costs, and low computational overhead of the proactive policy. We describe the design principles we followed and the architectural decisions we made during this cross-team, multi-year journey. Given the size and scope of our solution, we believe that the relational cloud databases in other companies could benefit from the proactive resource allocation capabilities.

CCS CONCEPTS

• **Computer systems organization → Self-organizing autonomic computing**.

KEYWORDS

autonomous database, proactive auto-scale of resources

ACM Reference Format:
Olga Poppe, Pankaj Arora, Sakshi Sharma, Jie Chen, Sachin Pandit, Rahul Sawhney, Vaishali Jhalani, Willis Lang, Qun Guo, Anupriya Inumella, Sanjana Dulipeta Sridhar, Dheren Gala, Nilesh Rathi, Morgan Oslake, Alexandru Chirica, Sarika Iyer, Prateek Goel, and Ajay Kalhan. 2024. Proactive Resume and Pause of Resources for Microsoft Azure SQL Database Serverless. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion'24), June 9–15, 2024, Santiago, Chile.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3626246.3653371

1 INTRODUCTION

Microsoft Azure SQL Database operates and manages millions of databases in tens of Azure regions [5, 50]. Resource utilization of these databases has been rigorously studied for over a decade to optimize resource allocation including CPU, memory, and disk. This analysis reveals that resources are largely under-utilized, while there are workload spikes that are throttled by fixed resource capacity limits [29, 45, 49, 56, 57, 59, 66]. These observations gave rise to serverless computing that continuously adjusts the amount of resources to meet the current resource demand of each database [7]. Resources of idle databases are reclaimed and reused to handle the current workload of active databases. In this way, the number of physical machines is reduced and thus operational costs are saved compared to fixed size resource provisioning per database. Customers are billed per second for the amount of compute resources they used. Billing stops once the workload completes.

Limitations of The Reactive Policy. Current demand-driven resource allocation is quite shortsighted due its reactive nature. Indeed, the reactive policy is based on the current demand and does not leverage the insights from unique historical workload traces for each database. Thus, this policy suffers from two severe limitations.

(1) *Delayed Resource Availability.* Resource allocation mechanisms are not instantaneous. Thus, if resources are reclaimed during prolonged idle intervals, then delays in resource availability are possible when the customer resumes activity. In the worst case, there is not enough resource capacity on the node to resume the resources for a database. Such database must be moved to another node with higher available amount of resources [42]. These delays make the reactive policy less suitable for latency sensitive cloud service than the fixed size provisioned policy. In this work, we aim to overcome the reactive nature of serverless compute and enable proactive resource allocation decisions for each database [59].

(2) *Computational Overhead.* Vast majority of idle intervals are within one hour (Figure 3(a)). Short idle intervals make resource availability time too fragmented for effective reuse of resources by active databases. Worst yet, if the resources are reclaimed immediately once the workload stops, then concurrent execution of resource allocation and reclamation workflows at high density can introduce significant computational overhead that consumes resources instead of saving them. In this work, we aim to reduce the provider infrastructure load by minimizing the number of concurrent workflows that can cause performance and reliability issues.

Challenges. Proactive resource allocation for database systems is a non-trivial endeavour for the following reasons.

(1) *Varying Resource Usage Patterns per Database.* Rigorous resource utilization analysis reveals that the resource usage patterns vary per database [29, 45, 49, 56, 57, 59, 66]. There are databases with stable usage, databases that follow a weekly or a daily pattern, and databases that have short unpredictable spikes of activity. Furthermore, resource utilization may change over time for each database. Therefore, proactive resource allocation decisions must be based on the recent resource usage history for each database.

(2) *High Quality of Service Requirement.* Azure SQL Database guarantees 99.99% or higher availability [2, 17]. Resources must be available upon customer login even if resources are reclaimed during prolonged idle intervals to save opertional costs. Thus, we design an infrastructure that enables proactive resource allocation decisions per database, has no dependency on external components, and no single point of failure [41]. Furthermore, the database history must be durable. In particular, if a database moves from one compute node to another to balance the load, its history must move with it to enable proactive resource allocation after the move.

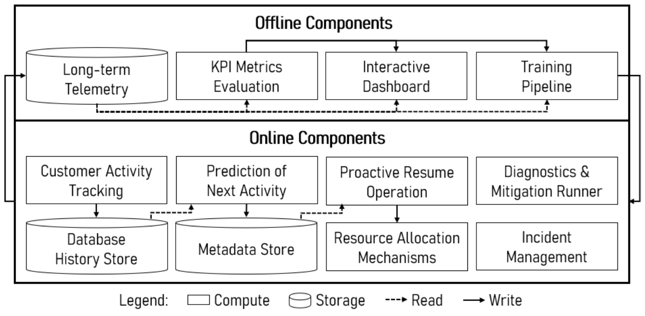

Offline Components
Long-term Telemetry
KPI Metrics Evaluation
Interactive Dashboard
Training Pipeline
Online Components
Customer Activity Tracking
Prediction of Next Activity
Proactive Resume Operation
Diagnostics & Mitigation Runner
Database History Store
Metadata Store
Resource Allocation Mechanisms
Incident Management
Legend: Compute Storage Read Write

Figure 1: ProRP Infrastructure

(3) *Continuous Resource Allocation for Millions of Databases.* The mechanisms implementing the proactive resource allocation decisions must be scaled to millions of serverless databases in tens of regions. They must handle tens of thousands of resource allocation and reclamation workflows per region per day (Figures 11 and 12). Such automated continuous resource allocation alleviates the burden of manual tuning which is labor-intensive, error-prone, costly, neither scalable to millions of databases, nor durable due to changing resource demand per database.

ProRP Infrastructure. To tackle these challenges, we designed and implemented an infrastructure for Proactive Resume and Pause of resources for Azure SQL Database Serverless, ProRP for short (Figure 1). For each database, the online components of the infrastructure track customer activity, compactly store the recent history, detect the customer login patterns to predict the next activity, and make proactive resource allocation decisions based on database lifespan, current and predicted customer activity. In particular, idle resources are paused if no customer activity is predicted in near future to save operational costs. Resources are resumed ahead of predicted customer login to guarantee high quality of service.

Customer activity and resource allocation decisions are persisted long-term for offline evaluation of KPI metrics. These metrics include quality of service, operational cost efficiency, and computational overhead. Given that customer activity changes over time, the training pipeline tunes the configuration knobs of the activity tracking and prediction based on long-term telemetry.

State-of-the-Art Techniques. Autonomous database management systems have become popular in academia [53] and industry [14] recently. Several existing approaches propose generally applicable infrastructures for machine learning [10, 18, 25, 26, 38, 48]. Unfortunately, they rely on products and services that are external to Azure SQL Database. Consequently, commercial license, compliance with Microsoft security and privacy requirements, compatibility with Azure SQL components, scalability to all Azure regions, and high availability guarantee have to be addressed prior to productization. Moreover, numerous previous studies to predict the load of Azure SQL databases reveal that the accuracy of simple statistical and probabilistic load prediction techniques is sufficient in practice [21, 27, 42, 45, 49, 56, 57, 59, 66]. We experimentally confirmed that this conclusion holds in our case (Section 9). Therefore, the cost and the engineering effort to maintain an external machine learning infrastructure long term in production worldwide is not justified by the slightly higher accuracy of more advanced machine learning models. Due to these practical considerations, we have decided to build the online components of the ProRP infrastructure within the code base of Azure SQL Database. Nevertheless, we learn from the state-of-the-art generic infrastructures and adapt their design principles when possible (Section 3). Furthermore, we leverage Azure ML [3, 57] to automate and distribute the offline training of next activity prediction (Section 8).

The system tuning solutions in academia propose sophisticated machine learning models without solving the practical challenges of an industrial product described above [32, 53, 54, 65, 68]. Other related approaches either auto-tune a specific system component (e.g., memory [28, 62], data structures [33, 37], indexes [22, 23, 43], query optimizer [35, 40, 46, 47], compiler [19, 24, 34]) or consider an orthogonal use case (e.g., overbooking [45, 66], tenant placement [42], backup scheduling [57], benchmarking [49]). Doppler [21] recommends only the initial amount of resources for provisioned databases and does not tackle the challenges of continuous proactive resource allocation for serverless databases. We leverage the insights from our prior research on the proactive resource allocation policy [59] and focus on design and implementation of the ProRP infrastructure in this publication. To the best of our knowledge, ProRP is the first infrastructure that is deployed for millions of serverless databases worldwide and enriches them with continuous proactive resource allocation capabilities.

Contributions. Our key contributions are the following.

(1) Guided by the system requirements of Azure SQL Database, we describe the design principles we followed, while enriching it with continuous proactive resource allocation capabilities.

(2) We implement the ProRP infrastructure in Figure 1 to automate the whole life cycle of the proactive resource allocation from telemetry emission, storage, and analytics to resource allocation mechanisms, KPI metrics evaluation, and parameter tuning.

(3) We propose the algorithms for the proactive resource allocation policy, the database history maintenance, the prediction of next activity, and the proactive resume of resources per database. We study the time and space complexity of these algorithms.

(4) We define the KPI metrics and experimentally evaluate the effectiveness of the ProRP infrastructure in production. It achieves near optimal balance between quality of service and operational cost efficiency. At the same time, ProRP is light-weight since the latency of proactive decisions stays within one second, while the size of history store is within a few kilobytes per database.

(5) We deploy and battle-test the ProRP infrastructure in production in all Azure regions and share our experience during this cross-team, multi-year journey. We also discuss the current limitations and sketch several future work directions.

Outline. We start with preliminaries in Section 2. We describe our design principles in Section 3. We propose the algorithms for all key components of the ProRP infrastructure in Sections 4–8. We experimentally evaluate our solution in Section 9. We review the related work in Section 10 and conclude the paper in Section 11.

Table 1: Notations

Parameter	Meaning
$\mathbb{D}$	Set of serverless databases, $d \in \mathbb{D}$
$\mathbb{T}$	Set of time points, $t \in \mathbb{T}$
$D(d,t)$	Resource demand of d at t
$A(d,t)$	Resource allocation for d at t
n	Number of tuples in database history, $n \in \mathbb{N}$
m	Number of tuples returned by a range query, $m \in \mathbb{N}, m \leq n$
l	Duration of logical pause (default: 7 hours)
h	History length (default: 28 days)
p	Prediction horizon (default: 1 day)
c	Confidence threshold (default: 0.1)
w	Window size (default: 7 hours)
s	Window slide (default: 5 minutes)
k	Pre-warm time interval (default: 5 minutes)

2 PRELIMINARIES

2.1 Basic Notions and Assumptions

Time is represented by a linearly ordered set of time points $(\mathbb{T}, \leq)$, where $\mathbb{T} \subseteq \mathbb{R}^+$ is the set of non-negative real numbers. Let $\mathbb{D}$ be the set of databases. Other notations are summarized in Table 1.

In this paper, we focus on the binary problem, i.e., the resources are either allocated or reclaimed for each database at each point of time. Our solution is a stepping stone towards proactive auto-scale of resources in small increments of capacity (Section 11).

Definition 2.1. (**Resource Demand and Allocation**) Resource demand $D : \mathbb{D} \times \mathbb{T} \rightarrow \{0, 1\}$ is a function that maps a database $d \in \mathbb{D}$ and a time point $t \in \mathbb{T}$ to a binary value indicating whether the resources of d are needed by the customer at t. $\forall d \in \mathbb{D}\ \forall t \in \mathbb{T}$ if the resources of d are needed at t then $D(d,t) = 1$, else $D(d,t) = 0$.

Analogously, resource allocation $A : \mathbb{D} \times \mathbb{T} \rightarrow \{0, 1\}$ is a function that maps $d \in \mathbb{D}$ and $t \in \mathbb{T}$ to a binary value indicating whether the resources are allocated or reclaimed for d at t.

Definition 2.2. (**Correctness of Resource Allocation**) For a database $d \in \mathbb{D}$ and a time point $t \in \mathbb{T}$, the resources are:
- Correctly allocated (used) if $D(d,t) = A(d,t) = 1$,
- Correctly reclaimed (saved) if $D(d,t) = A(d,t) = 0$,
- Wrongly allocated (idle) if $D(d,t) = 0$ and $A(d,t) = 1$,
- Wrongly reclaimed (unavailable) if $D(d,t) = 1$ and $A(d,t) = 0$.

2.2 Limitations of the Current Reactive Policy

Resources of Azure SQL Database Serverless are automatically scaled based on demand [7]. While the workload is running, resources are *resumed* for the database. While the database is idle, resources are *paused* for the database and possibly assigned to other active databases. Customers are billed per second for compute resources only while they use these resources. In this way, serverless compute improves both the resource utilization and the costs for the customers [6]. Unfortunately, the current resource allocation policy is merely reactive to the current resource demand.

Figure 2 illustrates the resource demand and allocation under the reactive, proactive, and optimal policies. Resource demand is

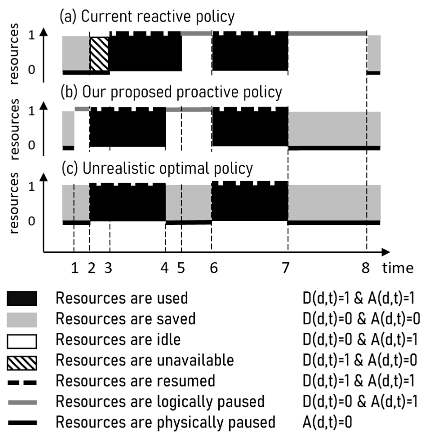
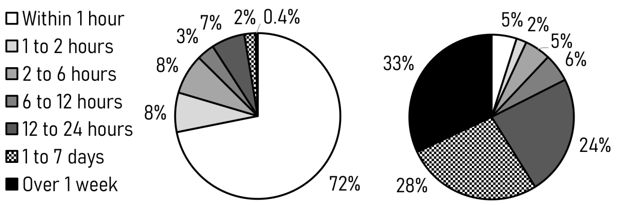

(a) Current reactive policy
(b) Our proposed proactive policy
(c) Unrealistic optimal policy
resources
1
0
1 2 3 4 5 6 7 8 time
Resources are used D(d,t)=1 & A(d,t)=1
Resources are saved D(d,t)=0 & A(d,t)=0
Resources are idle D(d,t)=0 & A(d,t)=1
Resources are unavailable D(d,t)=1 & A(d,t)=0
Resources are resumed D(d,t)=1 & A(d,t)=1
Resources are logically paused D(d,t)=0 & A(d,t)=1
Resources are physically paused A(d,t)=0

Figure 2: Resource allocation policies [58]



(a) Number of idle intervals (b) Duration of idle intervals

Figure 3: Fragmentation of idle time

shown as black area. Resource allocation is shown as horizontal line. In ideal case, resource allocation is the minimal bounding box of resource demand (Figure 2(c)). The current reactive policy often fails to meet the resource demand for the following reasons.

Idle Resources due to Logical Pauses. We analyzed two month of production telemetry from a large Azure region where hundreds of thousands of serverless databases are currently deployed. We concluded that 72% of idle intervals are within one hour (Figure 3(a)). However, these short idle intervals contribute only 5% to the total idle time duration (Figure 3(b)). Short idle intervals make resource availability time too fragmented for effective reuse and thus do not save operational costs. On the contrary, frequent resource allocation and reclamation workflows can introduce significant computational overhead and thus waste operational costs.

To avoid reclaiming resources for short idle intervals, we *logically pause* the resources of a database that became idle and wait if the customer resumes activity (Time 5 and 7 in Figure 2(a)). During logical pause, the resources are still available but customers are not billed. If there is no activity during logical pause, resources are *physically paused*, i.e., reclaimed (Time 8 in Figure 2(a)).

On the upside, resources are available when the workload returns at Time 6 in Figure 2(a) and the overhead of resource reclamation followed by reactive allocation is avoided at Time 5 and 6 in Figure 2(a). On the downside, resources stay idle from Time 5 to 6 and from Time 7 to 8 in Figure 2(a) and operational costs are wasted. We reduce resource idleness by physically pausing idle databases if no customer activity is predicted in near future (Time 7 in Figure 2(b)).

Unavailable Resources due to Physical Pauses. Given that resources are physically paused during prolonged idle intervals (Time 8 in Figure 2(a)), delays are possible on resume due to the reaction time between demand signal and effective change in resource allocation (from Time 2 to 3 in Figure 2(a)). We reduce these delays by proactively resuming resources at Time 1 ahead of predicted customer activity at Time 2 in Figure 2(b) [59].

2.3 Opposing Optimization Objectives

The ultimate goal of a resource allocation policy is to find the middle ground between quality of service (QoS) and operational cost efficiency [58, 59, 66]. QoS is the highest if resources are always available when the customer needs them. QoS is lower if resources are throttled or even unavailable. Operational cost efficiency is the highest if resources are only allocated when they are needed. Efficiency is lower if resources are underutilized.

The optimal balance between these opposing objectives is achieved when resources are allocated if and only if they are needed, i.e., $\forall d \in \mathbb{D}\ \forall t \in \mathbb{T}\ D(d,t) = A(d,t)$ (Figure 2(c)). Such optimal resource allocation requires a perfect resource demand prediction which is hard to achieve in practice due to continuously changing customer activity. Nevertheless, the effectiveness of any resource allocation policy is measured as the difference from this optimum [58, 59, 66]. Hence, we aim to minimize the time intervals when resources are unavailable (shown as striped area in Figure 2(a)), while maximizing the time intervals when resources are saved (shown as gray area). Lastly, we aim to minimize the overhead of the proactive policy to ensure scalability of our solution.

3 PRORP DESIGN PRINCIPLES

We align our design principles with the system requirements of Azure SQL Database, while enriching it with proactive resource allocation capabilities. In particular, our solution must be *self-sufficient* to guarantee up to 99.99% or higher availability [2, 17, 41], *scalable* to millions of serverless databases worldwide, *supportable* long term in production, and *effective* in finding the middle ground between quality of service and operational cost efficiency. Our design principles cover the infrastructure, analytics, and storage.

3.1 Infrastructure-Related Design Principles

No Single Point of Failure. To ensure reliability, scalability, and low latency of ProRP, we follow the best principles of distributed architecture and tightly couple both the storage of historical data and its analysis with each database.

No Human in the Loop. There are multiple configuration knobs of ProRP that influence its effectiveness. They include history retention interval, prediction horizon, seasonality, confidence threshold, etc. These knobs must be exposed and automatically tuned to adapt to continuously changing workload per database.

Seamless Integration into Azure SQL Ecosystem. To reduce the engineering effort to build the ProRP infrastructure and facilitate its long-term maintenance in production, our solution must be seamlessly integrated into the Azure SQL ecosystem and reuse its established components. In particular, we leverage the big data platform Cosmos [61], Azure ML pipelines [3], PowerBI monitoring tools [15], configuration, testing, and deployment infrastructure, diagnostics and mitigation runners, incident management, backup and restore mechanisms of Azure SQL Database.

3.2 Analytics-Related Design Principles

Proactive Database-Scoped Decisions. Given that resource utilization history varies per database [29, 45, 49, 56, 57, 59, 66], there is no single policy that fits all databases. Thus, we tailor the resource allocation decisions for each database to optimize the trade-off between quality of service and operational cost efficiency. In addition, these decisions are proactive, i.e., based on both current and predicted resource demand per database.

Simple Forecast Techniques. We have evaluated time series forecast models, decision tree-based models, Neural Networks, and Bayesian Optimization to predict the load and/or tune various configuration knobs [27, 39, 56, 57, 59, 66]. Some of these models (e.g., ARIMA [1] and Prophet [16]) do not scale to millions of databases. While there are machine learning tools that are highly optimized (e.g., NimbusML [13], GluonTS [9], and ML.NET binary trainer [11]), they rely on libraries that are external to Azure SQL Database which contradicts to the self-sufficiency requirement. Furthermore, statistical and probabilistic forecast techniques are easy to understand, explain, implement, debug, and maintain long-term in production worldwide. Their accuracy is sufficient in practice [21, 27, 42, 45, 49, 56, 57, 59, 66]. Due to these practical considerations, we deploy the probabilistic forecast techniques to predict the next activity per database (Section 6).

Database-State-Aware Processing. A serverless database is either resumed, or logically paused, or physically paused (Figure 4). Different proactive capabilities are relevant in different states of the database. At each point of time, we focus all efforts on handling the current situation by activating only those capabilities which are relevant in the current database state (Section 4). All other capabilities are suspended to save resources [60].

Default to Reactive Database-Scoped Decisions. If any component of ProRP goes down, the system must default to the reactive policy until the failed component comes up. Even though the reactive policy is less effective than the proactive policy [59], reactive database-scoped decisions are the only way to guarantee high quality of service, while proactive capabilities are unavailable.

3.3 Storage-Related Design Principles

Precise Timestamps of Customer Activity. The choice of the right resource utilization signal is crucial for the effectiveness of the proactive policy. We store and analyze the start and end of customer activity, rather than the resume and pause timestamps because certain system maintenance operations also trigger resume of resources. System maintenance operations are ignored by the proactive policy to save operational costs since the customer performance and availability experience is unaffected.

Given that customers are billed per second, even one second of delayed resource availability after an idle interval is undesirable. Therefore, the timestamps of customer activity must be precise to ensure accurate resource demand prediction.

Compact Database History Store. Given that resource usage patterns vary per database [29, 45, 49, 56, 57, 59, 66], there must be no limit on the size of database history. To enable real-time complex analytics, the history must be compact, i.e., contain only recent customer activity. The history store must expose the familiar SQL interface to efficiently update, retrieve, and aggregate the data.

Durable Database History Store. Given that databases may move from one node to another to balance the load on the cluster, database history must be available after the move to enable proactive resource allocation decisions without interruption.

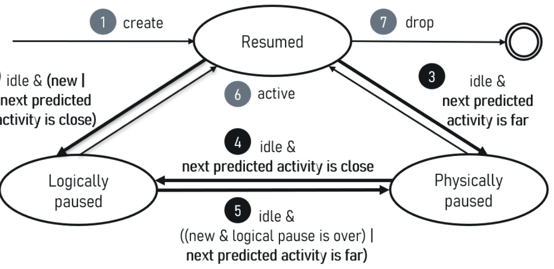

1 create
Resumed
7 drop
2 idle & (new | next predicted activity is close)
6 active
3 idle & next predicted activity is far
4 idle & next predicted activity is close
Logically paused
Physically paused
5 idle & ((new & logical pause is over) | next predicted activity is far)

Figure 4: Proactive resume and pause lifecycle of a database

4 PROACTIVE POLICY

Figure 4 represents the proactive resume and pause lifecycle of a serverless database as a Finite State Automaton. The bold Transitions ❷–❺ and conditions enable proactive resource allocation capabilities in addition to the reactive policy (Transition ❻). The proactive policy is defined in Algorithm 1. These functions implement the functionality within the respective states of a serverless database in Figure 4. Table 1 summarizes the configuration parameters of Algorithm 1 and their default values.

Resumed Resources. We track start and end of customer activity in Lines 3 and 6. As long as the database is active, its resources are resumed to serve the current workload in Lines 4–5. Once the database becomes idle, we delete its old history to keep the recent history compact in Line 8 and predict the start and end of next activity in Line 9. We skip deletion of old history and prediction of next activity if the previous predicted activity is not over yet in Line 7. While deleting old history, we determine if the database is old, i.e., existed at least the history retention time interval h time units and has enough history to make a reliable activity prediction. If a database is new, then the resource allocation policy defaults to reactive (Section 2.2).

If no customer activity is expected within the duration of logical pause of l time units, then the resources are physically paused to save operational costs in Lines 10–11 (Transition ❸). Otherwise, the resources are logically paused to relieve the backend from frequent resource allocation operations in Line 12 (Transition ❷). Additionally, by delaying or gradually tapering resource reclamation while logically paused, the impact to database performance is reduced. In particular, resources are logically paused for a new database that did not accumulate enough resource utilization history yet and therefore the next activity cannot be predicted.

Logically Paused Resources. Resources stay logically paused until either the database becomes active (Transition ❻) or l time units of logical pause are over for a new database or next predicted activity is not over yet or is expected to start within the next l

Algorithm 1 Proactive resource allocation policy

```
1:  function RESUME()
2:      AllocateResources()
3:      InsertHistory(now,1)
4:      while active do
5:          ProcessWorkload()
6:      InsertHistory(now,0)
7:      if nextActivity.end < now then
8:          old ← DeleteOldHistory(h,now)
9:          nextActivity ← PredictNextActivity(h,c,w,s,now)
10:     if idle & (now+l ≤ nextActivity.start ||
           (old & nextActivity.start = 0)) then
11:         PhysicalPause()
12:     else LogicalPause()
13: function LOGICALPAUSE()
14:     AllocateResources()
15:     pauseStart ← now
16:     pauseEnd ← 0
17:     resume ← false
18:     while pauseEnd = 0 do
19:         while idle & ((!old & now < pauseStart+l) ||
                  now < nextActivity.end ||
                  now < nextActivity.start < now+l) do
20:             Sleep()
21:         if active then
22:             pauseEnd ← now
23:             resume ← true
24:         else old ← DeleteOldHistory(h,now)
25:             nextActivity ←
                    PredictNextActivity(h,p,c,w,s,now)
26:             if idle & ((!old & pauseStart+l < now) ||
                    now+l ≤ nextActivity.start ||
                    (old & nextActivity.start = 0)) then
27:                 pauseEnd ← now
28:     if resume then Resume()
29:     else PhysicalPause()
30: function PHYSICALPAUSE()
31:     InsertMetadata(nextActivity.start)
32:     ReclaimResources()
```

time units for an old database in Lines 19–20 (Transition ❺). If the database is still idle after the logical pause is over, then the next activity is predicted to physically pause an old database if no activity is expected in the next l time units in Lines 24–29. A new database is also physically paused after l time units of idleness even though the next activity is unknown for it in Lines 26–29.

Physically Paused Resources. The start of next predicted activity is stored in the metadata store and the resources are reclaimed in Lines 31–32 (Transition ❹). Lastly, the periodic management operation accesses the next predicted activity of physically paused databases in the metadata store and resumes resources k time units ahead of predicted customer activity per database (Section 7).

Algorithm 2 Insertion of database history

```
1: CREATE PROCEDURE sys.InsertHistory (
2:     @time BIGINT, @type INT) AS
3: IF NOT EXISTS
4:     (SELECT *
5:     FROM sys.pause_resume_history
6:     WHERE time_snapshot = @time)
7:         INSERT INTO sys.pause_resume_history
8:                     (time_snapshot, event_type)
9:         VALUES (@time, @type)
```

5 CUSTOMER ACTIVITY TRACKING

Database History Store. To ensure durability of database history (Section 3.1), we persist it in the dedicated internal table `sys.pause_resume_history` of the database itself. In this way, we avoid creating an additional storage component that has to be moved across nodes when the database moves to balance the load.

Furthermore, a SQL database can store its variable-length history, update and analyze it via SQL interface. In fact, the algorithms for the database history maintenance and the prediction of next activity are implemented as SQL stored procedures in Algorithms 2–4. Lastly, we leverage the established backup and restore mechanisms of Azure SQL Database to tackle data loss.

The schema of this table consists of the following two columns:
(1) `time_snapshot` is an integer that represents the epoch time[1] of the start or end of customer activity. To facilitate history maintenance and analytics in Algorithms 2–4, we have chosen the machine-readable integer format to represent timestamps. In addition, we require that the values in this column are unique, while inserting a new tuple in Lines 3–6 in Algorithm 2. Lastly, we create a clustered B-tree-based index on the values in this column [8].

(2) `event_type` is a binary integer where 1 indicates a start of customer activity, while 0 indicates an end of activity.

We will publish a materialized view over this history to the customers. To this end, we convert both columns to human-readable format, i.e., epoch time is converted to date time, while event type is converted to string. The customers will have read access to this table but no write access to prevent modification of the history.

Precision of Login Timestamps. The activity prediction algorithm analyzes the login timestamps to detect recurring activity patterns (Section 6). Therefore, the accuracy of prediction depends on the precision of login timestamps. To ensure that these timestamps are precise, they are set on the critical login path, while the tuple insertion runs off the critical path on a timer.

Deletion of Old History. While database history does not exceed five hundred tuples per week on average, it can grow up to several thousands of tuples per week in the worst case (Figure 10(a)). To keep the database history compact, only h days of recent customer activity are kept by Algorithm 3, where h is a configurable parameter. We compute the start of recent history in Line 3.

To determine whether the database has enough history to make a reliable activity prediction in Lines 10, 19, and 26 in Algorithm 1, Algorithm 3 returns a boolean value indicating whether the database is old, i.e., existed before the start of recent history in Lines 6,

[1]Epoch time corresponds to the number of seconds passed since January 1, 1970.

Algorithm 3 Deletion of old history

```
1:  CREATE PROCEDURE sys.DeleteOldHistoy (
2:      @h INT, @now BIGINT, @old INT OUTPUT) AS
3:  DECLARE @historyStart BIGINT = @now - @h*24*60*60
4:  SELECT @minTimestamp = MIN(time_snapshot)
5:  FROM sys.pause_resume_history
6:  IF @minTimestamp < @historyStart
7:      SET @old = 1
8:      DELETE FROM sys.pause_resume_history
9:      WHERE @minTimestamp < time_snapshot AND
10:             time_snapshot < @historyStart
11: ELSE SET @old = 0
```

7, and 11 in Algorithm 3. To determine the lifespan of the database, we keep the oldest tuple in the history. The oldest tuple has the minimal timestamp in the history which is determined in Lines 4–5. All tuples between the minimal timestamp and the start of recent history are permanently deleted in Lines 8–10.

Complexity Analysis. Let n be the number of tuples in the table and m be the number of tuples in a range of timestamps, $m \leq n$. Given the B-tree index on the `time_snapshot` column, the search of one tuple in Lines 4–6 in Algorithm 2 and in Lines 4–5 in Algorithm 3 and the insertion of one tuple in Lines 7–9 in Algorithm 2 have logarithmic time complexity $O(\log n)$. The deletion of m tuples in Lines 8–10 in Algorithm 3 corresponds to a range query that also has logarithmic time complexity $O(\log m)$. The space complexity of the database history store is linear $O(n)$.

6 PREDICTION OF NEXT ACTIVITY

Due to the practical considerations described in Sections 1 and 3, we follow the probabilistic approach to predict the next activity per database. Below, we first explain the algorithm by example and then define it as a SQL stored procedure.

Example. To predict customer activity within the next 24 hours, we detect a daily pattern in history of length h time units. Prediction horizon is set to 24 hours because the algorithm detects the daily pattern, i.e., after 24 hours the pattern will repeat. Assume we predict activity on Day 6 based on 5 previous days in Figure 5.

To detect the daily activity pattern, we slide a window of length w time units every s time units, retrieve the timestamps of customer activity during this window and compute the probability of activity as the number of windows with activity divided by the duration of history of h days. If the probability of activity per window exceeds the confidence threshold c, then we consider the earliest and the latest hour and minute of activity during this window as start and end of predicted activity on the following day.

Let the confidence threshold be 0.8 in Figure 5. Then, there are several windows that satisfy this threshold, e.g., Window 1 with confidence $4/5 = 0.8$ and Window 2 with confidence $5/5 = 1$. In such cases, we select the predicted activity with the earliest start and the highest confidence, i.e., the predicted activity during Window 2.

If the window w is wide, then there can be several first logins after idle intervals during the window w on the same day, e.g., during Window 2 on Day 3 in Figure 5. Therefore, we count the number of windows with activity on h previous days, rather than

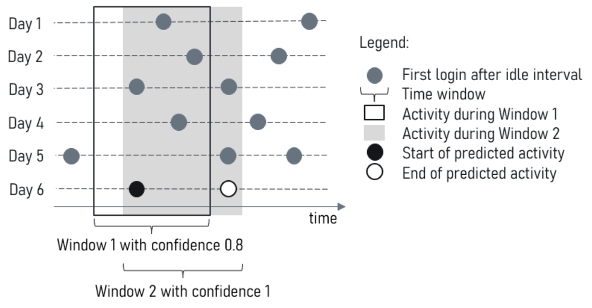

Day 1
Day 2
Day 3
Day 4
Day 5
Day 6
time
Window 1 with confidence 0.8
Window 2 with confidence 1
Legend:
First login after idle interval
Time window
Activity during Window 1
Activity during Window 2
Start of predicted activity
End of predicted activity

Figure 5: Prediction of next activity

the number of first logins during windows on h previous days. In this way, we ensure that the customer activity pattern consistently repeats during the window w on several previous days.

We activate the resources k time units before the predicted customer activity rather than at the beginning of the window for the sake of efficiency because resources will stay idle until the customer uses them (Section 7). Given that customer activity may continue beyond the end of predicted activity within the window (e.g., in Figure 5), we verify that no activity is predicted in the next l time units before reclaiming resources in Lines 7–11 and 25–29 in Algorithm 1.

Probabilistic Algorithm 4 consumes the history length h, the prediction horizon p, the confidence threshold c, the window size w, and the window slide s as parameters and returns the start and end of the next predicted activity on the next day. All local variables are set to 0 at the beginning. These variable declarations are skipped to keep Algorithm 4 compact.

The algorithm consists of two nested while-loops. The outer while-loop in Lines 9–47 slides the time window [@winStart, @winEnd] of length w minutes every s minutes and computes the probability of customer activity during this window as the number of past windows in history that contain activity divided by the size of history of h days in Line 36. If the probability of activity exceeds the confidence threshold c, then the earliest predicted activity with highest confidence is returned in Lines 37–46.

The inner while-loop in Lines 15–35 accesses the database history during the time windows [@winStartPrevDay, @winEndPrevDay] on h previous days, computes the timestamps of the first and last logins during these windows in Lines 19–33, and counts the number of windows with activity in Line 34.

Complexity Analysis. The number of iterations of the outer while-loop corresponds to the number of windows which is computed as the prediction horizon p in minutes divided by the window slide of s minutes. The number of iterations of the inner while-loop equals the history length of h days. Given the B-tree index on the `time_snapshot` column, the range query in Lines 19–24 has logarithmic time complexity $O(\log m)$ where m is the number of tuples in the range of timestamps. All other operations have constant time complexity $O(1)$. Therefore, the time complexity of Algorithm 4 is $p/s \times h \times O(\log m)$. Given that p, s, and h are constants and m is bound by the size of database history n, the latency of next activity prediction is within one second (Section 9). The space complexity is linear $O(n)$ in the size of database history n which does not exceed a few kilobytes on average (Section 9).

Algorithm 4 Prediction of next activity

```
CREATE PROCEDURE sys.PredictNextActivity (
    @h INT, @p INT, @c FLOAT, @w INT, @s INT,
    @now BIGINT,
    @startOfPredActivity BIGINT OUTPUT,
    @endOfPredActivity BIGINT OUTPUT) AS
DECLARE @historyStart BIGINT = @now - @h*24*60*60
DECLARE @winStart BIGINT = @now
DECLARE @predEnd BIGINT = @now + @p*60*60
WHILE @winStart + @w ≤ @predEnd
    SET @winWithActivity = 0
    SET @earliestLoginPerWin = @w
    SET @lastLoginPerWin = 0
    SET @winEnd = @winStart + @w
    SET @prevDay = 1
    WHILE @prevDay ≤ @h
        SET @winStartPrevDay = @winStart -
            @prevDay*24*60*60
        SET @winEndPrevDay = @winStartPrevDay + @w
        SELECT @firstLogin= MIN(time_snapshot),
                 @lastLogin = MAX(time_snapshot)
        FROM sys.pause_resume_history
        WHERE event_type = 1 AND
                 @winStartPrevDay ≤ time_snapshot AND
                 time_snapshot ≤ @winEndPrevDay
        IF @firstLogin is NOT NULL
            IF @firstLoginPerWin > @firstLogin -
                @winStartPrevDay
                SET @firstLoginPerWin = @firstLogin -
                    @winStartPrevDay
            IF @lastLoginPerWin < @lastLogin -
                @winStartPrevDay
                SET @lastLoginPerWin = @lastLogin -
                    @winStartPrevDay
            SET @winWithActivity = @winWithActivity + 1
        SET @prevDay = @prevDay + 1
    SET @prob = @winWithActivity / @h
    IF @c ≤ @prob AND @prevProb < @prob
        IF @prevStart = 0 OR
          @prevStart = @startOfPredActivity
            SET @startOfPredActivity = @winStart +
                @firstLoginPerWindow
            SET @prevStart = @startOfPredActivity
            SET @endOfPredActivity = @winStart +
                @lastLoginPerWindow
            SET @prevProb = @prob
        ELSE BREAK
    SET @winStart = @winStart + @s
```

7 PROACTIVE RESUME OPERATION

Proactive Resume Algorithm. Given that the resources of physically paused databases are reclaimed, the proactive resume operation is executed as part of the Management Service in the Control Plane of Azure SQL Database. The proactive resume operation is implemented as a periodic activity which runs on a timer for all physically paused databases in one Azure region.

Algorithm 5 Proactive resume operation

```
1: function ProactiveResume()
2:     SELECT @dbs = database_id
3:     FROM sys.databases
4:     WHERE state = 'physical_pause' AND
5:               @now + @k ≤ start_of_pred_activity AND
6:               start_of_pred_activity ≤ @now + @k + 1
7:     for all $d$ ∈ @dbs do
8:         $d$.LogicalPause()
```

Before a database is physically paused, the start of next predicted activity is stored in the metadata store in Line 31 in Algorithm 1. The proactive resume operation accesses this metadata table `sys.databases` and extracts all physically paused databases for which customer activity is predicted to start during k^{th} minute from now in Lines 2–6 in Algorithm 5. The pre-warm time interval of k minutes makes sure that the resources are available before the customer activity returns. Lastly, Algorithm 5 proactively allocates resources for the selected databases in Lines 7–8 by calling the `LogicalPause()` function defined in Lines 13–29 in Algorithm 1.

Diagnostics and Mitigation. The diagnostics and mitigation runner monitors the number of databases in the proactive resume and physical pause queues and the resource allocation and reclamation progress. The runner makes sure that these queues drain and mitigates databases that get stuck during resume or pause. In rare cases, this automatic mitigation process times out or fails, incidents are triggered and resolved by an on-call engineer.

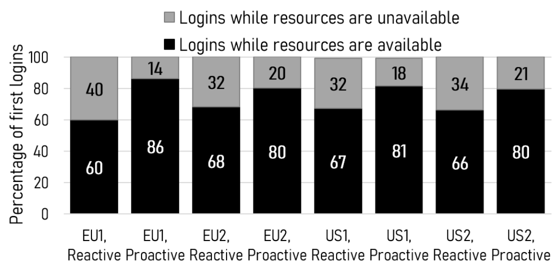
8 KPI METRICS AND TRAINING

KPI Metrics. We measure quality of service (QoS) in terms of the percentage of first logins after idle intervals that occurred while the resources were available (aka proactive resume of resources) versus the percentage of first logins after idle intervals that occurred while the resources were unavailable (aka reactive resume of resources).

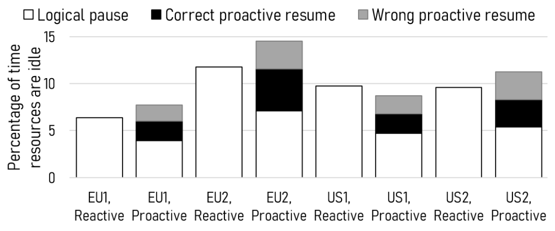
We quantify the operational costs (COGS) in terms of the percentage of time during which resources are idle due to logical pause and proactive resume of resources (Definition 2.2). We classify proactive resumes into correct and wrong. If the customer used the proactively allocated resources, then we consider the proactive resume as correct. Otherwise, the proactive resume is wrong. Even correct proactive resume contributes to idle time since the resources are not used immediately as they become available.

We evaluate the overhead of the online components of the ProRP infrastructure. We measure the storage overhead in terms of the size of database history in kilobytes, the computational overhead in terms of the latency of activity prediction in milliseconds and the frequency of resource allocation and reclamation workflows.

Training Pipeline. As demonstrated by experiments in Section 9.2, the accuracy of customer activity prediction depends on several tunable parameters, including the window size, the confidence threshold, the history length, and the seasonality. To account for potential data drifts over time and prevent accuracy drops, we reset the values of these parameters if better configuration can be

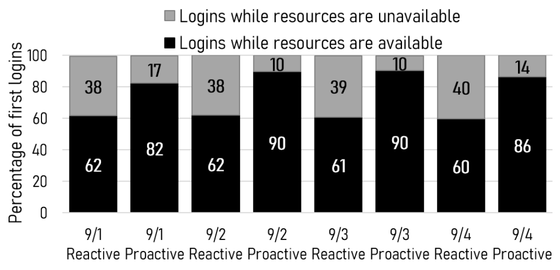


(a) QoS

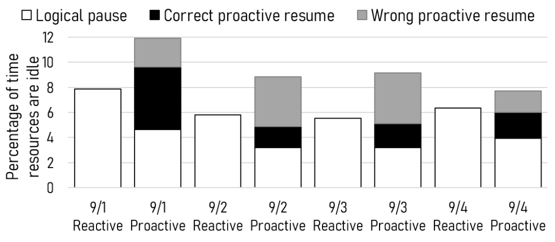


(b) Operational costs

Figure 6: Validation across different Azure regions



(a) QoS



(b) Operational costs

Figure 7: Validation across different training and test intervals

found. To automate and distribute the offline training, we leverage Azure ML that allows us to automatically schedule one run of the training pipeline per Azure region per month [3, 57] (Figure 1). The pipeline accesses several months of customer activity for hundreds of thousands of serverless databases worldwide. The size of the training data is in tens of terabytes. The training data has the same schema as the inference data described in Section 5. The training data is available in the big data platform Cosmos [61]. The pipeline varies the parameters of activity prediction, computes the KPI metrics, and selects the configuration that finds the best middle ground between quality of service and operational cost efficiency. We leverage the existing deployment infrastructure of Azure SQL Database to re-configure the customer activity prediction.

9 EXPERIMENTAL EVALUATION

9.1 Experimental Setup

Implementation. We implemented the ProRP infrastructure in Figure 1 in C++ within the code base of Azure SQL Database.

Production Telemetry. We analyzed several months of production telemetry from the top two largest European Azure regions referred to as EU1 and EU2 and the top two largest US Azure regions referred to as US1 and US2. Hundreds of thousands of Azure SQL databases are currently deployed in these four regions. This telemetry is emitted by the customer activity tracking, the prediction of next activity, and the proactive resume operation per Sections 5–7. Each event carries timestamp in seconds, database identifier, and results of each component of the ProRP infrastructure. Unless stated otherwise, the default experimental Azure region is EU1.

Methodology. We compare our proposed proactive policy to the current reactive policy per Sections 2.2 and 4.

Default Configuration. Unless stated otherwise, next activity is predicted one day ahead based on 4 weeks of history per database. Seasonality is set to daily. The confidence threshold is 0.1. The window size is 7 hours. The window slides every 5 minutes. Resources are proactively resumed 5 minutes ahead of predicted activity. We experimentally configure these knobs in Section 9.2. Duration of logical pause is set to 7 hours per our prior analysis [59].

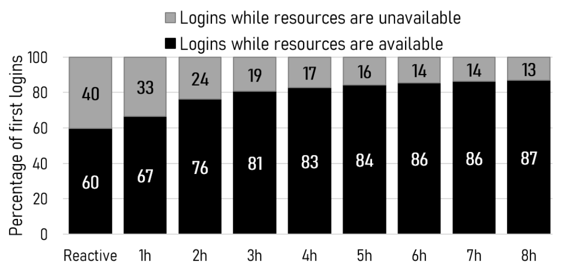
Metrics. We measure the trade-off between quality of service, operational costs, and the overhead of ProRP per Section 8.

9.2 Quality of Service versus Operational Costs

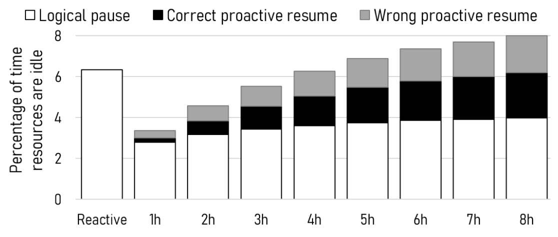
In Figures 6–9, we experimentally compare the current reactive resource allocation policy to our proposed proactive policy (Figure 2). We validate these results across four largest Azure regions EU1, EU2, US1, and US2 in Figure 6. We also validate the results across four evaluation days on September 1–4, 2023 in Figure 7. We tune the parameters of the proactive policy in Figures 8 and 9.

Reactive Policy. The reactive policy always logically pauses resources once the customer activity stops to relieve the backend from frequent scaling operations and improve quality of service.

On the upside, 60–68% of first logins after idle intervals occur during the time intervals when resources are logically paused and customers experience no delay in resource availability (Figures 6(a) and 7(a)). Remaining 32–40% of first logins trigger reactive resume of resources and customers may experience a brief time interval during which resources are unavailable.

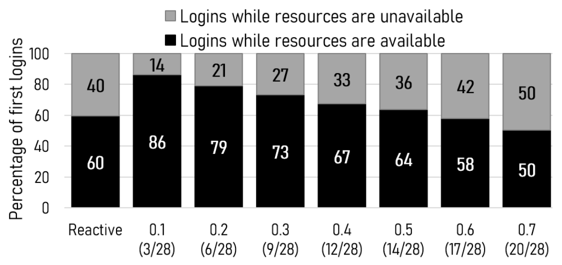


(a) QoS

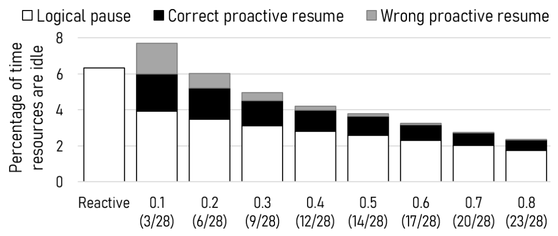


(b) Operational costs

Figure 8: Varying window size (hours)



(a) QoS



(b) Operational costs

Figure 9: Varying confidence of prediction

On the downside, 5–12% of the time resources stay idle and operational costs are wasted during logical pauses (Figures 6(b) and 7(b)). This percentage corresponds to tens of thousands of hours of idle resources for all databases per Azure region and day.

Proactive Policy. To improve both the quality of service and the operational cost efficiency, our proactive policy detects daily activity pattern per database and leverages these patterns to proactively resume resources ahead of predicted activity.

On the upside, thanks to proactive resume of resources, 80–90% of first logins after idle interval occur during time intervals when resources are available (Figures 6(a) and 7(a)). This percentage corresponds to tens of hours of improved customer experience across all databases per Azure region and day.

Moreover, the proactive policy physically pauses the resources of idle databases for which no activity is predicted in the next 7 hours. In this way, the proactive policy reduces resource idleness due to logical pauses to 3–7% of the time (Figures 6(b)–7(b)).

On the downside, wrong proactive resumes contribute 1–4% of the time when resources are proactively resumed but not used by the customers, i.e., resources stay idle. Even correct proactive resume contributes 1–5% of the time when resource stay idle because proactively resumed resources are not used immediately when they become available but only after the customer logs in and uses them. In summary, the improved quality of service is enabled at the cost of slightly higher percentage of time when resources stay idle (7–14% of the time), compared to the reactive policy.

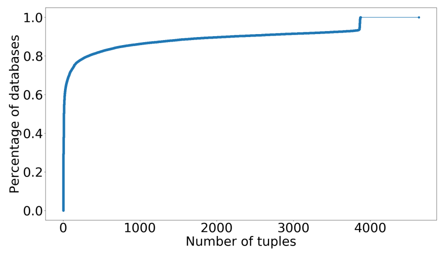
Training. The trade-off between quality of service and operational costs is controlled by the configuration parameters of the proactive policy. For example, as the window size grows from 1 to 8 hours, higher number of historical logins fall into the window, the probability of activity per window increases, and resources are proactively resumed more frequently. Thus, the percentage of first logins that happen during the time intervals when resources are available increases from 67 to 87% in Figure 8(a). However, the percentage of idle time also grows from 3 to 8% in Figure 8(b).

We observe the opposite trends in Figure 9. Namely, as the confidence threshold increases from 0.1 to 0.8, fewer windows satisfy this constraint, and resources are proactively resumed less frequently. Therefore, the percentage of first logins that do not trigger reactive resume of resources decreases from 86 to 50%, while the percentage of idle time reduces from 6 to 2%. In Figures 8 and 9, we prioritize quality of service over operational costs and set the window size to 7 hours and the confidence threshold to 0.1.

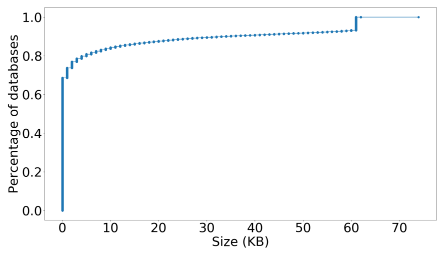
In contrast to Figures 8 and 9, the trade-off between quality of service and operational costs is relatively independent from history length. We set the history length to 4 weeks as a compromise between prioritizing recent history and ensuring periodicity of activity pattern across several weeks. Weekly seasonality achieves similar results to daily seasonality (Algorithm 4). We skip similar charts in this publication due to tight space constraints.

9.3 Overhead of the Online Components

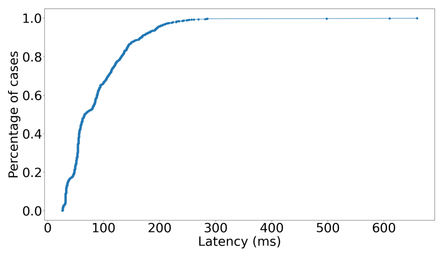
Size of Database History. Figure 10 shows the CDFs that measure the overhead of ProRP. Per Section 5, the start and end of customer activity are stored in an internal table of the database. Given that the lifespan and activity patterns vary per database, the number



(a) Number of tuples in database history



(b) Size of database history



(c) Latency of activity prediction

Figure 10: Overhead of the proactive policy

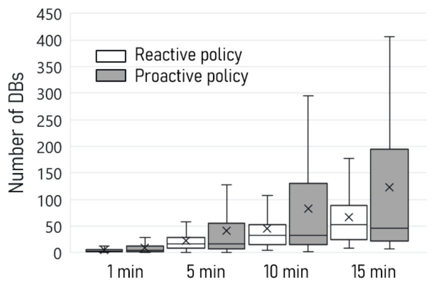


Figure 11: Frequency of resource allocation workflows

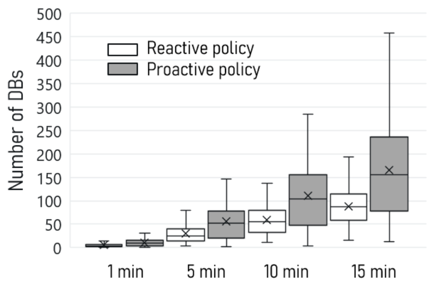


Figure 12: Frequency of resource reclamation workflows

of tuples stored in the database history varies as well. While the average number of tuples stays within 500, the maximal number of tuples can grow over 4K in rare cases in Figure 10(a). Each tuple consists of two integer values of size 64 bits (Section 5). Therefore, the size of database history stays within 7 KB on average and does not exceed 74 KB in the worst case in Figure 10(b).

Latency of Activity Prediction. Given that different size of database history is analyzed in each case and activity patterns vary per database, the latency of activity prediction also varies in Figure 10(c). This latency is within 90 milliseconds on average and does not exceed 700 milliseconds in the worst case.

Based on the experimental results in Figure 10, we conclude that the computational and storage overhead of ProRP is negligible. We are confident that this overhead will have no noticeable negative impact on the performance of customer workloads nor billing.

Frequency of Resource Allocation Workflows. In Figure 11, we measure the number of proactively resumed databases in one iteration of the proactive resume operation in a large Azure region per day on the y-axis. We vary the frequency of the proactive resume operation on the x-axis. The gray box plots illustrate that the maximal number of proactively resumed databases increases from 29 to 406 as the frequency of the proactive resume operation reduces from 1 to 15 minutes. Our goal is to experimentally tune the frequency of the proactive resume operation such that the number of proactively resumed databases does not exceed one hundred in one iteration of the proactive resume operation to keep the overhead of resource scaling mechanisms manageable by the current infrastructure of Azure SQL Database [4]. Therefore, we set the frequency of the proactive resume operation to one minute.

Frequency of Resource Reclamation Workflows. Figure 12 measures the number of physically paused databases per time interval in a large Azure region. The gray box plots illustrate that the maximal number of physically paused databases increases from 31 to 458 as the time interval increases from 1 to 15 minutes. The number of physically paused databases is slightly higher than the number of proactively resumed databases for the same time interval because some of the databases are new and did not accumulate history to predict activity yet. Therefore, proactive resource allocation is not possible for them. Instead, they are resumed reactively and physically paused based on their idle time (Section 4).

The frequency of scaling operations under the current reactive policy is illustrated by the white boxes in Figures 11 and 12. The number of proactive resumes and physical pauses per time interval is doubled by the proactive policy, compared to the reactive policy. This is explained by the fact that the proactive policy skips logical pauses and goes directly into physical pause if no activity is predicted in near future. Higher number of physical pauses causes higher number of proactive resumes. Our stress tests confirmed that the ProRP infrastructure handles this increased workload well.

10 RELATED WORK

The approaches to autonomous database management can be classified by their optimization objectives into physical design improvements, knob tuning, and resource allocation and by their methodology into rule-based, cost-model-based, and machine-learning-based approaches. We briefly summarize each class below.

Optimization Objective. Several approaches focus on the physical design, choice, and tuning of data structures [33, 37] and indexes [22, 23, 43]. Others auto-tune a specific system component such as query optimizer [35, 40, 46, 47] and compiler [19, 24, 34]. Some approaches define database partitioning [36, 52, 55]. Several approaches automate the selection and tuning of the most impactful knobs in database management systems [32, 65, 68]. These knobs include the configuration parameters of the cost model for query optimizer, frequency and granularity of logging, degree of parallelism, etc. These approaches tackle challenges that are orthogonal to the focus of this paper. We consider reusing some of them (Section 11).

There are approaches that automate demand-driven resource allocation for cloud databases. Some of them are merely reactive, i.e., based on the current resource demand [29, 30], others are proactive, i.e., based on both current and predicted demand [12, 21, 45, 49, 51, 56–59, 63, 66]. Predictive provisioning approaches [12, 45, 58, 59, 66] forecast the workload pattern per database, proactively resume resources to guarantee high quality of service, and reclaim idle resources to save operational costs. Resource Advisor [51] predicts resource demand and the effect of resource upgrades. PStore [63] adjusts the number of machines based on predicted load. Toto [49] models disk utilization, database creation and deletion over time to benchmark the efficiency of cloud databases. Picado et al. [56] predict the database survivability to guide the resource provisioning policy. Seagull [57] predicts the customer activity and schedules backups of their databases during expected lowest resource utilization. Doppler [21] analyzes resource utilization to recommend the initial amount of resources to customers who migrate their provisioned databases to the cloud. To the best of our knowledge, ProRP is the first deployed infrastructure that enables continuous proactive resource allocation capabilities for an industrial product running millions of serverless databases worldwide.

Methodology. Rule-based approaches are widely used in industry. They often deploy simple yet accurate statistical or probabilistic workload forecast techniques to analyze historical traces and make decisions based on predefined rules. IBM DB2 [36, 44, 62, 64] uses rules to determine how much memory to allocate to components of database management. Oracle 10g [28, 31] provides a rule-based tool to identify bottlenecks due to mis-configuration. Azure SQL Database [29, 45, 49, 51, 56, 57, 59, 66] observes the load per database for the last few weeks, computes statistics, and defines rules for automated resource allocation.

Cost-model-based approaches explore the search space of possible choices and evaluate their quality using a cost model to select a "good" configuration. A variety of algorithms were applied to explore the search space such as approximation [20, 22], local search [67], greedy search [23], and branch-and-bound [52, 55].

Machine-learning-based approaches are gaining popularity especially in academia in the last decade. iTuned [32] uses a Gaussian Process model to explore the solution space, runs experiments when the database is not fully utilized until the result converges to a near-optimal configuration. BestConfig [68] partitions the parameter search space, randomly selects one point from each partition, and explores the space near the point with the best performance in a sample set. OtterTune [65] uses a combination of supervised and unsupervised methods to select the most impactful knobs, map previously unseen database workloads to known workloads, and recommend knob settings. NoisePage [54] implements machine-learning-based tuning agents to predict the expected benefit of actions that improve database physical design, knob configuration, and hardware resources.

While machine learning models are more accurate than simpler forecast techniques [27, 39, 56–59, 66], we currently deploy the probabilistic forecast algorithm to production due to several practical considerations described in Sections 1 and 3. However, we plan to improve the accuracy of workload prediction in the future.

11 CONCLUSIONS AND FUTURE WORK

In this publication, we have presented the infrastructure for proactive resume and pause of resources shown in Figure 1. The ProRP infrastructure is deployed in all Azure regions to optimize the trade-off between quality of service and operational cost efficiency for millions of Azure SQL Databases. Given the size and scope of our solution, we believe that our design principles and lessons learned generalize to the cloud databases in any company.

Even after several years of work, we are by no means at the end of this journey. Below, we briefly sketch several ideas to further develop the proactive resource allocation capabilities.

(1) The proactive resource allocation policy makes binary decisions so far, i.e., the resources are either allocated or reclaimed for each database (Definitions 2.1 and 2.2). Going forward, we plan to auto-scale the resources in small increments of capacity to better accommodate the current resource demand for each database.

(2) ProRP has multiple configuration knobs (Table 1). So far, we have manually selected the most impactful knobs to tune based on our domain knowledge. However, knob selection can be automated, as defined by the state-of-the-art approaches in academia [32, 65].

(3) The proactive resource allocation policy improves operational cost efficiency if and only if the reclaimed resources are indeed reused. If other databases on the same node are idle or have enough resources to serve their workloads, then the released resources will not be reused, the number of physical machines will not be reduced, and the operational costs will not be saved. Worst yet, the computational overhead of physically pausing the resources and then resuming them again will consume resources and waste operational costs. Therefore, the proactive resource allocation policy must align with the data-driven tenant placement and load balancing algorithms to amplify the business impact.

(4) So far, the proactive policy ignores the system maintenance operations such as backups, software updates, version upgrades, and stats refresh (Section 3.3). In the future, we will schedule these operations when the database is predicted to be online to minimize impact of increased backend load of resuming just for the purpose of running these operations. Furthermore, the prediction will identify time windows when usage from customer workload is low or idle (but still online) and run the system operations then to minimize their performance impact on customer workload [57].

ACKNOWLEDGMENTS

The authors are deeply gratefull to Youyuan Wu, Emilio Chavez Madero, Chenxi Dai, Dhruv Kothari, Dhrub Kumar, and Patrick Pastore for their hard work implementing and optimizing the resource allocation mechanisms. The authors sincerely thank Hanuma Kodavalla and the anonymous SIGMOD reviewers for their insightful and encouraging comments to this publication.

REFERENCES

[1] 2024. ARIMA. https://pypi.org/project/pmdarima/

[2] 2024. Availability Capabilities of Azure SQL Database. https://learn.microsoft.com/en-us/azure/azure-sql/database/sql-database-paas-overview?view=azuresql#availability-capabilities

[3] 2024. Azure ML. https://azure.microsoft.com/en-us/services/machine-learning/

[4] 2024. Azure Service Fabric. https://azure.microsoft.com/en-us/services/service-fabric/

[5] 2024. Azure SQL Database. https://azure.microsoft.com/en-us/products/azure-sql/database

[6] 2024. Azure SQL Database Pricing. https://azure.microsoft.com/en-us/pricing/details/azure-sql-database

[7] 2024. Azure SQL Database Serverless. https://docs.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview

[8] 2024. Clustered and Nonclustered Indexes of SQL Server. https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver16

[9] 2024. GluonTS. https://gluon-ts.mxnet.io/

[10] 2024. MLflow. https://mlflow.org/

[11] 2024. ML.NET Binary Trainer. https://docs.microsoft.com/en-us/dotnet/api/microsoft.ml.trainers.fasttree.fastforestbinarytrainer

[12] 2024. MySQL Autopilot Shape Advisor. https://dev.mysql.com/doc/heatwave-aws/en/heatwave-aws-autopilot-shape-advisor.html

[13] 2024. NimbusML. https://docs.microsoft.com/en-us/python/api/nimbusml/nimbusml.timeseries.ssaforecaster

[14] 2024. Oracle Autonomous Database. https://www.oracle.com/autonomous-database/

[15] 2024. Power BI. https://powerbi.microsoft.com/

[16] 2024. Prophet. https://facebook.github.io/prophet/

[17] 2024. SLA for Azure SQL Database. https://azure.microsoft.com/en-us/support/legal/sla/azure-sql-database/v1_8/

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

[19] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2019. A Survey on Compiler Autotuning using Machine Learning. *ACM Computing Surveys (CSUR)* 51 (2019), 1 – 42.

[20] Nico Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD*. 227–238.

[21] Joyce Cahoon, Wenjing Wang, Yiwen Zhu, Katherine Lin, Sean Liu, Raymond Truong, Neetu Singh, Chengcheng Wan, Alexandra Ciortea, Sreraman Narasimhan, and Subru Krishnan. 2022. Doppler: Automated SKU Recommendation in Migrating SQL Workloads to the Cloud. *Proc. VLDB Endow.* 15, 12 (2022), 3509–3521.

[22] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "What-If" Index Analysis Utility. In *SIGMOD*. 367–378.

[23] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155.

[24] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *Proc. of Int. Symposium on Code Generation and Optimization*. 12–23.

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

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

[27] Carlo Curino, Neha Godwal, Brian Kroth, Sergiy Kuryata, Greg Lapinski, Siqi Liu, Slava Oks, Olga Poppe, Adam Smiechowski, Ed Thayer, Markus Weimer, and Yiwen Zhu. 2020. MLOS: An Infrastructure for Automated Software Performance Engineering. In *DEEM@SIGMOD*. 1–5.

[28] Benoît Dageville and Mohamed Zait. 2002. SQL Memory Management in Oracle9i. In *VLDB*. 962–973.

[29] 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.

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

[31] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle. In *CIDR*. 84–94.

[32] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with ITuned. *Proc. VLDB Endow.* 2, 1 (August 2009), 1246–1257.

[33] Jonathan Eastep, David Wingate, and Anant Agarwal. 2011. Smart Data Structures: An Online Machine Learning Approach to Multicore Data Structures. In *Proc. of Int. Conf. on Autonomic Computing*. 11–20.

[34] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher Williams, and Michael O'Boyle. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *Int. Journal of Parallel Programming* 39 (06 2011), 296–327.

[35] Sunny Gakhar, Joyce Cahoon, Wangchao Le, Xiangnan Li, Kaushik Ravichandran, Hiren Patel, Marc Friedman, Brandon Haynes, Shi Qiao, Alekh Jindal, and Jyoti Leeka. 2022. Pipemizer: An Optimizer for Analytics Data Pipelines. *Proc. VLDB Endow.* 15, 12 (September 2022), 3710–3713.

[36] Michael Hammer and Bahram Niamir. 1979. A Heuristic Approach to Attribute Partitioning. In *SIGMOD*. 93–101.

[37] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *SIGMOD*. 535–550.

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

[39] Alekh Jindal, K. Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas C. Müller, Wentao Wu, and Hiren Patel. 2021. Magpie: Python at Speed and Scale using Cloud Backends. In *CIDR*.

[40] Alekh Jindal and Jyoti Leeka. 2022. Query Optimizer as a Service: An Idea Whose Time Has Come. *SIGMOD Record* 51, 3 (2022), 49–55.

[41] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. 2018. Service Fabric: A Distributed Platform for Building Microservices in the Cloud. In *EuroSys*. 1–15.

[42] Arnd Christian König, Yi Shan, Tobias Ziegler, Aarati Kakaraparthy, Willis Lang, Justin Moeller, Ajay Kalhan, and Vivek Narasayya. 2022. Tenant Placement in Over-subscribed Database-as-a-Service Clusters. *Proc. VLDB Endow.* 15, 11 (2022), 2559–2571.

[43] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504.

[44] Eva Kwan, Sam Lightstone, K. Bernhard Schiefer, Adam J. Storm, and Leanne Wu. 2003. Automatic Database Configuration for DB2 Universal Database: Compressing Years of Performance Expertise into Seconds of Execution. In *BTW*, Vol. 26. 620–629.

[45] 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.

[46] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Learning to Steer Query Optimizers. In *SIGMOD*. 1275–1288.

[47] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718.

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

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

[50] Kunal Mukerjee, Tomas Talius, Ajay Kalhan, Nigel Ellis, and Conor Cunningham. 2011. SQL Azure as a Self-Managing Database Service: Lessons Learned and Challenges Ahead. *IEEE Data Eng. Bull.* 34, 4 (2011), 61–70.

[51] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. 2005. Continuous Resource Monitoring for Self-predicting DBMS. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 239–248.

[52] Rimma Nehme and Nicolas Bruno. 2011. Automated Partitioning Design in Parallel Database Systems. In *SIGMOD*. 1137–1148.

[53] 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*.

[54] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. 2019. External vs. Internal:

An Essay on Machine Learning Agents for Autonomous Database Management Systems. *IEEE Data Eng. Bull.* 42, 2 (2019), 32–46.

[55] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*. 61–72.

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

[57] 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.

[58] Olga Poppe, Pablo Castro, Willis Lang, and Jyoti Leeka. 2023. Proactive Resource Allocation Policy for Microsoft Azure Cognitive Search. *SIGMOD Record* 52, 3 (2023), 41–48.

[59] Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. 2022. Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless. *Proc. VLDB Endow.* 15, 6 (2022), 1279–1287.

[60] Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and Dan Dougherty. 2016. Context-aware Event Stream Analytics. In *EDBT*. 413–424.

[61] Conor Power, Hiren Patel, Alekh Jindal, Jyoti Leeka, Bob Jenkins, Michael Rys, Ed Triou, Dexin Zhu, Lucky Katahanas, Chakrapani Bhat Talapady, Josh Rowe, Fan Zhang, Rich Draves, Ivan Santa, and Amrish Kumar. 2021. The Cosmos Big Data Platform at Microsoft: Over a Decade of Progress and a Decade to Look Forward. *Proc. VLDB Endow.* 14, 12 (2021), 3148–3161.

[62] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive Self-Tuning Memory in DB2. In *VLDB*. 1081–1092.

[63] 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.

[64] Wenhu Tian, Pat Martin, and Wendy Powley. 2003. Techniques for Automatically Sizing Multiple Buffer Pools in DB2. In *CASCON*. 294–302.

[65] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *SIGMOD*. 1009–1024.

[66] 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.

[67] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. 2004. A Smart Hill-Climbing Algorithm for Application Server Configuration. In *WWW*. 287–296.

[68] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proc. of Symposium on Cloud Computing*. 338–350.