# Toto - Benchmarking the Efficiency of a Cloud Service

Justin Moeller, Zi Ye, Katherine Lin, Willis Lang
Microsoft
{jumoell},{zi.ye},{katlin},{wilang}@microsoft.com

## ABSTRACT

Microsoft aims to increase the efficiency of Azure SQL DB by maximizing the number of databases that can be hosted in a cluster. However, resource contention among customers increases when changing the configurations, policies, and features that control database co-location on cluster nodes. Tuning and evaluating the efficiency and customer impact of these variables in a scientific manner in production, with a dynamic system and customer workloads, is difficult or infeasible. Here, we present Toto, a benchmark framework for evaluating the efficiency of any cloud service that leverages orchestrators like Service Fabric or Kubernetes. Toto allows for reliable and repeatable specification of a benchmarking scenario of arbitrary scale, complexity, and time-length. An implementation of Toto is deployed in all SQL DB staging clusters and is used to evaluate system efficiency and behaviors. As an example of Toto's capabilities, we present a study to explore the balance between cluster database density and quality of service.

## CCS CONCEPTS

• **Information systems → Data management systems**; • **Software and its engineering → Cloud computing**.

## KEYWORDS

Cloud Databases; Benchmarking; Efficiency; Orchestration

**ACM Reference Format:**
Justin Moeller, Zi Ye, Katherine Lin, Willis Lang. 2021. Toto - Benchmarking the Efficiency of a Cloud Service. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457555

## 1 INTRODUCTION

Microsoft Azure SQL DB is engaged in a perpetual effort to increase the efficiency of running the service by maximizing the number of customer databases that can be hosted on a fixed-size cluster of servers. The cost of these clusters is already minimized due to the purchasing power of one of the dominant public cloud providers. As such, the ability to place more databases given a fixed capex cost is paramount. Additionally, the need to get the most mileage out of the cluster is not always driven from the bottom line, but out of practical necessity. For instance, delays from hardware vendors due to the scale of orders may also force Azure to seek efficiency.

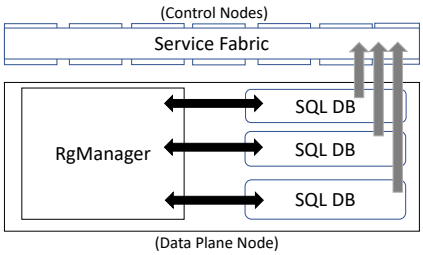



**Figure 1: SQL DB simple flow/communication architecture.**

In an ideal system, the operating density of databases in a cloud database cluster is directly tied to the degree of database underutilization. (Consequently, this relationship drives the technology behind the new SQL DB Serverless offering.) In a complex system such as SQL DB, there are numerous parameters, policy thresholds, and improvement features that can impact the number of databases that can co-exist on a node in a cluster, all the while maintaining the highest levels of service quality (QoS). The question for a service provider is: How can one accurately and reliably evaluate the effect of different policies and mechanisms on the degree of database density in the cluster and simultaneously measure the effect on the QoS? Here, we present our answer – ***Toto*** – a benchmark framework to help reveal the impact of changes on a cloud service.

The problem here is twofold: a service provider needs a way to compare the effects of a change (e.g., a tuning parameter) in a scientific manner - reliably and repeatably - and representative of the true production environment. To evaluate a change, we can either do so on two comparable clusters running side-by-side, or on a single cluster and compare the before-and-after key performance indicators (KPIs) (The granularity of deploying changes in SQL DB is at a cluster level). For example, to fairly evaluate, we must replicate the database population ratios, the databases' properties (e.g., size), the databases' workloads, and the system itself (e.g., deployed features and hotfixes). To control variables like these without negatively impacting customer experience may be impossible (or prohibitively costly). Consequently, we introduce Toto as a cloud service framework whereby we can deploy and evaluate the code base and avoid negative customer impact. Toto is part of an active, day-to-day effort to increase efficiency within SQL DB. We are using Toto to: (a) evaluate production configuration changes in SQL DB before they deploy (e.g., buffers, placement policies), (b) quantify the benefits of proposals (e.g., what-if), and (c) debug ("repro") problems from the production clusters.

The system framework we describe focuses on benchmarking a single staging cluster (SQL DB regions are made up of hundreds of clusters), by focusing on the node-level systems that govern the resources and performance of the database engines (deemed "RgManager") and the cluster orchestration system (e.g., Microsoft


*SIGMOD '21, June 20–25, 2021, Virtual Event, China*

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457555

Service Fabric – **SF**, or Kubernetes). Figure 1 presents a very simple diagram of these components and interactions. For our purposes, benchmarking a database service for efficiency is less focused on the capabilities of the database query optimizer or processing engine, but rather, how well systems like RgManager and **SF** can (i) control the resources provided to the DB engines, and (ii) effectively collocate databases, respectively. Therefore, at its heart, Toto has the goal to present thousands of SQL "workloads" to these two systems so that they react to maximize cluster efficiency (see Section 3).

For Toto, the workloads are not SQL query workloads or traditional RDBMS performance benchmarks. We list three reasons for this. First, customer databases generally do not exhibit full-bore performance workloads such as TPC-E/H, but are rather low utilization or bursty, or completely idle (which is where the efficiency opportunity is derived) – see Section 2. Secondly, we wish to *avoid reverse engineering SQL workloads based on observed utilization traces in the hopes that they can recreate those utilization traces.* (Providers like Microsoft neither have access to customer queries nor data details.) Finally, we wish to avoid the complexity and cost of relying on SQL drivers, coordinated and hosted on sufficient compute power.

Toto was conceived based on *existing* Azure SQL DB production debugging mechanisms that force the infrastructure to perceive "fake" resource load. This existing mitigation mechanism is used because, through the lens of the service, the job of the database engine is to transform SQL queries into resource requests to be consumed by RgManager and Service Fabric (SF). While this mechanism for "fake" resource load is currently Azure SQL DB and SF specific, the Toto framework is applicable to any orchestration system that manages customly defined resources (e.g., Kubernetes [4]). The job of RgManager and SF is to manage the database engine instances and cluster nodes based on these resource signals. We should note that this is a distinction between Toto and component simulators – Toto reveals full-stack impact. Toto utilizes production-derived models of database resource consumption and customer behavior (i.e., DDL) and produces request streams *on behalf* of the SQL engine instances. Other important models for SQL DB include the service tier and SLO configuration options of the database population. Given that Toto aims to benchmark efficiency, the *temporal richness* of the modeling matters just as much, if not more, than amplitude of the signals. For example, business hours and week days must be treated differently than evenings or weekends. Toto consumes declaratively specified models and parameters, allowing us to easily (re)specify a benchmark scenario of arbitrary *scale, complexity, and time-length* and target any SQL DB cluster. (At this point, we hope that experienced cloud practitioners have considered that Toto can apply to almost *any* Kubernetes-orchestrated cloud service.)

To demonstrate evaluating service efficiency, we present an example of a density tuning study of a SQL DB stage cluster using models of production databases and customer behavior. If a cluster's density is tuned high enough, the provider begins to face significant challenges [34, 35, 56]. Different SQL DB offerings (e.g., Serverless) and "SLO" configurations have distinct maximum CPU and disk quotas (ignoring other resources for this example). A node at high density risks a scenario where *if* all of the databases on the node consumed their *actual* quota limit of a resource like disk, the node may not have the disk capacity to dispatch a given request. If a node in the cluster hits its logical capacity limits (also a tunable

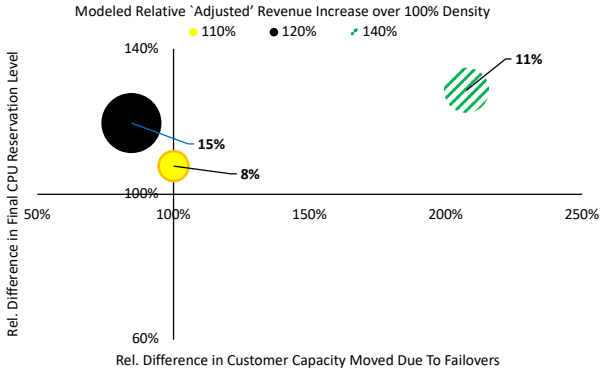



**Figure 2: Toto result on a SQL DB cluster, circle size reflects relative "adjusted" revenue over a 100% cluster density.**

parameter) due to resource consumption by the databases placed on it, then one or more of the databases must be moved to another node (suffering a *failover*). This *may cause* a brief moment of unavailability and varying performance. *It is the job of the system to intelligently manage the likelihood of failovers happening on any node through placement, resource governance, and balancing.*

Figure 2 presents a result from our implementation of *Toto* showing the balance between database density on the cluster, a quantification of failovers, and the modeled "adjusted" revenue. This notion, *adjusted revenue*, is a means to normalize density and failovers (see Section 5.1). Here, we tuned a single parameter in a real SQL DB gen5 stage cluster (currently the predominant SKU) which enables/controls the degree of allowable DB density (see Section 5) and using *Toto*, we evaluate this using a specific benchmark scenario that models clusters in a US region of Azure (see Section 4). This means more databases can be admitted into the cluster over a given setting (i.e., 100%). Depending on the density level and the database demographics (e.g., edition and performance configuration), different resources may become the bottleneck for the cluster. On the y-axis, we plot the final cluster CPU reservation level that was achieved during the benchmark (relative to a 100% result). On the x-axis, we plot the relative amount of customer capacity (in cores) that had to be moved around in the cluster to adjust for when high density resulted in a database temporarily needing to wait for resources it has requested. The size of the circles represent the relative change in adjusted revenue from the databases in the cluster. For instance, here we see that at 140% density, CPU reservation level has increased as expected, however, the amount of failover movement is higher while the adjusted revenue is lower than seen at 120% density. *The takeaway is, using Toto, we have been able to quantify the dollar impact of improvements in SQL DB down to the locale and scenario that may otherwise be infeasible to determine.*

Here, we present Toto, which we believe to be the first presentation of a public-cloud, production-oriented benchmarking framework. Toto allows for declarative benchmark submission defined by different models of customer and workload behavior to reliably and repeatably evaluate different service settings and configurations. *Toto can be applied to any cloud infrastructure that is built upon orchestrators like Kubernetes or Service Fabric.* Our implementation of Toto is integrated into the resource governance component of Azure SQL DB, deployed worldwide, and provides a means to benchmark the efficiency of SQL DB. Our contributions are:

**(1)** A first-of-its-kind benchmarking framework – Toto – to evaluate the cost-efficiency of a cloud service that uses orchestrators like Kubernetes and Service Fabric.
**(2)** An implementation of Toto deployed within Azure SQL DB.
**(3)** A description of key service and database utilization models trained from production environments that can create benchmark scenarios and employed by Toto.
**(4)** Presented a density study using our Toto implementation and scored using "adjusted" revenue that considers actual SQL DB SLAs.

## 2 BACKGROUND

**Rings and SKUs:** At the time of this paper's preparation, Azure runs 55 regions worldwide with ten more announced. Each region can be thought of as one or more physical datacenters and each datacenter housing hundreds to thousands of clusters (or rings) of nodes. Different Azure services such as SQL DB occupy different rings (while some may share). SQL DB rings vary in their size but can be thought of in the range of 50-150 nodes. SQL DB rings can also be considered homogeneous in their hardware SKU, such as currently gen4, gen5, and gen6 SKU. Different hardware generations differ in their compute (CPU), memory, local storage, and networking power. More importantly, they vary in their different resource ratios; for instance, the CPU cores to memory ratio, or the memory to local storage ratio are different from generation to generation (as the prices of different commodity components change over time). Resource ratios plays an outsized role in determining the efficiency of SQL DB clusters as it must be in alignment with the customer's database resource demands or unused resources will be "stranded" and efficiency will suffer.

**SQL DB Editions:** Categorizing SQL DB databases and their configurations can first be done according to where the data is stored. Remote-store databases include editions like "Standard DTU" and "General Purpose VCore" (GP) and these databases have their SQL data and log files stored remotely from the compute node. Local-store databases include editions like "Premium DTU" and "Business Critical VCore" (BC) and the database files are stored on the compute node local SSDs [3]. For redundancy, these local-store databases are also replicated four times on four different compute nodes. The Service Level Objectives (SLOs) in each edition and hardware SKU have different configurations such as the amount of compute units (cores) or the amount of DRAM memory available to the SQL process. The performance of query processing on local-store databases outpaces that of similarly configured remote-store databases, but from a provider perspective, it comes at higher cost (and revenue) due to local SSD and replication.

**Resources:** While the SQL engine processes T-SQL queries, the rest of the SQL DB infrastructure stack (e.g, Figure 1) is only concerned with the resources it is consuming and releasing. In addition to the core DBMS SQL engine, Azure SQL DB contains components that manage the resource governance of the SQL engines. The main resources that are considered are CPU consumption, DRAM memory consumption, and disk consumption for data storage. While CPU and memory resources may be straight forward, the disk resource is a little more nuanced, especially given the SQL DB editions discussion above. For the SQL DB infrastructure, the local disk capacity consumption is of utmost importance because it is not transiently consumed. Further, the local storage configurations of Premium/BC-VCore databases have a high maximum allowable capacity which consumes a significant fraction of a single machine. For local-store databases, the disk consumption includes all data and log space used as well as *tempDB* storage capacity (e.g., spilling to disk during QP), while for remote-store databases, only the tempDB storage contributes to its local disk resource consumption.

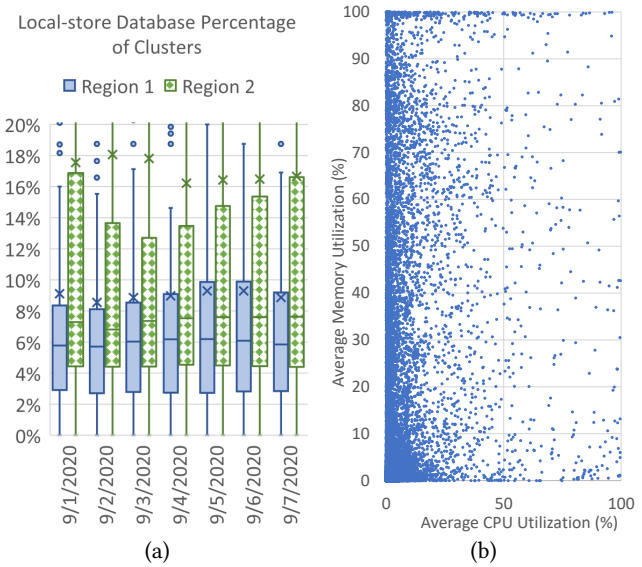



**Figure 3: (a) Daily percentage of DBs that are local-store. (b) CPU and Memory utilization levels of DBs in a region.**

**Production Environments:** Evaluating any efficiency-oriented changes in production is extremely challenging. In Azure SQL DB, code and parameter changes are deployed in a cluster-by-cluster basis. For a change to be evaluated, either we would perform a before-and-after analysis on a single cluster, or a side-by-side A/B test on two clusters. A before-and-after analysis on a single cluster is simply not practical: database population demographics on the cluster may change over time, and the databases themselves are largely growing over time as well. In a before-and-after production analysis, we cannot "rewind" the production cluster back to a starting state, though this is what we achieve with Toto.

Identifying two similar clusters for side-by-side evaluation of a change is non-trivial, if not as impractical as a before-and-after evaluation. For instance, there are distinct regional differences in workloads and edition/SLO demographics. Consider Figure 3(a), which shows the dispersion of the local-store database fraction of each cluster's population for two different Azure regions over a week. The X's on the box plots shows the average percentage over all clusters of the regions and clearly, we see that Region 2 has a significantly larger proportion of local-store databases than Region 1. Still, it may be possible to find two clusters with similar database "demographics", but during the evaluation, we would need these two clusters to exhibit the same workload/growth behavior as well as have similar database create assignments (and drops). Evaluation in this way is also practically prohibitive.

**Representing Workloads:** Most databases in the public cloud have low utilization levels and do not exhibit the resource consumption behavior of running full-bore TPC-x benchmarks. Figure 3(b)

shows the average database CPU and memory utilization level in a single Azure region over a 12 hour daytime period (we have removed all of the completely idle databases - a substantial number). It is clear that a large proportion of databases have low CPU and memory utilization and thus we avoid using TPC-x benchmarks in our benchmarking framework. We have chosen to design an efficiency benchmarking framework for a database service around resource consumption, in part, because it allows us to use telemetry data to produce the most production-representative load to impose onto the service and cluster infrastructure.

# 3 BUILDING INTO SQL DB INFRASTRUCTURE

Our implementation of Toto is composed of two components - an orchestrator that is built into the heart of the Azure SQL DB's resource governance stack (Section 3.3.1) and the Population Manager (Section 3.3.3) that calls public CRUD APIs. Together, these components are "the man behind the curtain", instructing when new databases should be created, when databases are dropped, and what each database's resource usage levels currently are. Here we describe the existing Azure SQL DB infrastructure and how we implemented Toto inside of it.

## 3.1 Service Fabric

Azure SQL DB runs on top of an orchestration framework called Service Fabric (SF). Similar to other container management systems like Kubernetes [13], Service Fabric is a distributed platform for deploying microservices and/or containers [30]. Service Fabric is responsible for management of the microservices in the cluster and it handles common challenges in cloud deployments such as availability, resource management, application lifecycle, and load balancing. Azure SQL DB regions are broken up into many Service Fabric clusters, each cluster either hosting control services (aka "control rings") or data plane services (aka "tenant rings"). When a customer wishes to create a new database, after a cluster is chosen, the request is forwarded to the cluster's Placement and Load Balancer (PLB), a component of SF that decides the placement and movement of databases. Depending on the replication factor of the microservice (see Section 2), the PLB will distribute the replicas across different nodes in the cluster. The PLB is responsible for maintaining the availability of single replica databases and multi-replica databases.

Every orchestration framework needs to be informed of application load so that it can make smart placement decisions and move replicas when nodes become heavily loaded. The PLB in Service Fabric addresses this with the notion of *dynamic load metrics* [2]. A metric can be arbitrary and model anything, but usually they model system resources such as CPU, memory, and disk. Every replica of the application reports their load metrics to the PLB where it aggregates a centralized view of the load on each node. For example in SQL DB, each replica reports the amount of disk space it is using to the PLB.

The replica's metrics are dynamic - they can change over time. Each replica in the cluster is responsible for reporting to the PLB when its load has changed or at some regular interval. Each resource metric has a predefined node-level logical capacity, which specifies the load threshold at which PLB will initiate a *failover*. A failover means that the replicas' aggregate resource demands on the node have exceeded the node's predefined logical capacity. In order to ensure that all customer's resource requirements are met, a replica must be moved out of the heavily loaded node. PLB will select a replica on the heavily loaded node and move it to another node in the cluster. For Premium/BC databases, a replica will need to be built on a new node which involves physically copying over the data from another available replica, whereas the data for a Standard/GP database can be accessed by detaching and reattaching the remote storage. Furthermore, while a failover to the primary is occurring, the application may experience a brief moment of unavailability while a secondary replica is becoming the primary or a new primary replica is built. For Azure SQL DB, the logical resource capacities of each node have been set conservatively to ensure that each replica is getting their share of each resource. However, as the density in our clusters increases, the probability that a failover will occur also increases. This may manifest itself as failed queries, dropped connections, or failed login attempts. Minimizing failovers in Azure SQL DB clusters is paramount in ensuring a good customer experience.

## 3.2 RgManager

In Azure SQL DB, there exists a helper service called RgManager, which is deployed for managing the resource metrics that each SQL database instance reports to PLB. There is a single RgManager instance running on every node in the cluster. RgManager contains a centralized view of the node and is responsible for governing the node's resources and mitigating potential noisy neighbor performance issues. However, it is the responsibility of each individual database to report their own load to the PLB. RgManager provides an internal API for retrieving the load metrics of a database. The SQL engines always communicate with the co-located RgManager instance. As shown in Figure 1, when a replica for a SQL database needs to report its CPU, memory, and disk usage to PLB, it first consults RgManager by issuing an RPC. RgManager does the work to compute and account the database's resources. These values are then returned to the SQL replica so they can be reported to the PLB.

## 3.3 Toto

*3.3.1 Orchestrator.* We implemented Toto to leverage the existing Azure SQL DB infrastructure by redirecting the metric request RPCs in RgManager to sample from defined models instead of returning the actual resource utilization. These models were trained on Azure SQL DB telemetry and they capture production resource behavior (see Section 4). Models can be specified for any resource and any subset(s) of databases and are provided as input to RgManager via an XML blob. Figure 4 shows Azure SQL DB's architecture modified to include Toto's orchestrator.

First, the models and respective parameters that were trained on the production telemetry are serialized into XML format and written into Service Fabric's Naming Service. Naming Service is a highly available metastore database in Service Fabric. In production today, Azure SQL DB uses it to store metadata about the services that are running in the cluster. In our implementation of Toto, RgManager reads the model XML every 15 minutes from Naming Service, parses them, and constructs internal model objects. These model objects

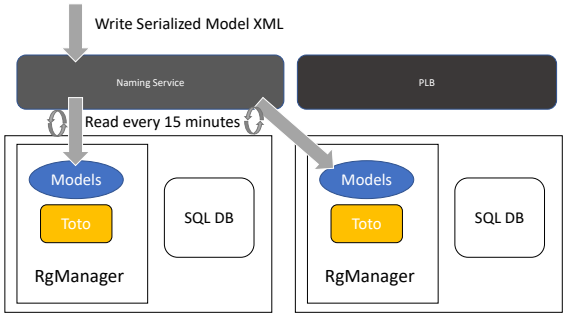



**Figure 4: Injection of the models in RgManager.**

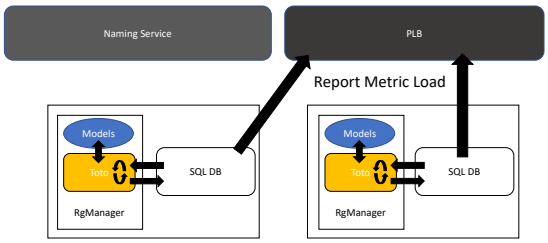



**Figure 5: Toto intercepting metric load requests.**

contain a description of the resource they are modeling, the set of databases it applies to (e.g., all remote store databases), and the periodicity of reporting resource load to the PLB.

Next, when a SQL replica needs to report its metric loads to PLB, it will still issue an RPC call to RgManager, but now RgManager will consult the models to compute the load it should report. This is represented in Figure 5. If no model exists for the replica and the load metric that is being reported, the replica's actual load usage will be reported – this is the normal operating behavior. Otherwise, the supplied models will be used to compute the value to report to the PLB. Because RgManager is stateless, all of the model objects are stateless as well. This allows the model objects to be updated without losing context of how to report the next load metric. The logic to sample from the models is directly coded into RgManager, so sampling is fast and efficient. Building the model execution directly into RgManager allows for declarative and dynamic resource behavior. The models can be updated by overwriting the XML in Naming Service. Tweaking the growth behavior of subsets of databases (e.g., grow disk usage of Premium/BC replicas 2x faster) is easily configurable simply by changing XML properties. These models are scalable and can be applied to many databases at once that exhibit similar resource patterns.

*3.3.2 Imitating Production Resource Behaviors.* We implemented Toto to override the resource behavior of specific metrics by consulting the model objects constructed from serialized XML, but for our purposes this is not sufficient to ensure realistic, production-like behavior. This is because, as mentioned above, the model objects are stateless – in our implementation they described how a particular metric's load changes, but they do not persistently track the previously reported metric value. Without persistently storing the previously reported metric load, on an application failover from one node to another, the newly promoted primary will not know what the previously reported metric load was (the new replica will be communicating with a different RgManager instance). For some resources (e.g., memory/CPU), this is fine and is the expected behavior (i.e., for the load to be completely reset upon failover). For example, in production after a failover the memory load of a newly promoted primary will be smaller than the memory load of the previous primary (because the new primary wasn't servicing queries before). Consequently, for our implementation of memory modeling in Toto, it is sufficient to sample from the model object using a default memory load value that describes a cold buffer pool. (For accuracy, models for resources like CPU and memory need to be distinct for the primary and secondary replicas in local-store Premium/BC databases.)

However, for the disk metric this load reset on failover will lead to unexpected behavior in Toto. For instance, the production disk usage behavior of local-store databases and remote-store databases differs during a failover. Each replica of a local-store database stores a local copy of the data. This means that the disk usage of the secondaries will be very close to the disk usage of the primary (modulo data in transit). Standard/GP databases only have one replica, store all of the user data in remote storage (which is also redundantly replicated), and only use the local disk for temporary data stored in tempDB. On a failover, a new replica will be built, but the data stored in tempDB will be lost. Because of this, the disk usage pattern of Standard/GP databases is similar to memory, where the load is reset after a failover. This is not the case for local-store databases and we needed to make their disk models stateful to ensure the correct behavior.

Our implementation of Toto captures these stateless/stateful nuances by allowing persistence to be a configurable parameter in the model XML. This allows disk usage for remote-store databases to be configured as non-persisted whereas disk usage for local-storage databases can be configured as persisted. When a metric is defined as non-persisted, RgManager will store the previously reported value in memory. To durably store the previous reported value, we leveraged the Service Fabric Naming Service again. After executing the persisted metric model's logic, the new metric load is written back to Naming Service. On the next report interval, the previous metric load will be read from Naming Service and used for the computation of the next load. For persisted disk usage, in order to ensure that only one replica is ever updating the load in Naming Service, only the primary replica executes the model and persists the load. Secondary replicas for Premium/BC databases read the previously reported disk usage from Naming Service, but they do not execute the model's logic. Secondaries simply report the disk usage read from Naming Service. In our implementation this guarantees that on a failover, the newly promoted primary of a Premium/BC database will have the same disk usage as the previous primary replica, which is the same behavior that is exhibited in production.

*3.3.3 Population Manager .* The orchestrator component of Toto allows for declaratively overriding resource utilization via models, but in order to re-create a cluster environment similar to production, there needs to be churn in the number of databases in the cluster. The database population in an Azure SQL DB cluster is

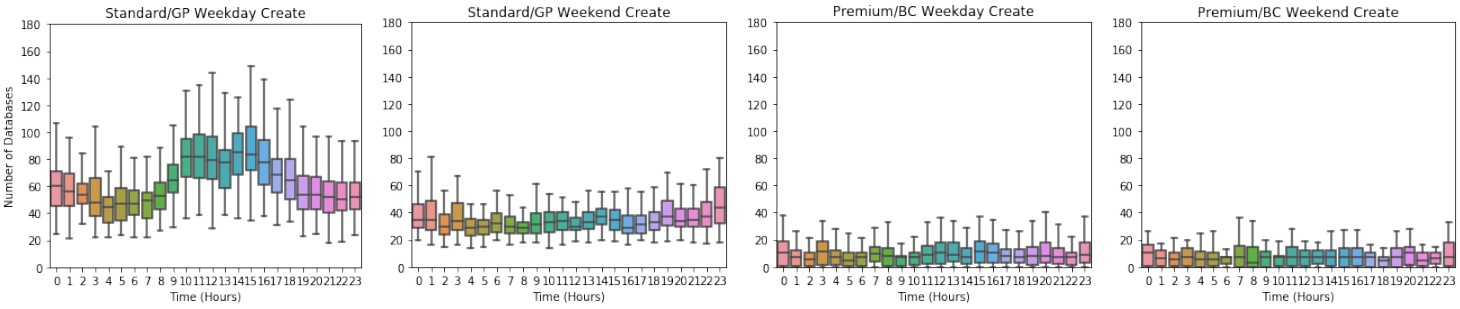



**Figure 6: Dispersion box plots of number of creates/hour of the day: Standard/GP (a, b), Premium/BC (c, d).**

constantly changing, databases are created and dropped regularly. Toto imitates this churn by injecting new databases into the Service Fabric cluster and dropping existing databases via the *Population Manager*. For example, in this study, the Population Manager executes create and drop requests according to weekday/weekend models trained on production creation/drop rates (see Section 4). The Population Manager's models describe how many databases to create/drop per hour, the service tier/edition and the Service Level Objective (SLO) of the databases to create, and the initial metric load for each database. The Population Manager runs as a stateless daemon - it wakes up at the top of each hour to execute, samples from the provided models, then schedules create or drop requests for the next hour. Each create and drop request will then call the corresponding control plane API with the provided metadata (e.g., Create a 4-core local store database at 5:37pm).

## 4 BUILDING MODELS OF BEHAVIOR

To imitate production resource behaviors, we built two modeling frameworks:

- *Create DB and Drop DB model*: We attempted to capture the create and drop patterns of Azure SQL DB by modeling them as events that occur with a probability that belongs to a distribution. Based on empirical observations that create database events exhibited different patterns than drop events, we modeled them separately. (*Note that the Create DB model incorporates the SLO which implicitly captures "CPU reservation" for provisioned SQL DB, which is an input for all placement decisions.*)
- *Disk usage model*: Since all databases persistently store data (even idle databases), disk usage is the most important resource to model in our clusters. Other resource usage models such as memory and CPU usage are left as future work (see Section 5.5). In this paper, we modeled various types of disk usage patterns separately.

In our implementation, the Create DB and Drop DB models needed to be executed by the Population Manager and the disk usage model needed to be executed by RgManager. As such, we preferred a model execution framework that was scalable, easy to implement in C++, computationally inexpensive and fast, did not rely on external libraries, and could still capture the production patterns accurately.

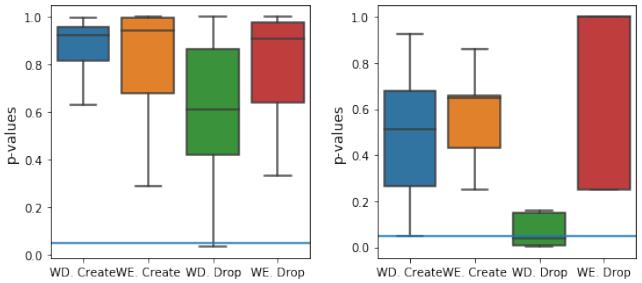



**Figure 7: Dispersion of p-values from K-S test: Standard/GP (a), Premium/BC (b). The blue straight line is the significance level at $\alpha = 0.05$. WD = Weekday, WE = Weekend.**

We explored various machine learning (ML) and statistical modeling approaches [12, 23, 26, 32, 38, 44, 50, 54, 59]. Our initial exploratory work on ML approaches, including random forest [12], linear regression [23] and ARIMA [54], suggested that the ML model accuracy was comparable with statistical approaches. The choice of a modeling approach was not only determined by its accuracy, but also by other factors such as scalability, as described above. For example, ARIMA is computationally intensive since the model needs to search the optimal values of several parameters and that can make the model fitting process very time consuming. Due to the complexity of integrating external libraries into RgManager and the Population Manager as well as other practical considerations, we decided to use simple statistical models as the building blocks for our modeling frameworks.

### 4.1 Create DB and Drop DB Model

*4.1.1 Overview.* The Create DB and Drop DB models aim to imitate the number of net creates of databases within a fixed time interval. When creating a database, the customer can select the Azure region where the database is going to be hosted, but it is the responsibility of the control plane to select the specific tenant ring. For this analysis, modeling the tenant ring selection logic of the control plane was out of scope. Because of this, we built the Create DB and Drop DB models using the create and drop events at the region level. We made a simplifying assumption that each tenant

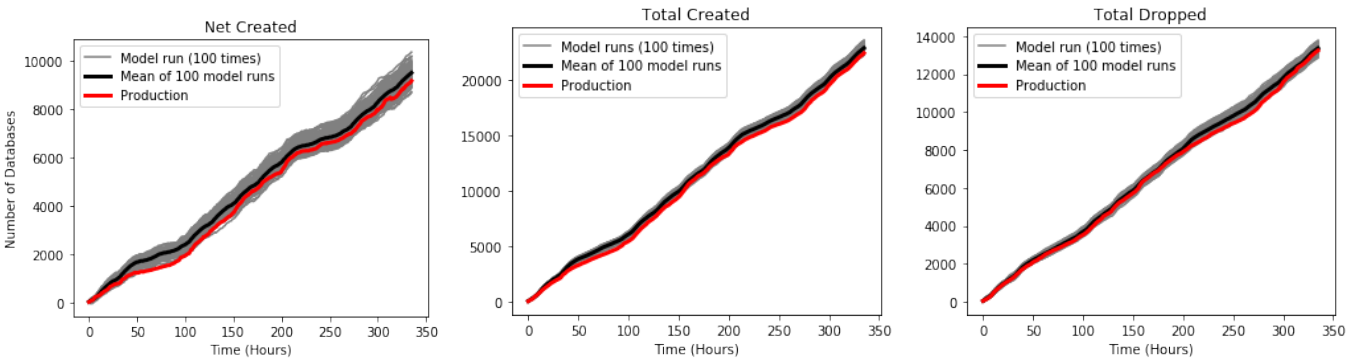



**Figure 8: Region-level results of Create DB and Drop DB models: net creates (a), number of creates (b), number of drops (c).**

ring in a region had equal probability of being selected and scaled the values of the model parameters by the total number of tenant rings within that region.

*4.1.2 Data Analysis.* When modeling creation/drop probabilities, if the analysis was performed on the granularity of seconds or a minute, there would be a low probability of a create or drop event occurring. Therefore, we expanded the aggregation time interval to one hour for both the Standard/GP and Premium/BC databases. The number of creates and drops were modeled separately based on a set of features derived from hourly aggregated production data (Figure 6). Due to space constraints, the figures of drop events were omitted but the patterns were similar to the create events. Below is a summary of the key features:

(1) The number of creates and drops exhibited hourly patterns for both Standard/GP or Premium/BC databases.
(2) In general, there were more creates and drops during the weekdays compared with the weekends for both Standard/GP or Premium/BC databases.
(3) Premium/BC databases had significantly fewer creates and drops than Standard/GP databases across all hours.

**Table 1: Features used for create and drop models**

| Features | Values |
|---|---|
| Temporal | Weekend vs. Weekday |
| Temporal | Hours |
| Database Edition | Standard/GP v.s Premium/BC |

*4.1.3 Model Formulations.* Based on the above findings and illustrated in Table 1, three features were used to formulate the model: weekday vs weekend, hour of the day, and service tier of the database. In total we built 96 ($2 \times 24 \times 2$) different Create DB models and another 96 different Drop DB models. For each hour, either on weekday or weekend and for each database edition, we first assumed that the number of creates or drops within the training period belonged to a well-defined probability distribution. Then, we fitted the hourly training dataset via various probability distributions including normal, uniform, Poisson and negative binomial [38, 59]. Normal was preferred over the other distributions for mainly two reasons: 1) its simulation results were most representative of our training dataset for database creates and drops; 2) it was found to be the best fitted distribution for the Steady State Growth of disk usage as well (see Section 4.2.2). The non-parametric *Kolmogorov-Smirnov* (K-S) test [1], a popular statistical test that helps determine whether a dataset follows a normal distribution, was then performed across all the hourly training datasets for both the Standard/GP and Premium/BC databases. For each of the box plots in Figure 7, there were 24 data points which corresponded to p-values for each of the 24 hours. All the p-values (except a few of them for the Premium/BC weekday drop) were greater than 0.05, hence we could not reject the null hypothesis that the training dataset followed a normal distribution. Based on the K-S test outcomes, we decided to model each hour as a separate normal distribution for the Create DB models and Drop DB models.

*4.1.4 Simulation Results.* To validate the trained models, they were executed in a simulated environment 100 times and the results were illustrated in Figure 8. Our "hourly normal" model was able to imitate the create and drop production trace closely. The modeled creates and drops (in gray) were very close to the production curves (in red). The mean (in black) of the 100 modeled curves nearly overlapped with the production curve.

## 4.2 Disk Usage Model

*4.2.1 Overview.* The disk usage model aims to imitate the disk usage growth patterns of databases within the cluster over a fixed time interval. We modeled this by discretizing the disk usage for each database into 20 minute time periods and computing the *Delta Disk Usage*. The *Delta Disk Usage* is the disk space usage difference between adjacent 20 minute time periods. After computing the *Delta Disk Usage*, we observed that around 99.8% of the time across databases and time stamps the disk usage showed a steady-state growth pattern (see Section 4.2.2). For the remaining 0.2%, it was dominated by initial creation growth (see Section 4.2.3) and predictable rapid growth patterns (see Section 4.2.4).

All databases in our implementation of Toto use the Steady-State Growth pattern (with different parameters for Standard/GP and Premium/GP databases) to determine what load should be reported

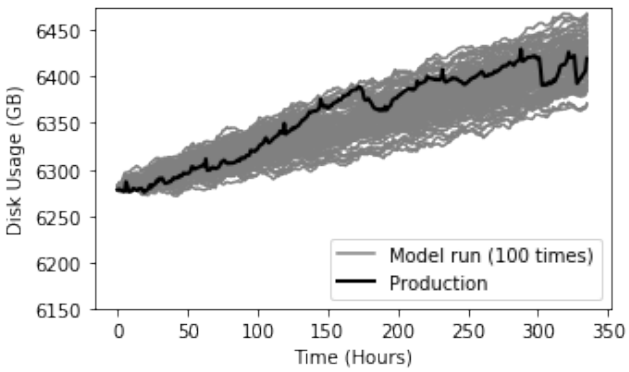



**Figure 9: Results of Steady State Disk Usage Pattern.**

to the PLB. A subset of databases use the Initial Creation Growth pattern immediately upon creation. This pattern attempts to capture a common customer behavior of restoring a database from an existing mdf file (the primary data file) [6]. Similarly, only a subset of databases use the Predictable Rapid Growth pattern. This growth pattern captures specific instances of temporal customer behavior (for example, a customer might do a batch import every day at midnight). The Initial Creation and Predictable Rapid Growth patterns model large increases in disk usage that the Steady State pattern cannot capture. For all of the growth patterns above, Standard/GP and Premium/BC databases were modeled separately.

*4.2.2 Steady State Growth.* The Steady State Growth Pattern was generated by training over the *Delta Disk Usage* values. We explored several statistical approaches including non-parametric kernel density estimations (KDE) [44] and a customized binning model in which the training set was divided into bins, each with a probability. However, similar to the Create DB/Drop DB model, we decided to imitate the *Delta Disk Usage* by using a "hourly normal" model for the following reasons:

(1) Unlike customized binning, it could capture temporal disk usage patterns.
(2) It had comparable or smaller dynamic time warping (DTW) [7, 52, 53] and root mean squared errors (RMSE) than KDE and the customized binning model.
(3) It was computationally efficient. Sampling from the model was just sampling from a normal distribution.
(4) Unlike KDE, the hourly normal model was easy to implement and did not rely on an external C++ library.

The hourly normal model was trained using the 99.8% of the data that corresponded to the steady-state growth pattern. The modeling results are illustrated in Figure 9. The time series of disk usage production data (in black) exhibited temporal patterns and our "hourly normal" models were able to capture those patterns. We primarily aimed to have the resulting cumulative disk usage from our models to be as close to production as possible over the two week training period while also achieving the modeled disk growth (in gray) to be similar to the production curve.

*4.2.3 Initial Creation Growth.* A common customer workload pattern is to have rapid growth upon the creation of the database, either to restore from an existing mdf file or to bulk load new data into the database. We attempted to capture these customer behaviors by generating a separate model for rapid growth during the first 30 minutes of a database's existence. This model assumes that the high growth period will last for 30 minutes, modeling the length of the initial growth is left as future work. Using the *Delta Disk Usage* values computed previously, we labeled databases in our training set as "High Initial Growth" if they had growth more than 12GB within the first five minutes of the database's lifetime. Using this subset of the training data that is "High Initial Growth", we computed a probability distribution describing how much the database should grow in the first 30 minutes and a probability that a database should exhibit "High Initial Growth" behavior. Normal and uniform distributions were used to fit the dataset and uniform was chosen because it performed better during model fitting. The probability distribution was then created by partitioning the "High Initial Growth" *Delta Disk Usage* values into five uniform bins, each with equal probability of being selected. When creating a database, Toto uses this probability and probability distribution to determine if the new database should have high initial growth. The simulation result from our model aligned well with the production data.

*4.2.4 Predictable Rapid Growth Patterns.* Upon empirical examination of the *Delta Disk Usage* values, we noticed that a subset of databases have a clear temporal pattern to their disk usage. This pattern can be characterized as a large rapid spike in their disk usage, followed by a rapid decrease in disk usage (perhaps this database is being used as part of an ETL pipeline, where new data is loaded in and old data is aged out). To capture this customer behavior, we manually partitioned the training set by selecting databases that exhibited this pattern by observing their *Delta Disk Usage* values over time. We then computed a probability that a database should exhibit Predictable Rapid Growth using the counts in each partition. The Predictable Rapid Growth Pattern was implemented as a state machine inside of Toto, where each state is described by the growth magnitude and the amount of time until moving to the next state. Below are the states and the order in which they are executed:

- Steady State Growth
- Rapid Disk Increase
- Steady State Growth Between Spikes
- Rapid Disk Decrease

Similar to the Initial Creation Growth, the growth magnitude parameter for the rapid growth states was computed by binning the *Delta Disk Usage* values into five buckets of equal probability and using a uniform distribution within each bucket. The time parameter for each state was computed by taking the average time in each state for every database in our Rapid Growth training set.

## 5 EXPERIMENTAL RESULTS AND DISCUSSION

To evaluate the efficiency and QoS of Azure SQL DB, we conducted a series of experiments using our implementation of Toto to orchestrate the disk usage load and the population of databases in the cluster (i.e., CPU reservations). At four different density levels, we collected telemetry on the cores reserved for databases, the disk utilization, and the failovers that occurred. In this context, increased density (e.g., 110%) refers to reserving more cores for databases than

the predefined *logical* capacity of the node. (Recall that in Azure SQL DB the logical capacities are set conservatively with respect to the physical capabilities of the node). As the density in a cluster increases, it is possible that more failovers will occur to relieve the resource pressure. The following experiments were conducted to find the optimal density while still ensuring a positive customer experience. Below is a description of the modeled adjusted revenue calculation used to evaluate efficiency, the experimental setup, the results, and a discussion on the experiments.

## 5.1 Modeled Adjusted Revenue Calculation

In production, Azure SQL DB customers are charged based on the SLOs and the lifetime of their databases. For our experiments, we modeled adjusted revenue based on the following two factors.

(1) *Revenue.* The modeled revenue of each database (the price the customer paid) was determined by its SLO [9]. For a single database, the compute revenue was calculated by multiplying the price of database instance by the lifetime of the database. The storage revenue was calculated by multiplying the size of the data by the price of storage and the lifetime of the database. The aggregate revenue is the sum of the compute revenue and storage revenue.
(2) *Penalty cost.* The service-level agreement (SLA) for Azure SQL DB [55] is 99.99%. To compute modeled adjusted revenue, we assumed that if a database was down 0.01% or more of its lifetime, service credits based on the SLA would be paid back to the customer and subtracted from the revenue.

## 5.2 Experimental Setup

After implementing Toto, we benchmarked the efficiency of a single Azure SQL DB cluster. To do this, we used a smaller 14 node, gen5, stage cluster. Stage clusters in Azure SQL DB are used for testing and validating new features before they are deployed into production – they are identical to production clusters in all other respects. Each experiment was executed in real time and observed by collecting telemetry from the cluster. The goal of the experiments was to measure the efficiency of the SQL DB service at different density levels by observing how many databases can be packed into the stage cluster, the corresponding modeled adjusted revenue generated from these databases, and the customer experience approximated by the number of failovers that occurred. We ran four experiments back-to-back, with each experiment running for 6 days. Each experiment had a different density level: 100%, 110%, 120%, and 140%. *Here, even 100% may be beyond the comfortable limit in production.*

At the beginning of each experiment, we bootstrapped the cluster to contain an initial population of databases. Using the production telemetry, we generated an initial population that had a representative mix of Premium/BC databases vs Standard/GP databases, a representative mix of SLOs within each service tier, and a representative mix of initial disk usage loads. The number of reserved cores in the cluster is determined by the modeled SLO sizes of the initial population. Table 2 contains a breakdown of the initial population.

Upon creation of each database in the initial population, the disk usage was initialized and the growth was fixed to 0. During bootstrap, the disk usage growth was fixed to 0 to prevent the databases from growing before the experiment had begun. This also allows the PLB to properly place and balance the databases throughout the cluster before the experiment. Once this was finished, each experiment officially began by modifying the model XML to specify the growth patterns for each database and instructing the Population Manager to begin creating and dropping databases.

**Table 2: Initial Population**

| Premium/BC Databases | Standard/GP Databases | Total |
|---|---|---|
| 33 | 187 | 220 |

Table 3 presents the resource usage breakdown of our initial population for each experiment. While the number of databases and disk utilization levels are held constant, the free remaining logical cores increases as the density level increases, as more logical cores are available at higher density levels.

**Table 3: Experiment Parameters**

| Density Level % | Free Remaining Logical Cores | Disk Usage % |
|---|---|---|
| 100 | 65 | 77 |
| 110 | 158 | 77 |
| 120 | 224 | 77 |
| 140 | 326 | 77 |

The determinism of each experiment was fixed as much as possible by explicitly setting the seeds of all the random objects used within the code. The Population Manager used a single seed which fixed the order and the SLO of the databases that were created. The random seeds used in Toto's orchestrator are specified through the XML and are constructed when the model objects are built. Every node in the cluster contains an RgManager modified with Toto, so a unique seed was provided to every node.

However, because we conducted these experiments on a real Azure SQL DB stage tenant ring, there is some inherent randomness in the cluster that is difficult to control. For example, this cluster was still subject to internal code upgrades to components other than Toto and intermittent failures that also happen in production. Additionally, the PLB in Service Fabric uses the Simulated Annealing algorithm to decide where to place replicas [30]. Simulated Annealing uses randomness to prevent getting stuck in locally optimal solutions when searching for placements. Similar to how production is configured, we were not able to use the same PLB random seed for each experiment. So while the initial population of databases is the same in each experiment, they are not guaranteed to be placed on the same node.

## 5.3 Experimental Results

*5.3.1 Creation Redirects.* Each experiment officially began after the Population Manager was instructed to begin creating and dropping databases. Because each experiment began with a different number of remaining free cores, the time at which the cluster can no longer accommodate new creation requests occurs earlier in the experiments with lower density levels. A creation redirect will

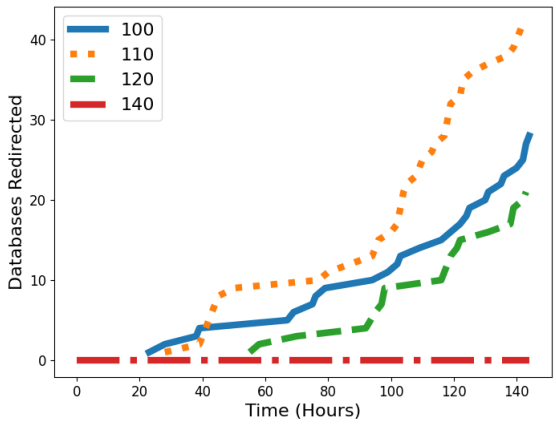



**Figure 10: Creation attempts redirected due to the cluster being exhausted in a resource.**

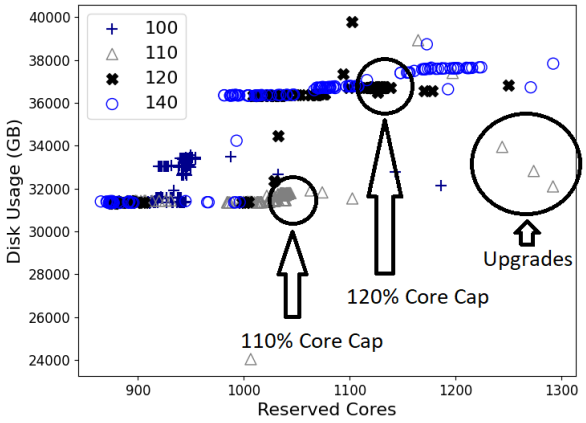



**Figure 11: Reserved Cores vs Disk Usage on the cluster over the six day experimental time period for four different density levels (each data point representing an hour).**

occur when the cluster does not have enough cores to satisfy the creation request. Instead of being placed in this tenant ring, the database will be redirected to another tenant ring that has enough capacity. Figure 10 shows that a creation redirect request first occurs at hour 23 in the 100% density experiment, which indicates that the cluster is running low on cores that can be reserved for new databases. At 110% density, the first creation redirect doesn't occur until hour 28, and at 120% density it doesn't occur until hour 55. The 140% experiment did not experience any creation redirects throughout the duration of the experiment. Interestingly, the 110% experiment crossed the 100% experiment in number of creation redirects around hour 40 and continued to have more creation redirects than the 100% experiment for the duration of the 6 days. This is because the 110% experiment was able to create a large 24 core Premium/BC database (replicated x4, 96 cores total) that the 100% experiment redirected to another tenant ring. This caused the remaining free cores of the cluster with 110% density to actually

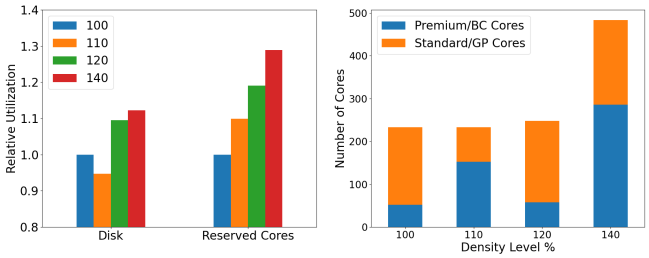



**Figure 12: (a) Relative Disk and Reserved Core Utilization and (b) Total failed-over CPU cores over the six days.**

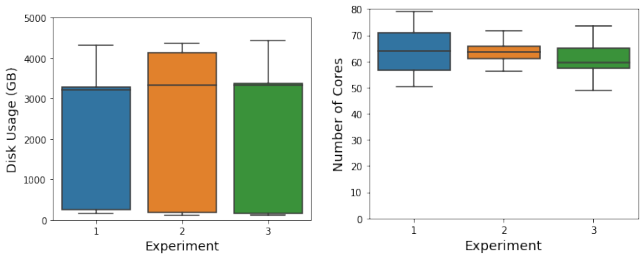
be smaller than 100%. This had a cascading effect on the number creation redirects, because the remaining free cores was smaller, more small core databases ended up being redirected in 110% than in 100%. For the duration of the 6 days, the 110% experiment had less remaining free cores than the 100% experiment, which is why 110% ended up with more creation redirects.

*5.3.2 Reserved Cores and Disk Usage.* As more databases are injected into the cluster, the number of reserved cores and disk usage begins to rise. Figure 11 shows the relationship between the reserved cores and the disk usage in the cluster for each density level. As the density level is increased, more cores are able to be reserved for databases (the outliers at each density level are when a cluster maintenance upgrade was occurring). The 140% experiment is able to reserve the most amount of cores, whereas the lower density levels eventually hit a cap and cannot reserve any more cores.

Figure 12 (a) shows the disk utilization and reserved core utilization relative to 100% density at the end of each experiment. As expected, as the density level increases, the reserved core utilization also increases. The 140% experiment is able to accommodate almost 30% more cores than the 100% density experiment.

As an affect of creating more databases, the disk usage and utilization in the cluster also increases. Figure 11 shows that there is a clear gap between the disk usage of the 120% and 140% experiments from the 110% and 100% experiments. This is because the 110% and 100% experiments redirected a 6 core Business Critical database that had high disk usage. This database had high initial growth (see Section 4.2.3), it grew about 1.3TB within the first 30 minutes of being created. This growth explains the gap in disk usage between the 120%, 140% experiments and the 110%, 100% experiments. This highlights the impact that a single Premium/BC database can have on the overall cluster state. A few Premium/GP databases contribute a disproportional amount of disk usage in the cluster and also reserve more cores than the Standard/GP databases. When the PLB needs to move a database to another node, it is important to minimize failovers of Premium/BC databases, as moving these databases is much more costly due to the higher disk usage.

*5.3.3 Failed-Over Cores.* Increasing the reserved core utilization and disk usage is desirable from a COGS perspective, but it can potentially have a negative effect on the quality of service of the system. With increased disk usage, depending on how the replicas are placed throughout the cluster, there is an increased probability that the disk capacity of a node will be breached. Figure 12(b)




(a) (b)

**Figure 13: Dispersion of mean (in 10min) node-level resource usage of: Disk Usage (GB) (a), Number of Cores (b) for three repeated, 18 hour experiments.**

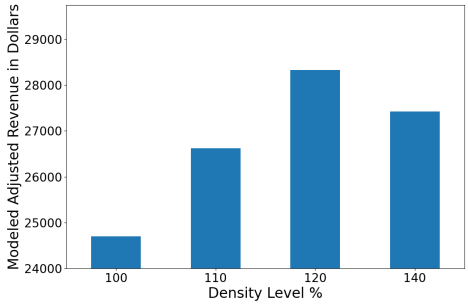



**Figure 14: Total modeled adjusted revenue generated over the six days.**

shows the number of cores that were failed-over during the course of the 6 days. Notably, 140% had the highest number of failed-over cores and failed-over more Premium/BC cores than the total of the other experiments. The 120% experiment had the lowest number of failed-over cores, but only slightly lower than 100% and 110%. When the disk usage becomes high enough, the PLB will need to fix the disk capacity violation by moving a Premium/BC database. Poor placement decisions can potentially disproportionately punish the number of failed-over cores later on by needing to move a Premium/BC database to fix a disk capacity violation. From a customer experience perspective, this is not desirable, as Premium/BC databases are more expensive (and generate more revenue than Standard/GP databases).

*5.3.4 Non-determinism of placements.* We wish to also present results quantifying the impact of the non-deterministic decision-making of the PLB within SF. To make the experiment of repeated runs manageable, and to avoid the impact of must-have, critical system changes deployed to our cluster, we evaluated the variance of different metrics observed within three identical 18 hour experiments. Figure 13(a) and (b) show the dispersion of node-level readings of disk usage (in GB) and the number of reserved CPU cores across the three runs, respectively. To appropriately test for significance, we used the Wilcoxon signed-rank test and quantified the significance of the differences between all pair-wise experiments for both metrics (e.g., six null hypothesis of "same distribution") and found that all but one (exp 2 versus 3 - reserved cores) of the significance tests ($\alpha = 0.05$) were insignificantly different. Unsurprisingly, following the insignificant differences in node-level distributions of these metrics due to non-deterministic variables, *our database failover counts for experiments 1-3 were one, zero, and one respectively.* With this, we have bolstered confidence in the results of our primary QoS KPI of database failovers in Figure 12(b).

*5.3.5 Modeled Adjusted Revenue.* The negative QoS (of failovers) is reflected in Figure 14. Figure 14 plots the modeled adjusted revenue generated from each 6 day long experiment, where the modeled adjusted revenue is computed according to the calculation described in Section 5.1. The modeled adjusted revenue for every experiment increases until 140%, where there is a noticeable decrease. The penalty applied to the 140% experiment is more than 60x larger than the other experiments. It is clear that increasing the density level in the cluster has a direct impact on the modeled adjusted revenue generated because the cluster can hold more databases, but at higher density levels the impact of the number of failovers becomes harmful to the service.

## 5.4 Discussion

Through our experimental evaluation of different density levels, it is evident that there is a trade-off between the density of the cluster and the rate of failovers that occur within the cluster. There are many components of the service that can influence the rate of failovers in a cluster. Most notably are: the tenant ring admission controller (in the control plane), the control exerted by the RgManager, and the PLB within Service Fabric. Here we focus on the evaluation post-admission - the RgManager and the PLB - but neither of these can ensure a good QoS for the customer if they are presented with a poor density scenario due to a "bad database population". We described in our experiment that even the admission of a single database exhibiting an innocuous behavior, can dramatically alter the rate of failovers for the population of a highly dense cluster. Ultimately, the different layers of the service need to have some sort of communication and coordination in order to achieve high efficiency and QoS.

Irrespective of the admission policy into the cluster, Toto can be used to find an optimal density level, given an initial population of databases at a specific resource utilization. In the above experimental evaluation, the disk utilization began at 77% of the logical capacity, but not all production clusters have such high disk utilization. The optimal density for a cluster is driven by the workload behavior of each database and overall resource utilization of the cluster. Clusters with lower disk utilization would be primary candidates for increased density (assuming they are not constrained by other resources).

There are also efficiency opportunities at the individual database level. Mechanisms for resource governance (and performance control) exposed by a database engine are available to the RgManager to facilitate efficient resource usage and minimize failovers. The details are out of scope for this paper, but smart performance/efficiency trade-off mechanisms and policies can be employed on each SQL database to further increase the cost-effectiveness of the system. Toto can also be used to evaluate the impact of these features by overriding the resource metrics used to inform these decisions.

Additionally, there are other ways to measure a cloud service's efficiency, other than from the service provider's perspective. There

are different notions of efficiency, such as how quickly an individual database can scale up to full resource utilization or the amount of time it takes to provision a new database. Exploring the intersections of different efficiency notions could help highlight the inherent tradeoffs that exist when providing a cloud service.

### 5.5 Future Work

Toto is currently being used inside Azure SQL DB to evaluate the impact on cluster wide feature improvements and configuration changes. We plan to improve Toto in various ways going forward to capture a wider variety of cluster states.

On modeling, we plan on adding models for memory and CPU usage (including distinct models for HA primary and secondary replicas). We also plan on more accurately mimicking the population of databases in our production clusters. For our experiments the population of databases was restricted to SQL DB singletons, but other offerings such as Elastic Pools [5] (which allow for multi-tenancy inside a single SQL DB instance) will add to environment accuracy. We will also be exploring how to use Toto to measure RgManager's effectiveness at mitigating potential performance issues. The models for the management of the database population can also be improved. Currently the Create/Drop DB models only aim to imitate the aggregate level of databases in the cluster, but future iterations will model an individual database's lifetime.

While Toto is currently built into RgManager of Azure SQL DB, the framework can be more broadly applied than just to a cloud database offering. The presented implementation modified a proprietary component the SQL DB stack to present fabricated resource loads to Service Fabric (SF), but alternative implementations can be implemented to be compatible with other orchestration frameworks. Any cloud service that needs to manage the resources allocated to applications using an orchestration system like SF or Kubernetes can benefit from implementing Toto by extending the resource management APIs (like those of Kubernetes [4]) and employing accurate and encompassing models to reflect their own production system. We will explore implementing Toto directly into the orchestration framework so it can be used by other cloud workloads.

## 6 RELATED WORK

There has been substantial work in the research community to examine the cost effectiveness of cloud database systems, benchmarking cloud services and database engines, and considering new service models that focus on customer-oriented cost-optimization. Select cloud-born, high efficiency database services include: Taurus [21], Aurora [57], and Socrates [8]. Starling [46] and Lambada [39] discuss low latency/cost using cloud functions, and customer oriented, cost effective performance. Other customer-oriented, cloud cost management, performance trade-off analysis include [16, 61]. The temporal patterns of customer actions and unplanned events is also critical to managing efficiency [36, 47, 48].

Additional work on Azure-related efficiency and reliability from the service back-end perspective include cost management [28], performance isolation [35, 41], demand forecasting [10, 17, 34], telemetry [33]. Considering the scale of the data issues in public cloud environments is also a major problem [11, 45]. In production, when things go wrong, measuring KPIs, and performing root cause analysis is crucial [20].

Recent work that considers cloud benchmarking for reproducibility and repeatability is similar to our work, but they consider running SQL queries in their benchmarking, and are also highly customer-oriented, rather than provider-side focus [22, 25].

To imitate resource usage patterns of databases in production, ML and other statistical learning approaches [14, 15, 18, 19, 24, 29, 31, 37, 40, 43, 49, 51, 58, 60] have been widely used. These studies suggest that database service providers can use utilization telemetry to effectively allocate resources or help the database learn how to perform faster. However, as previously discussed in Section 4, ML approaches usually require dependencies on external libraries and would have been difficult to be integrated into a critical production component such as RgManager. Furthermore, they may not be scalable for a large live service such as Azure SQL DB. Finally, non-ML approaches can achieve comparable or even better results than ML approaches for certain scenarios [27, 42, 50, 58].

## 7 CONCLUSION

A tremendously complex cloud service like Azure SQL DB faces numerous challenges when trying to innovate on efficiency. Microsoft cannot always determine what the impact of a system change may be prior to being deployed out into production, or even after the change has been deployed due to the inability to do any reliable or repeatable "A/B" testing. Through experience, we have found that attempts to reason about the outcomes of a system change from key performance indicators are largely unreliable or inconclusive. We present **Toto**, a framework to benchmark the efficiency of the service, as our solution. Toto effectively hijacks the communication of customer applications (i.e., SQL databases via RgManager) to a cluster orchestrator such as Kubernetes, or in our case, Service Fabric (SF). Toto leverages models of individual database behavior, sub-population behavior, and behaviors of higher-level subsystems in the service stack, such as request routing and sends the relevant inputs to the cluster orchestrator. *Toto is not limited in its relevance to a cloud database service, but applies to any cloud service that leverages cluster orchestration using a system like Kubernetes or SF.* With Toto, we are able to reliably and repeatably create benchmarking scenarios in which we can change the service variables to measure the impact. After expanding on an existing mechanism within the RgManager stack of SQL DB, we implemented Toto into RgManager and it was subsequently deployed in staging clusters worldwide.

We presented a study on the potential benefits and effects of increasing database density in SQL DB clusters. Using statistical modeling to imitate the disk usage of a database, we then tuned the target density level within a single Azure SQL DB stage cluster. The study helped quantify the degree that increasing the density of the cluster has on negative customer experience. Internally, Toto is being used for: configuration change evaluations, justification evaluations of efficiency feature proposals, and reproducing production problem scenarios. We plan to further enhance Toto with more detailed models, as well as introducing new models for resources such as memory and models of SQL engine's performance-efficiency mechanisms.

## REFERENCES


[1] K-S test Python Package. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kstest.html.

[2] Managing resource consumption and load in service fabric with metrics. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-metrics, 2017.

[3] Azure sql database and azure sql managed instance service tiers. https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tiers-general-purpose-business-critical, 2020.

[4] Custom resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/, 2020.

[5] Elastic pools help you manage and scale multiple databases in azure sql database. https://docs.microsoft.com/en-us/azure/azure-sql/database/elastic-pool-overview, 2020.

[6] Recover using automated database backups - azure sql database & sql managed instance. https://docs.microsoft.com/en-us/azure/azure-sql/database/recovery-using-backups, 2020.

[7] A. Abanda, U. Mori, and J. A. Lozano. A review on distance based time series classification. *Data Mining and Knowledge Discovery*, 33(2):378–412, 2019.

[8] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1743–1756, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Microsoft corporation. http://azure.microsoft.com/en-us/pricing/details/sql-database, 2016.

[10] J.-H. Böse, V. Flunkert, J. Gasthaus, T. Januschowski, D. Lange, D. Salinas, S. Schelter, M. Seeger, and Y. Wang. Probabilistic demand forecasting at scale. In *VLDB*, pages 1694–1705, 2017.

[11] E. Boutin, P. Brett, X. Chen, J. Ekanayake, T. Guan, A. Korsun, Z. Yin, N. Zhang, and J. Zhou. Jetscope: Reliable and interactive analytics at cloud scale. In *VLDB*, pages 1680–1691, 2015.

[12] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[13] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):70–93, 2016.

[14] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya. Workload prediction using arima model and its impact on cloud applicationsâĂŹ QoS. *IEEE Transactions on Cloud Computing*, 3(4):449–458, 2014.

[15] E. Caron, F. Desprez, and A. Muresan. Forecasting for grid and cloud computing on-demand resources based on pattern matching. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 456–463. IEEE, 2010.

[16] S. Chawla, S. Deep, P. Koutrisw, and Y. Teng. Revenue maximization for query pricing. *Proc. VLDB Endow.*, 13(1):1–14, Sept. 2019.

[17] 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 1923–1934, 2016.

[18] S. Das, V. Narasayya, F. Li, and M. Syamala. CPU Sharing Techniques for Performance Isolation in Multi-tenant Relational Database-as-a-Service. In *PVLDB*, 2013.

[19] V. Debusschere and S. Bacha. Hourly server workload forecasting up to 168 hours ahead using seasonal arima model. In *2012 IEEE international conference on industrial technology*, pages 1127–1131. IEEE, 2012.

[20] M. B. Demarne, J. Gramling, T. Verona, and M. Cilimdzic. Reliability analytics for cloud based distributed databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1479âĂŞ1492, New York, NY, USA, 2020. Association for Computing Machinery.

[21] A. Depoutovitch, C. Chen, J. Chen, P. Larson, S. Lin, J. Ng, W. Cui, Q. Liu, W. Huang, Y. Xiao, and Y. He. Taurus database: How to be fast, available, and frugal in the cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1463âĂŞ1478, New York, NY, USA, 2020. Association for Computing Machinery.

[22] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, Dec. 2013.

[23] N. R. Draper and H. Smith. *Applied regression analysis*, volume 326. John Wiley & Sons, 1998.

[24] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan. Characterizing Tenant Behavior for Placement and Crisis Mitigation in Multitenant DBMSs. In *SIGMOD*, pages 517–528, 2013.

[25] P. K. Erdelt. A framework for supporting repetition and evaluation in the process of cloud-based dbms performance benchmarking. In *TPCTC*, 2020.

[26] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[27] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16. Ieee, 2010.

[28] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM*, 39(1):68–73, 2008.

[29] S. Islam, J. Keung, K. Lee, and A. Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2012.

[30] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi, M. Mohsin, R. Kong, A. Ahuja, O. Platon, A. Wun, M. Snider, C. Daniel, D. Mastrian, Y. Li, A. Rao, V. Kidambi, R. Wang, A. Ram, S. Shivaprakash, R. Nair, A. Warwick, B. S. Narasimman, M. Lin, J. Chen, A. B. Mhatre, P. Subbarayalu, M. Coskun, and I. Gupta. Service fabric: A distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[31] A. Khan, X. Yan, S. Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium*, pages 1287–1294. IEEE, 2012.

[32] D. Lambert. Zero-inflated poisson regression, with an application to defects in manufacturing. *Technometrics*, 34(1):1–14, 1992.

[33] W. Lang, F. Bertsch, D. J. DeWitt, and N. Ellis. Microsoft Azure SQL Database Telemetry. SoCC, pages 189–194, 2015.

[34] 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. *PVLDB*, 2016.

[35] W. Lang, S. Shankar, J. Patel, and A. Kalhan. Towards Multi-Tenant Performance SLOs. In *ICDE*, pages 702–713, 2012.

[36] Y. Li, E. L. Miller, and D. D. E. Long. Understanding data survivability in archival storage systems. In *SYSTOR*, 2012.

[37] R. Marcus and O. Papaemmanouil. Wisedb: A learning-based workload management advisor for cloud databases. In *VLDB*, pages 780–791, 2016.

[38] P. McCullagh. *Generalized linear models*. Routledge, 2018.

[39] I. Müller, R. Marroquín, and G. Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 115–130, New York, NY, USA, 2020. Association for Computing Machinery.

[40] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri. Sharing Buffer Pool Memory in Multi-tenant Relational Database-as-a-service. *PVLDB*, pages 726–737, 2015.

[41] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *CIDR*, 2013.

[42] S. Pacheco-Sanchez, G. Casale, B. Scotney, S. McClean, G. Parr, and S. Dawson. Markovian workload characterization for QoS prediction in the cloud. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 147–154. IEEE, 2011.

[43] Y. Park, A. S. Tajik, M. Cafarella, and B. Mozafari. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 587–602, 2017.

[44] E. Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962.

[45] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. In *VLDB*, pages 1816–1827, 2015.

[46] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 131âĂŞ141, New York, NY, USA, 2020. Association for Computing Machinery.

[47] J. Picado, W. Lang, and E. C. Thayer. Survivability of cloud databases - factors and prediction. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 811âĂŞ823, New York, NY, USA, 2018. Association for Computing Machinery.

[48] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST*, 2007.

[49] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. Data management challenges in production machine learning. In *SIGMOD*, pages 1723–1726, 2017.

[50] O. Poppe, T. Amuneke, D. Banda, A. De, A. Green, M. Knoertzer, E. Nosakhare, K. Rajendran, D. Shankargouda, M. Wang, et al. Seagull: An infrastructure for load prediction and optimized resource allocation. *arXiv preprint arXiv:2009.12922*, 2020.

[51] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507. IEEE, 2011.

[52] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing*, 26(1):43–49, 1978.

[53] S. Salvador and P. Chan. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, 11(5):561–580, 2007.

[54] R. H. Shumway and D. S. Stoffer. *Time series analysis and its applications: with R examples*. Springer, 2017.

[55] SLA for Azure SQL Database. https://azure.microsoft.com/en-us/support/legal/sla/sql-database/v1_4/, 2019.
[56] R. Taft, W. Lang, J. Duggan, A. J. Elmore, M. Stonebraker, and D. J. DeWitt. STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments. SoCC, 2016.
[57] A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvilli, and X. Bao. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 789âĂŞ796, New York, NY, USA, 2018. Association for Computing Machinery.
[58] L. Viswanathan, B. Chandra, W. Lang, K. Ramachandra, J. Patel, A. Kalhan, D. J. Dewitt, and A. Halverson. Predictive provisioning: Efficiently anticipating usage in azure sql database. In *IEEE ICDE*, pages 1111–1116, 2017.
[59] C. Walck. Hand-book on statistical distributions for experimentalists. *University of Stockholm*, 10, 2007.
[60] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432, 2019.
[61] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 415âĂŞ432, New York, NY, USA, 2019. Association for Computing Machinery.