Tenant Placement in Over-subscribed Database-as-a-Service Clusters

Arnd Christian König
Microsoft Corporation
chrisko@microsoft.com

Yi Shan
Microsoft Corporation
shayi@microsoft.com

Tobias Ziegler
TU Darmstadt
tobias.ziegler@cs.tu-darmstadt.de

Aarati Kakaraparthy
University of Wisconsin-Madison
aaratik@cs.wisc.edu

Willis Lang
Microsoft Corporation
wilang@microsoft.com

Justin Moeller
Microsoft Corporation
jumoell@microsoft.com

Ajay Kalhan
Microsoft Corporation
ajayk@microsoft.com

Vivek Narasayya
Microsoft Corporation
viveknar@microsoft.com

ABSTRACT

Relational cloud Database-as-a-Service offerings run on multi-tenant infrastructure consisting of clusters of nodes, with each node hosting multiple tenant databases. Such clusters may be over-subscribed to increase resource utilization and improve operational efficiency. When resources are over-subscribed, it is possible that a node has insufficient resources to satisfy the resource demands of all databases on it, making it necessary to move databases to other nodes. Such moves can significantly impact database performance and availability. Therefore, it is important to reduce the likelihood of such resource shortages through judicious placement of databases in the cluster. We propose a novel tenant placement approach that leverages historical traces of tenant resource demands to estimate the probability of resource shortages and leverages these estimates in placement. We have prototyped our techniques in the *Service Fabric* cluster manager. Experiments using production resource traces from Azure SQL DB and an evaluation on a real cluster deployment show significant improvements over the state-of-the-art.

PVLDB Reference Format:
Arnd Christian König, Yi Shan, Tobias Ziegler, Aarati Kakaraparthy, Willis Lang, Justin Moeller, Ajay Kalhan, and Vivek Narasayya. Tenant Placement in Over-subscribed Database-as-a-Service Clusters. PVLDB, 15(11): 2559 - 2571, 2022.
doi:10.14778/3551793.3551814

1 INTRODUCTION

In Database-as-a-Service (DBaaS) settings, the service provider is responsible for maintaining the database software, resource management and high availability of the service [27]. The service provider hosts the *database tenants* on sets of nodes called *clusters*. On a node, one or more database processes, each assigned to a database tenant, execute within one or more virtual machines; the resources of the node are shared between these processes.

To ensure very high availability, multiple replicas of the same database tenant can be hosted in the cluster; these typically consist of one *primary* replica, which serves both read and write operations, as well as *secondary* replicas, which provide scale-out for read operations and high availability (see e.g., [4] for details). The orchestration of the various services underlying a DBaaS system is managed by a *cluster manager* such as *Kubernetes* [23] or *Service Fabric* [20]. In particular, the cluster manager is responsible for the placement of replicas within the cluster.

Cluster Resources can be *over-subscribed*, i.e., the sum of resources promised to databases on a node may exceed the node's capacity. There are multiple reasons for over-subscription: First, if tenants use only a fraction of the promised resources, reserving the maximum resources promised means a large fraction of cluster capacity is idle. Careful over-subscription can lead to better cluster utilization while still providing tenants with the resources needed. A second reason for over-subscription is to handle capacity shortages (which can be due to node failures, hardware unavailability, or when nodes reboot during a cluster upgrade). Here, temporary over-subscription protects availability, as tenants from unavailable nodes can still be placed in the cluster, even if the sum of resources promised exceeds the available node capacities.

When a cluster is over-subscribed, it is possible that the aggregate resource demand on a node exceeds the node's capacity. Therefore, if the aggregate resource demand on a node exceeds a certain threshold, we consider this to be a *resource violation*. If a resource violation persists, it becomes necessary to move tenant replicas to other nodes in the cluster to alleviate the violation. We refer to such a move as a *failover*. Note that failovers can also occur for other reasons, e.g., load balancing, node upgrades, etc.

Resource violations are a significant issue as the resulting failovers can significantly impact performance and tenant availability. Specifically, (a) the tenant failed over may become unavailable, (b) queries executing on the failed-over database may be canceled, (c) the state in the database caches may be lost during failovers, and (d) for databases using local storage, the on-disk state has to be copied to the new node. Thus, minimizing the number of violations is crucial. For this, the key decision becomes on which node to place each tenant replica, which is the problem we study in this paper.

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551814

Tenant placement has been widely studied, both in research as well as industrial practice. Unfortunately, the existing techniques are not well-suited to address the problem described above. Many existing techniques make placement decisions based on a static *snapshot* of the resource demands in a cluster. However, tenant resource demands can change rapidly. Such changes in demand can be challenging to anticipate, in particular for new tenants, because little is known about tenants *before* they are placed. As a result, existing placement techniques may result in excessive failovers.

Some approaches solve different optimization tasks, such as maximizing tenant density (leading to additional violations), or minimizing the effect of violations on tenant performance *without* actually resolving the violations, making them non-applicable for our scenario.

We propose a different approach to tenant placement, which uses the historical resource usage data that DBaaS providers collect. This data allows us to characterize the distributions of resource demands (over time) and tenant lifetimes. Based on these distributions, we estimate the *probability of a future resource violation* for a set of tenants sharing a node. This probability and its computation is the key contribution of this paper; we will show how to integrate it into tenant placement algorithms to achieve a significant reduction in resource violations compared to the state-of-the-art.

Properties of the approach: The probability of violation estimates are computed using *Monte Carlo* (MC) simulations, which driven by historical demand traces. The use of Monte Carlo techniques has a number of advantages: (1) They are able to account for dynamic changes in resource demand over time. (2) There is no requirement to have precise predictions of future resource usage. Instead, the simulation implicitly models the uncertainty in future resource demand and combines the uncertain estimates for different tenants in a principled way. (3) Because historical traces are replayed for all resources in lock-step, the approach automatically captures correlations in demand across different resources.

Evaluation: We compare our approach to state-of-the-art placement algorithms from research, including *Virtual Machine* placement [28], cluster scheduling [17] and theoretical work on *on-line vector (re-)packing* [31]. In addition, we compare to the placement logic of *Service Fabric* [20], and placement heuristics used in *Kubernetes* [23]. We integrated our technique into a prototype of *Service Fabric* and use a deployment on a real cluster as part of the evaluation. Overall, the proposed approach performs consistently better than the state-of-the-art, with even the *best-performing* alternatives resulting in at least 2.1x as many violations as our approach.

While the proposed approach is also applicable to other services with varying resource demands hosted in over-subscribed clusters, we limit the scope of this paper to relational database systems.

2 RELATED WORK

Industrial Cluster Managers: A common approach used in industrial practice is to base placement decisions on a *snapshot* of the resource demands of existing tenants, and an estimate of the resource demands of new tenants, and then use heuristics such as *WorstFit* [39], *BestFit* [39], etc. to make placement decisions. However, the demand snapshots fail to account for changes in resource demands over time, including tenant departures.

Other cluster managers have studied *different optimization tasks*: [32] models placement as a minimization problem, where the number of concurrently used servers is minimized subject to constraints on service-level objectives and node load. In contrast, in our scenario, cluster sizes are fixed and incidence of violations is the optimization criterion. [12] manages resources based on the impact shortages have on workloads; the allocation mechanism allocates the *minimum resources needed to satisfy a performance target*. This differs from our scenario, where violations must be resolved, regardless of their performance impact.

Online Packing Techniques: The task of placing tenants on nodes is highly related to the well-known *bin-packing* problem. While *offline* and *static* instances of bin-packing for a single resource admit efficient solutions [15], DBaaS tenant placement is more challenging due to (1) there being multiple resources, (2) tenants arriving and departing dynamically, and (3) tenant's resource requirements changing over time. Packing algorithms that account for dynamic arrivals/departures and changes in resource demand have been developed in the theory community. These use *online (vector) bin-packing* techniques [8, 9, 14, 28] to maintain tenant packings *competitive with optimal offline bin-packing* schemes, while providing worst-case bounds on failovers required for different *events* (such as a new placement). Because the approaches *prioritize the packing density*, the bounds on failovers are too loose to be practical for our setting. For example, [31] may require up to 10 failovers for a single change in demand.

Tenant Consolidation: These techniques (e.g., [10, 37]) initially observe the demand of each tenant for a minimum time period, use these observations to predict future usage and subsequently consolidate tenants on a smaller set of nodes. The challenge here is that the consolidation itself requires failing over new tenants at least once, making the approach impractical, as we seek to avoid failovers altogether for most tenants.

Cluster Scheduling: The tenant placement problem is also related to the task of *cluster scheduling* (e.g., [19, 38]). However, the techniques proposed in this context differ from our scenario in a number of respects: first, in many of these approaches, resources for a service are *reserved statically* and remain assigned independently of whether they are used (e.g., see [38], Section 2.4). Moreover, the key metrics optimized in scheduling (e.g., *job completion times*, *fairness*) are very different from minimizing violations.

Database Migration: How to migrate database replicas within a cluster to alleviate resource contention has been an active area of research as well [7, 11, 13, 26]. [26] studies *swap-removable* resource contention scenarios, which can be resolved by swapping primary and secondary replicas. This approach is applicable to CPU oversubscription, it fails for other resources (e.g., disk space violations, as swaps do not necessarily reduce the disk footprint); moreover, it is not applicable to non-replicated tenants.

[7] proposes *SWAT*, an end-to-end tenant migration framework. Here, the primary focus is on how to *minimize tenant downtime* during the migration, as well as minimizing the impact of the migration operations on workload throughput. However, how to place tenants to minimize disruption is not studied. Thus, [7] complements the approach of this paper. The same holds for [11, 13], which study live migration of database replicas within a cluster.

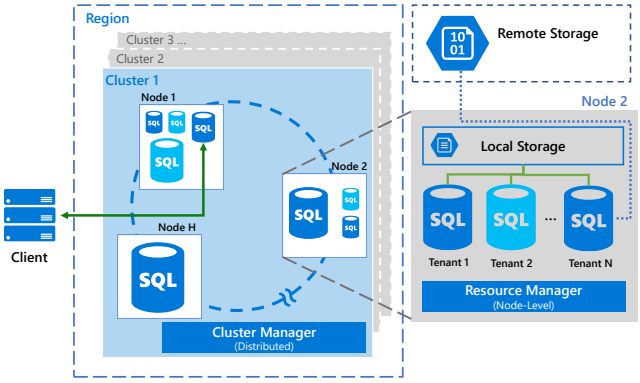

Figure 1: Database tenants are hosted in clusters of nodes

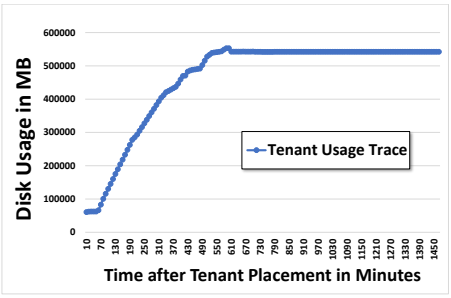

Figure 2: Real-life Example of Disk Demand in Azure SQL

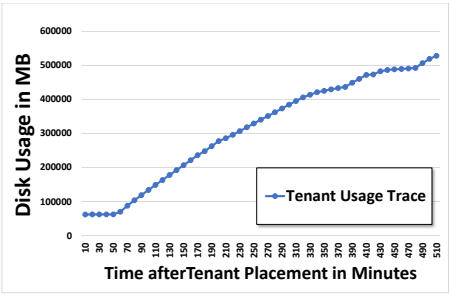

Figure 3: Real-life Example of Disk Demand in Azure SQL

3 BACKGROUND

We study database tenant placement in over-subscribed DBaaS clusters (Figure 1). Sets of clusters are organized into larger groups (e.g., different geographical regions). Each cluster is managed by a *cluster manager* (e.g., Service Fabric [20] or Kubernetes [23]), which handles tenant placement, state replication, reconfiguration of replicas on node failure, and resource load balancing.

A node-level *resource manager* manages the distribution of resources across tenant databases on the node, which share the node's resources. Thus, the key issue for resource management becomes *which databases are placed together* on a node, as this implicitly determines whether their aggregate demand can be satisfied. Each tenant is assigned a *tenant class*, which is a partitioning of tenants into groups of tenants with similar resource demands, based on information available *before* a tenant is placed. Different ways to assign these classes are possible; in this paper we assign the tenant classes based on 4 factors: (a) the maximum resources available to each tenant, (b) the location of the data itself (local SSD vs. remote storage), and (c) the billing model used (e.g., *Serverless*) and (d) the number of replicas. Any tenants that agree on all of these factors are assigned to the same tenant class.

Resource violations: A node is in resource violation if the aggregate demand for a resource exceeds a threshold (e.g., 95% of the resource capacity), in which case replicas need to be moved to other nodes in the cluster until the demand is below the threshold again. These moves are not instantaneous, which is why the violation thresholds are set lower than the node's capacity – triggering a violation at the lower threshold allows for sufficient time for the move to complete before the resource is physically exhausted.

3.1 Characteristics of Resource Demands

In this section, we describe the relevant characteristics of the resource demand curves seen in real DBaaS tenants. Our observations are based on tenant traces collected in Azure SQL Database [34].

The shape of resource demand curves: When a new tenant is placed in a DBaaS cluster, the underlying databases are typically initialized (e.g., from a backup). As a result, the disk demand for new tenants often grows rapidly, followed by a period of only small increases (see Figure 2 for an example). A smaller fraction of tenants continues to grow throughout their lifetime, as data is continuously added (see Figure 3). In both cases, disk demand very rarely sees substantial decreases, with occasional small decreases due to changes in temporary workspaces (e.g., a disk spill). Similar observations have been made in [25] (see Section 4.2).

We have observed similar patterns in the typical memory demand curves of newly arriving tenants, which grow rapidly in the beginning of their lifetime, as caches are populated with the working set of the application. Unlike disk demand, noticeable reductions in the memory footprint are more common, especially in *Serverless* tenants which may reclaim memory aggressively (see e.g., [35]).

To quantify these observations, we analyzed demand traces of 10K random tenants from 4 different geographical regions: For memory and disk demand, over 90% of tenants reach 95% of their maximum disk/memory demand within (depending on the region) 2%-5% of their lifetime. After they have reached 95% of their maximum disk demand, disk demand rarely drops back below 90% of the maximum – less than 1% of tenants show this pattern. Drops in memory demand are more common, but still atypical: once tenants have reached 95% of their max. memory demand, less than 7% of tenants ever drop below 80% of the maximum.

We will use these observations in Section 5.3 to propose a compressed representation of demand curves for these resources.

3.2 Notation

We consider a cluster of H nodes $Nodes = \{N_1, \ldots N_H\}$, each of which offers *resources* $\mathcal{R} = \{r_1, \ldots\ r_v\}$. For a resource r, every node has a *capacity* c_r.

Database tenants: we consider a set of W database tenants $Tenants = \{T_1, \ldots T_W\}$; each tenant has 1 or more replicas. The set of all replicas is denoted as $\mathcal{DB}$; each replica $db \in \mathcal{DB}$ has a *lifetime* denoted

by $lifetime(db)$ which is the time between replica creation and it being removed from the cluster, and a $creationtime(db)$, which corresponds to the time when db was initially placed in the cluster. We denote the resource demand for a replica db and a resource r at δ units of time after db's creation time by $usage^r_{db,\delta}$. We refer to the sequence of a tenant's resource demands for a resource r

$$\langle usage^r_{db,0}, \ldots, usage^r_{db,lifetime(db)-1} \rangle$$

as the tenant's *demand curve*. Each tenant replica resides on one of the nodes at any point of its lifetime; we denote the replicas placed on a node N_i at time t by $tenants(N_i, t)$. For ease of exposition, we use the terms *replica* and *database* interchangeably throughout the paper. We divide time into discrete intervals (e.g., 1 minute), and use the time "*now*" to specify the current time. Using this, we can specify the aggregate demand on a node N_i at time t as:

$$nodeload^r_{i,t} := \sum_{db \in tenants(N_i,t)} usage^r_{db,t-creationtime(db)}$$

In addition to the actual resource demand, each tenant replica has a *maximum* demand $MaxUsage^r_{db}$, which is an upper bound on how much of resource r database db can use. Using this max. usage, we capture the degree a resource is over-subscribed via the *oversubscription-ratio* of a node N_i for a resource r as

$$OR^r_i := \Big(\sum_{db \in tenants(N_i,t)} MaxUsage^r_{db} \Big) / c_r$$

and for the cluster OR^r as the average of the OR^r_i over all nodes.

Resource violations: we consider a node to be in violation for a resource r if the aggregate demand on the node is at least as large as a threshold $f_r \cdot c_r$, where $0 < f_r \leq 1$.

3.3 Problem Formulation

Problem definition: Given an (initially empty) cluster of H nodes and an input sequence $\mathcal{WL}$ of *operations*, with each operation being associated with a timestamp t and being one of (a) a new tenant being placed in the cluster, (b) a tenant departing the cluster and (c) a change in a replica's resource demand, compute – for each timestamp t – the placement of tenants such that

- every replica db *active* at t (i.e., $creationtime(db) < t$ and $creationtime(db) + lifetime(db) \geq t$) is placed on a node,
- no node is in violation (i.e., $\forall i, r : nodeload^r_{i,t} < f_r \cdot c_r$),
- and the placement of a replica db can be (re-)assigned only (i) when db is admitted to a cluster, (ii) if a new replica cannot be placed due to resource fragmentation, which can be resolved by moving db, or (iii) if there would be a resource violation if the placement for timestamp $t-1$ were in effect, i.e., $\sum_{db \in tenants(N_i,t-1)} usage^r_{db,t-creationtime(db)} \geq f_r \cdot c_r$ (we record a resource violation at t for every node for which we (re-)move tenants due to (iii))

and the total number of resource violations is minimized. This is an *online problem*, meaning at any time t, nothing is known about the future tenant demands.

Resource Fragmentation: Even if $OR^r < 1$ for all resources r, there can still be violations, due to *resource fragmentation*. Consider the example of two classes of tenants; class A, which consumes 25% of a node's capacity, whereas class B consumes 80% of a node's capacity. Now, in a cluster of H nodes, if at least H tenants of class A have been placed using e.g., the *WorstFit* heuristic, then any incoming tenant of class B will trigger a violation (or a placement failure), even though up to 75% of the cluster's capacity is unused.

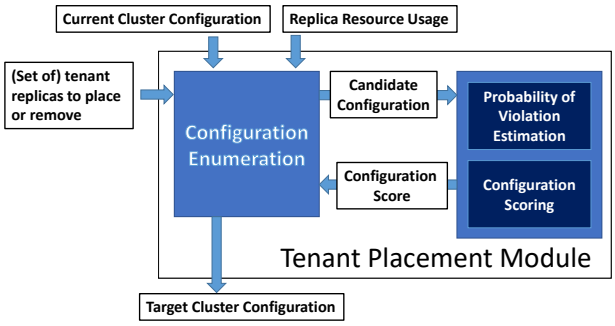

Figure 4: Components of the Tenant Placement Module

4 SOLUTION OVERVIEW

The key challenge in tenant placement are dynamic changes in tenant resource demands, as well as tenant departures, both of which are difficult to predict (e.g., see [29]), particularly for new tenants. As a result, our approach does not rely on precise predictions of node-level demand. Instead, we try to capture *both* (i) changes in demand over time, as well as (ii) the underlying uncertainty in the demand estimates. To do so, we utilize fast *Monte Carlo* simulations, which leverage previously observed demand curves, and are used to estimate – for a given set of tenants co-located on a node – a probability distribution for the aggregate resource demand on that node. Based on these, we estimate the *probability of a future resource violation* for co-located databases (see Section 5), and then use these probabilities in placement (see Section 6).

Interface to the Cluster Manager: We assume that the cluster manager provides a *tenant placement module* (Figure 4), which has interfaces that take as input (a) the current cluster *configuration*, (b) the current resource usage per replica and (c) tenant arrivals/departures. Based on these inputs, the placement module enumerates candidate configurations, which are passed to a scoring module. Finally, the configuration with the best score is implemented. In practice, a valid configuration has to satisfy additional constraints (e.g., minimum replica counts, affinity constraints, etc.).

5 MODELING VIOLATION PROBABILITIES

As discussed in Section 2, the issue with using snapshots of resource demands in placement is that demands may vary significantly over time. This is especially true directly after a new tenant has been placed, as data is imported, and memory caches are populated. Thus, a node on which a tenant has recently been placed, typically has a higher risk of a future violation than a node with similar current resource demand, but no new tenants.

To capture the distribution of changes in demand over time, our approach uses demand traces collected from previous tenants in *Monte Carlo* (MC) simulations. These simulations "replay" demand curves for replicas that are *similar* to the replicas we are considering

to co-locate on a node. By summing these demand curves, we obtain the estimated distribution of node-level demand over time.

5.1 Estimation of Resource Demand

Demand traces: We use a set of demand curves $\mathcal{D} = \{demand_1, \ldots, demand_g\}$ we have collected for tenants observed previously. A single element $demand_i \in \mathcal{D}$ encodes a sequence of resource demands for all resources $r \in \mathcal{R}$ over the full lifetime of the tenant, i.e., it contains $|\mathcal{R}|$ sequences (one for each resource) of the form $\langle usage^r_{db,0}, \ldots, usage^r_{db,lifetime(db)-1} \rangle$. We use the notation $demand^r_{i,\delta}$ to denote the demand for resource r at time δ (for $r \in \mathcal{R}$ and $1 \leq i \leq k$); again, δ refers to a time offset relative to when the tenant had been placed.

We use $\mathcal{D}$ to simulate the future resource demands for a set $\mathcal{S} = \{db_1, \ldots db_l\}$ of databases co-located on a node N. First, to illustrate the idea, we will describe simple approach, which ignores similarity between traces and the current databases they are used to simulate. Initially, for each tenant db_j we draw the traces from $\mathcal{D}$ at random to simulate a possible resource demand curve for N; here, the index of the trace used to simulate database db_j is denoted by o_j and drawn from $[|\mathcal{D}|] := \{1, \ldots, |\mathcal{D}|\}$. Subsequently, we introduce constraints on the set $\mathcal{D}$ used to simulate a concrete replica db, referring to the resulting subset of $\mathcal{D}$ as $\mathcal{D}(db)$. Based on these indices $(o_1, \ldots, o_l)$, the (estimated) aggregate demand on node N for a resource r at time t can be written as:

$$\sum_{j=1}^{l} demand^r_{o_j, t-creationtime(db_j)}.$$

Now, by repeating this process of drawing offsets for all possible combinations of indices $(o_1, \ldots, o_l)$, with each $o_j \in [|\mathcal{D}|]$, we obtain an estimated distribution of node load on node N over time. Based on this distribution, we can estimate the likelihood for a resource violation as the fraction of such index-vectors that result in a violation at some time-point t (see Equation 1 on the next page).

Monte Carlo Simulation: Evaluating all possible $(o_1, \ldots, o_{|\mathcal{S}|})$ in Equation 1 for violations has prohibitive overhead. Instead (see Algorithm 1), we perform Monte Carlo simulations during which we repeatedly sample an (uniformly) random assignment of offsets (line 5) and evaluate whether it results in a violation (see line 7); the fraction of such assignments (see line 11) is an unbiased estimate of the probability of violation as defined in Equation 1 (if the set of of eligible traces is constrained to $(D)(db_k)$).

Using traces *similar* to existing tenants: To improve the accuracy of the MC simulation, we further restrict the set of demand curves from $\mathcal{D}$ used to simulate a database db_k to curves from databases that are *similar* to db_k, in the sense that they share a set of common features. Concretely, we restrict the traces used to simulate db_k to traces of databases of the same *tenant class*, further differentiating between *primary* and *secondary* replicas.

Furthermore, for a database db_k which has been placed on the cluster already, we are able to observe (a prefix of) the demand curve and (a lower bound on) lifetime. So, when simulating such a database db_k, which was placed in the cluster L units of time ago, we only use traces of past tenants whose lifetime was at least L. This allows us to obtain more accurate estimates of resource load when estimating the probability of violations, as the different properties of demand curves are correlated: for example, longer-lived tenants are more likely to have higher resource demands.

We extend this idea to other resources: for a database db_k in the cluster, we track, for each resource r, the maximum demand $LBMax_r$ observed so far. Then, when simulating db_k, we only use traces for which, at some time, demand for r is at least $LBMax_r$. We use the notation $\mathcal{D}(db_k)$ to describe the subset of $\mathcal{D}$ that satisfies the constraints derived from db_k's observed demand curve.

Algorithm 1 Estimating the Probability of a Violation

```
1:  procedure ProbViolation(S) // Input set of tenants on N
2:      NumViolations := 0
3:      for l ∈ {1, ..., NumRepetitions} do // Main Monte Carlo loop
4:          for k ∈ {1, ..., |S|} do //
5:              o_k = random offset of trace in D(db_k)
6:          end for
7:          if ∃r, t : ( Σ_{j=1}^{|S|} demand^r_{o_j, t−creationtime(db_j)} ) ≥ f_r · c_r then
8:              NumViolations = NumViolations + 1
9:          end if
10:     end for
11:     Return (NumViolation/NumRepetitions)
12: end procedure
```

5.2 Reducing the Computational Overhead

Because the computation of $Pr_{Violation}(\mathcal{S})$ is part of the inner loop executed when enumerating candidate configurations (see Figure 4), it needs to have low computational and memory overhead. As a result, we introduce two optimizations to this process.

First, storing $\mathcal{D}$ may require significant memory, as a trace can contain large numbers of data points. Thus, we compress $\mathcal{D}$ by storing, instead of each demand curve, a compact model of it. Using these models has little effect on the estimated probability of violation, but reduces the required memory footprint significantly.

Second, evaluating whether a sample of traces $(o_1, \ldots, o_{|\mathcal{S}|})$ results in a violation requires computing the sum of tenant demands for *all* time points where multiple databases are active on a node, which can be computationally expensive. To reduce this overhead, we leverage properties of the compressed models, which allow us to evaluate if a violation occurs by evaluating only $|\mathcal{S}|$ time points.

5.3 Trace Compression

Based on the observations made in Section 3.1 on the typical shapes of demand curves, we now formulate a simple compressed representation of them. Concretely, we use three different representations, one each for the three relevant resource types – disk space, memory and CPU. Obviously, these representations may be significantly different than the original trace at individual time points. However, the purpose of these compressed traces is not to accurately replay the *individual* demand of a specific tenant, but to capture the aggregate tenant demand distribution across many traces sufficiently well to accurately estimate the probability of a violation.

Modeling disk demand: The disk model is based on the observation that most tenants grow to (or close to) their maximum disk demand and subsequently retain a disk demand at close to this level. Consequently, we model the disk demand using 3 parameters:

$$Pr_{violation}(\mathcal{S}) = \frac{\left|\{(o_1,\ldots,o_{|\mathcal{S}|}) \in [|\mathcal{D}|] \times \ldots \times [|\mathcal{D}|] \mid (o_1,\ldots,o_{|\mathcal{S}|}) \text{ results in violation}\}\right|}{|\mathcal{D}|^{|\mathcal{S}|}}$$
$$= \frac{\left|\left\{(o_1,\ldots,o_{|\mathcal{S}|}) \in [|\mathcal{D}|] \times \ldots \times [|\mathcal{D}|] \mid \exists r \in \mathcal{R}, t \in \mathcal{T} : \left(\sum_{j=1}^{|\mathcal{S}|} demand^r_{o_j, t-creationtime(db_j)}\right) \geq f_r \cdot c_r\right\}\right|}{|\mathcal{D}|^{|\mathcal{S}|}} \quad (1)$$

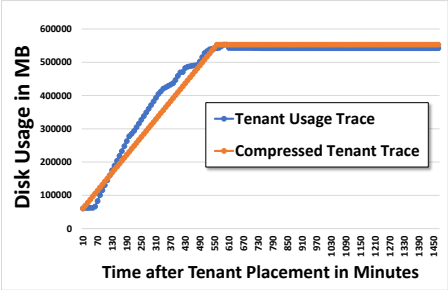

Figure 5: Compressing the Disk Demand curve of Figure 2

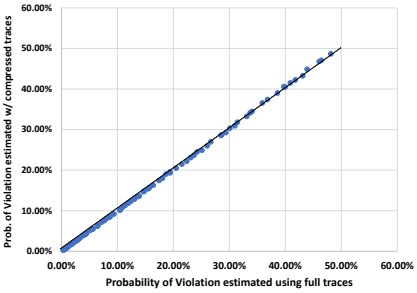

Figure 7: Evaluation of Compressed Models

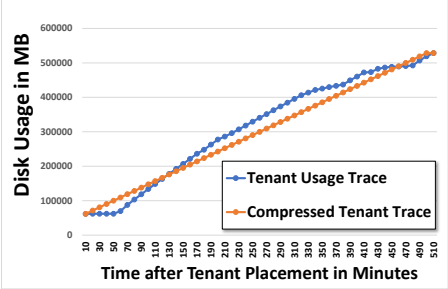

Figure 6: Compressing the Disk Demand curve of Figure 3

(1) The duration G_{disk} of the growth phase of the tenant, (2) the maximum size M_{disk} which the tenant grows to, and (3) the initial disk demand I_{disk} first reported for the tenant. Note that these three parameters are chosen individually for every trace in $\mathcal{D}$ that we represent in this compressed form. Based on these parameters, we then model the disk demand of a tenant db at a time-point δ as

$$demand^{disk}_{db,\delta} = \begin{cases} I^r_{disk} + \frac{\delta}{G_{disk}} \cdot M_{disk}, & \delta \leq G_{disk} \\ M_{disk}, & \text{otherwise.} \end{cases} \quad (2)$$

To illustrate the effect of compressing the real-life demand curves shown in Figures 2 end 3, we plotted them with their compressed representations in Figures 5 and 6; for these examples, the compressed representation matches the real demand curve quite well.

Modeling memory demand: Because memory demand curves follow a similar overall pattern as disk demand, we use the same representation for memory demand as we use for disk, again fitting 3 parameters G_{mem}, M_{mem} and I_{mem}; however, as discussed in Section 3.1, the distributions of the parameters are considerably different – for example, tenants often reach their 95% of their peak memory demand at a much later time than 95% of their disk demand.

Modeling CPU demand: We found that CPU demand typically does not follow a simple pattern, but changes rapidly and often unpredictably (e.g., see [30]). Thus, we do not model any long-term trends for CPU, but use the 90th percentile of the observed CPU usage $U90_{cpu}$. We selected a relatively high percentile to err on the side of over-estimating the likelihood of a violation. In addition to the compressed representations, we also retain the lifetime L of the tenant in question. We can now represent the demand curves for a replica using 8 parameters: M_{disk}, G_{disk}, I_{disk}, M_{mem} G_{mem}, I_{mem}, $U90_{cpu}$ and L, so $\mathcal{D}$ becomes a set of 8-tuples.

Effect of Compression: The key metrics for this representation are (a) the resulting reduction in storage space required to represent $\mathcal{D}$, and (b) the effect on the estimated probability of violation (Equation 1). To evaluate the latter, we set up an experiment where we placed e tenants (with e ranging from 10 to 200), chosen at random from $\mathcal{D}$, on a single node. The creation times for the tenants are staggered, such that a new tenant is placed every 10 minutes. We replay the corresponding traces (using the real-life tenant traces, which we will describe in more detail in Section 8) and check for violations, repeating the experiment with 10000 different e-tenant samples from $\mathcal{D}$ (for each value of e) to estimate the probability of violation. We then repeat this experiment, using the compressed representations instead of the full traces. Figure 7 shows the resulting probability of a violation based on the full traces (on the X-axis) and the compressed traces (on the Y-axis), with every individual point corresponding to a different value of e. There is an almost exact correspondence between the estimates; only for large numbers of tenants we see a very slight increase in the estimated probability.

Reducing Computational Overhead: As described in Section 5.2, evaluating if an assignment of trace offsets $(o_1,\ldots,o_{|\mathcal{S}|})$ results in a violation requires computing the aggregate demand for all time points t for which multiple tenants are present on a node N.

We reduce the overhead of this step by leveraging the monotonicity of the compressed demand curves. Concretely, for a set

$\mathcal{S} = \{db_1, \ldots, db_l\}$ of databases co-located on node N, we only need to consider $|\mathcal{S}|$ points in time. To define these points, we use the notation $tracelifetime(o_j)$ to denote the lifetime of the past replica whose trace is referred to by offset o_j (we use this notation to differentiate from $lifetime(db_j)$, which are unknown). Now, we define this set of time points as

$$\mathcal{T}' := \left\{creationtime(db_j) + tracelifetime(o_j) | j \in \{1, \ldots, l\}\right\}.$$

Obviously, $\mathcal{T}' \subseteq \mathcal{T}$. Now, we will show that for a given sample $(o_1, \ldots, o_l)$ of offsets into $\mathcal{D}(db_1) \times \ldots \times \mathcal{D}(db_l)$ that if there is a time $t \in \mathcal{T}$ such that there is a violation at time t, then there exists a time $t' \in \mathcal{T}'$ such that there is a violation at time t' as well. Showing the implication in the opposite direction is trivial, as $\mathcal{T}' \subseteq \mathcal{T}$. Combined, these two implications mean that we only need to consider all $t' \in \mathcal{T}'$ when testing for violations.

Proof: Let r be the resource for which there is a violation at time t. Let $\mathcal{S}' := tenant(N, t)$ be the set of active tenants on N at time t. Then define $t' := \min_{db \in \mathcal{S}'} creationtime(db_j) + tracelifetime(o_j)$; intuitively, this time is the latest time that all tenants in $\mathcal{S}'$ are active on N. This means that $tenant(N, t) \subseteq tenant(N, t')$. Now, we know that – because of the way we defined the compressed demand models in Equation 2 – that the resulting demand curves for each resource r are monotonically increasing (note that, because we do not require strict monotonicity, this also holds for the CPU representation). Therefore, based on the compressed models, the demand for resource r on the node at time t' must be at least as large as the demand at time t. □

6 NEW PLACEMENT ALGORITHMS

In this section, we describe new placement algorithms, which use the estimated *probability of violation* to reduce the expected number of violations directly. For this, we estimate the expected number of violations as the sum of the probabilities of violation over all nodes, under the simplifying assumption that resource demands are independent across nodes.

Secondary placement criteria: Because probability of violation estimates on not over-subscribed nodes will be 0, potentially resulting in multiple placements having identical expected violations, we combine the estimates with a secondary heuristic to select the best configuration. It is known that simple heuristics, such as *WorstFit* perform well when resource demands do not change dynamically (e.g., [17, 18]). So, intuitively, we use the simpler heuristics for placement among not (highly) overbooked nodes, while probability of violation estimates are utilized when tenants are packed more densely. Concretely, we propose two algorithms: The first approach combines probability of violation estimates with the *BestFit* heuristic, the 2nd algorithm with the *WorstFit* heuristic.

Space of Configurations: For ease of exposition, we assume that a single replica *db* is placed at a time. The configuration-space therefore corresponds to all nodes that can accommodate the (estimated) initial demand of *db*. In Section 7, we then describe the integration of our approach with the *Service Fabric* cluster manager, which allows for multiple replicas to be placed/moved in parallel.

6.1 Probability of Violation + BestFit

The first algorithm we propose (see Algorithm 2) uses the *BestFit* heuristic as the secondary criterion. *BestFit* places *db* on the node that has the *smallest* amount of resource capacity remaining after placing *db*. BestFit concentrates tenants on a small(er) number of nodes. This reduces resource fragmentation, but may increase capacity violations when tenant demands increase.

Therefore, this algorithm uses the probability of violation estimates to rule out nodes that are packed "too densely" for a new tenant *db*: if there exist nodes for which the estimated probability of violation (after placing *db*) is below a threshold θ (see line 10), only these are considered as destinations for *db* (see the variable *Candidates*); if multiple such nodes exist, *BestFit* is used to select among them (line 14). Otherwise, *db* is placed such that the *expected number of violations* is minimized (see line 16).

Weighting Functions: One remaining question is how to define the "best fit" when there is a vector of resource demands $nodeload_i := \left(nodeload_i^{r_1}, \ldots, nodeload_i^{r_v}\right)$, as opposed to a single value. For this purpose, different *weighting functions* have been proposed (see e.g., [28]), which combine these to a single scalar. We have experimented with different weighting functions, and found two functions to perform best[1]: the first is the *FFDSum* heuristic [28] (also used in [17]), which is defined as

$$W_{Sum}(nodeload_i) = \sum_{j=1,\ldots,v} w_j \cdot nodeload_i^{r_j}, \tag{3}$$

where $w_j = \frac{1}{|\mathcal{DB}| \cdot c_j} \sum_{db \in \mathcal{DB}} usage_{db,now}^{r_j}$. Intuitively, this function assigns a weight to each resource, which corresponds to the aggregate resource demand within the entire cluster. The other weighting function we use is

$$W_{Max}(nodeload_i) = \max_{j=1,\ldots,v} nodeload_i^{r_j} / c_{r_j}, \tag{4}$$

which uses the maximum demand (relative to capacity) across all resources. When describing the algorithms, we simply use the notation $W()$ for the weighting function; each algorithm can be instantiated with a different weighting function.

Initial demands: When placing a new tenant, the initial resource demand is unknown, meaning we need to use an estimate. We use the notation $\overline{usage}_{db,\delta}^{r}$ to differentiate the estimated demand from the actual one. We will refer to this algorithm as *PrV-BestFit*.

6.2 Probability of Violation + WorstFit

The 2nd algorithm we propose (Algorithm 3) uses the *WorstFit* heuristic as a secondary criterion. The *WorstFit* heuristic is the "opposite" of BestFit in that it places a replica *db* on the node with the *largest* resource capacity remaining.

Thus, WorstFit spreads aggregate resource demand evenly across the cluster, making it an efficient heuristic for cases where multiple nodes have a small probability of violation if *db* is placed on them; however, this may cause resource fragmentation (see Section 3.3).

[1]We also evaluated (1) the unweighted average over all resources and (2) using *only* the resource that is the biggest bottleneck in the cluster. Both perform worse with respect to the total number of violations observed.

Algorithm 2 Combining the Probability of Violation with BestFit

1: **procedure** PLACETENANTPRVBF(Tenant db)
2: $Candidates := \emptyset$
3: **for all** $n \in Nodes$ **do**
4: $\mathcal{S}_n :=$ { All databases currently placed on node n}.
5: **if** $Pr_{violation}(\mathcal{S}_n \cup \{db\}) < \theta$ **then**
6: $Candidates := Candidates \cup \{n\}$
7: **else** $Delta_n := Pr_{violation}(\mathcal{S}_n \cup \{db\}) - Pr_{violation}(\mathcal{S}_n)$
8: **end if**
9: **end for**
10: **if** $Candidates \neq \emptyset$ OR $\min_{n \in Nodes} Delta_n = 1$ **then**
11: **for all** $n \in Candidates$ **do**
12: $nodeload_n := \left(nodeload_n^{r_1} + \overline{usage}_{db,0}^{r_1}, \ldots, nodeload_n^{r_v} + \overline{usage}_{db,0}^{r_v}\right)$
13: **end for**
14: $DestinationNode := \underset{i \in Candidates}{\arg\max}\ W(nodeload_i)$
15: **else**
16: $DestinationNode := \underset{i \in Nodes}{\arg\min}\ Delta_i$
17: **end if**
18: Return $DestinationNode$
19: **end procedure**

Algorithm 3 Combining the Probability of Violation with WorstFit

1: **procedure** PLACETENANTPRVWF(Tenant db)
2: $Candidates := \emptyset; HeldOut\_Cand := \emptyset$
3: $HeldOut\_Nodes := \underset{\mathcal{H} \subseteq Nodes, |\mathcal{H}|=M}{\arg\min} \sum_{i \in \mathcal{H}} W(nodeload_i)$
4: **for all** $n \in Nodes$ **do**
5: $\mathcal{S}_n :=$ { All databases currently placed on node n}.
6: **if** $Pr_{violation}(\mathcal{S}_n \cup \{db\}) < \theta$ **then**
7: **if** $n \in HeldOut\_Nodes$ **then**
8: $HeldOut\_Cand := HeldOut\_Cand \cup \{n\}$
9: **else** $Candidates := Candidates \cup \{n\}$
10: **end if**
11: **else** $Delta_n := Pr_{violation}(\mathcal{S}_n \cup \{db\}) - Pr_{violation}(\mathcal{S}_n)$
12: **end if**
13: **end for**
14: **for all** $n \in Candidates \cup HeldOut\_Nodes$ **do**
15: $nodeload_n := \left(nodeload_n^{r_1} + \overline{usage}_{db,0}^{r_1}, \ldots, nodeload_n^{r_v} + \overline{usage}_{db,0}^{r_v}\right)$
16: **end for**
17: **if** $Candidates = \emptyset$ **then**
18: $Candidates := HeldOut\_Nodes$
19: **end if**
20: **if** $Candidates \neq \emptyset$ OR $\min_{n \in Nodes} Delta_n = 1$ **then**
21: $DestinationNode := \underset{i \in Candiates}{\arg\min}\ W(nodeload_i)$
22: **else** $DestinationNode := \underset{i \in Nodes}{\arg\min}\ Delta_i$
23: **end if**
24: Return $DestinationNode$
25: **end procedure**

Therefore, our algorithm "holds out" M nodes, only allowing placement on them if the probability of violation is always larger than θ when placing on a node not held out (see line 17-18).

Similar to *PrV-BestFit*, this scheme selects the configuration that minimizes the *expected number of violations* when placing a tenant db on any node will result in an estimated violation probability over θ for that node (line 22). We refer to this algorithm as *PrV-WorstFit*.

6.3 Algorithmic Complexity

Both *PrV-BestFit* and*PrV-WorstFit* have two loops, the first of which iterates over all nodes in the cluster and the 2nd one over a subset of them, meaning they are invoked $O(|Nodes|)$ times. The main overhead in either algorithm is the first loop, which – for each node – computes the estimated probability of violation. With the optimizations introduced in Section 5.3, computation of $Pr_{violation}(\mathcal{S})$ requires up to $O(|\mathcal{S}|^2 \cdot NumRepetitions \cdot |\mathcal{R}|)$ operations: for each Monte Carlo iteration, we need to test if there is a violation for each resource $r \in \mathcal{R}$. This means to compute, for each resource, the aggregate demand (which involves adding up $|\mathcal{S}|$ tenant demands) at up to $|\mathcal{S}|$ distinct points in time. Note that $Delta_n$ can be computed as a side-effect of computing $Pr_{violation}(\mathcal{S}_n \cup \{db\})$), and therefore does not introduce additional overhead. So the overall complexity of either algorithm is $O(|Nodes| \cdot |\mathcal{S}|^2 \cdot NumRepetitions \cdot |\mathcal{R}|)$.

7 INTEGRATION WITH SERVICE FABRIC

Next, we will describe how to integrate probability of violation estimates into an industrial-strength cluster manager, *Service Fabric*. The SF codebase is available as open source [16]. Concretely, we describe a modification of its *Placement-and-Load-Balancing* (*PLB*) component (also called *Cluster Resource Manager*). PLB has the ability to place/move multiple replicas at the same time, with the space of valid configurations defined by complex constraints (e.g., service affinity [1], max. replicas per fault/upgrade domain [2], etc.).

To navigate this large space with low overhead, the configuration enumeration component is based on *Simulated Annealing* [21] (SA). Any cluster configuration is associated with an *energy function*, which assigns it a score, with low scores being more desirable. When exploring the state space, the SA algorithm generates a random move[2] (e.g., moving a replica) and computes the energy function for the resulting configuration. Depending the new energy value, the new configuration is adopted with a certain probability (see [20], Section 5.3) and used for further exploration.

Since the energy function determines which configuration is chosen, any change to placement behavior requires modifying it. The two challenges to address here are (a) how to modify the energy function to use the probability of violation estimates and (b) to ensure that configuration scoring continues to be highly scalable.

Modifying the energy function: The energy function in PLB [16] has 3 components, which are combined into a single score:

(1) The number of failovers needed to reach a configuration.
(2) An *imbalance penalty* that quantifies how imbalanced resource demand is within a cluster. For this, PLB uses the (weighted) average of the standard deviations of node-level resource usage across all nodes in the cluster.
(3) A *fragmentation penalty* assigned when resource demand is placed on a held-out node, to avoid resource fragmentation.

The modification we make is to add the *expected number of future failovers* for the candidate configuration to the 1st component of the energy function (multiplied by a weight $w_{failovers}$). The expected

[2]With additional logic used to correct any resulting constraint violations.

number of future failovers is computed using the expected number of violations (see Section 6).

Fast energy computation: Computing the original PLB energy function requires computing its three components: here, the *number of failovers* needed for a target configuration is tracked during the generation of new configurations, and the *fragmentation penalty* is straight-forward to compute. To speed up the computation of the *imbalance penalty*, PLB uses a special *accumulator* data structure. Each accumulator maintains aggregates over the reported tenant demands for a specific resource. When a new configuration is generated by moving a replica, the accumulators compute the new energy function in $O(\mathcal{R})$ operations, using the incremental method of computing the standard deviation.

We use these accumulator structures to compute the violation probabilities as well, maintaining separate accumulators for every Monte Carlo iteration in Algorithm 1. For further scalability, we do not compute the demand for *all* time points in $\mathcal{T}$ (see Section 5.2), but instead report, for a database db_j, the values of $\max_t(demand^r_{o_j,t})$ to PLB, which are then used by the accumulators. Implicitly, this makes the assumption that the lifetimes of all tenants on a node are sufficiently large for each tenant's resource demand to reach their maximum before the first tenant departure. This simplification may overestimate violation probability, but reduces the overhead of the score computation by a factor of $|\mathcal{T}|$, allowing us to compute the new energy function in $O(|\mathcal{R}| \cdot NumRepetitions)$ operations. We evaluate the resulting overhead in Section 8.5.

8 EXPERIMENTAL EVALUATION

In this section, we compare the algorithms from Sections 6 and 7 to approaches from research and industrial practice. We evaluate three aspects: (a) the quality of tenant placement, (b) the overhead of the probability of violation computation and (c) the robustness of the new techniques, which we evaluate by varying several parameters, such as the set of resource demands, the cluster hardware, etc.

We further study if the violation probability estimates are essential to the observed improvements. For this purpose, we evaluate if other modifications to existing algorithms can yield comparable gains. Concretely, we study the effect of (a) varying the initial resource demand estimates for new tenants and (b) using techniques that predict future tenant demand.

8.1 Experimental Methodology

Our experiments use a mix of simulations (to capture effects over long time periods and at scale), which we describe in Sections 8.2 and 8.3, as well as a real cluster deployment, described in Section 8.4. For the experiments, we use resource usage traces of real customers in the simulation loop, allowing us to generate realistic resource demand curves without executing the underlying SQL workloads.

Simulation Details: To simulate clusters for the algorithms using Service Fabric's *Placement-and-Load-Balancing* (PLB) component (see Section 7), we use an industrial-strength cluster simulator originally developed to debug Service Fabric placement decisions. It uses existing PLB interfaces to report resource demand and tenant arrivals/departures, and invokes PLB's algorithm for tenant placement and violation resolution.

Because the simulation covers all relevant input interfaces to PLB (including SF configuration parameters), and we intercept and implement the PLB output in the simulated cluster, these simulations are faithful to the PLB behavior in real clusters. The simulator also implements all placement constraints used in production.

Since this simulator requires PLB in the simulation loop, other algorithms we want to evaluate with this simulator need to be implemented inside PLB. This is challenging for some algorithms, e.g., [31], which cannot easily be realized as part of the central PLB loop. Therefore, to compare the algorithms of Section 6 to the techniques proposed in [31], [17], [28], or heuristics such as *WorstFit*, we use a 2nd simulator without the PLB component, which uses the same cluster representation and interfaces.

Resolving Violations: For the PLB-based algorithms, we re-use PLB's internal logic to decide which tenants to fail over after a violation. However, most placement algorithms we compare to do not specify which tenants to fail over. Therefore, when evaluating non-PLB based placement techniques, we need to specify how to select tenants for failovers. The logic we use is inspired by industrial cluster managers, which use a (partial) ordering of tenants to determine which tenants are evicted/preempted: for example, *Kubernetes* uses a combination of *priorities* assigned to a POD in combination with their resource usage [22]. In *Service Fabric*, "*Move Costs*" [6] are assigned to tenants, with tenants of lower move costs (everything else being equal) being failed over first.

Concretely, we order tenants by aggregate resource usage, with smaller tenants being moved first. The reasons for this ordering are that (a) tenants with smaller resource usage typically correspond to less expensive tenant classes (i.e., higher-paying customers are moved less) and (b) "smaller" tenants can be placed more easily.

Compared Techniques: We compare our approach to state-of-the-art placement algorithms from research, namely techniques used for *Virtual Machine* placement [28], cluster scheduling [17] and placement using on-line vector (re-)packing [31]. In addition, we compare to techniques used in industry, such as the placement logic used in *Service Fabric*, as well as different placement heuristics (concretely, *BestFit* [39] and *WorstFit* [39]) used in *Kubernetes* [23].

Experimental setup: To assess the effect of different levels of over-subscription, we vary how densely tenants are packed: initially, each cluster is filled with tenants until a target *tenant density* is reached. This *tenant density* is defined as the cluster-level over-subscription ratio OR^r with the resource r being CPU cores. During experiments, we maintain this tenant density by, after tenant departures, admitting new tenants until the target tenant density is reached again. New tenants are chosen uniformly at random, with each compared algorithm placing the same sequence of tenants.

To set the hardware specs used in the simulations, we assume that the node capacities for industrial DBaaS providers correspond to the documented resource-limits for single DB instances (e.g., see [33, 36]): based on this, a node in our simulated cluster has 80 CPU vCores, 5.1 GB of memory per vCore and 5TB of local disk.

Demand traces: We use 4 different sets of demand traces, each collected from a different geographical regions, containing several million distinct tenants. Each trace contains the resource usage at 10-minute granularity, for 3 resources: *CPU*, *main memory* and *local disk* usage. The regions differ significantly in the distribution of tenant classes and the resource usage per tenant. We use 115K

traces (sampled at random) to populate $\mathcal{D}$, requiring 7.19 MB of storage after trace compression. The remaining traces are used to simulate tenants in our experiments. Estimating the probability of violation uses $NumRepetitions$ = 100 MC iterations; as discussed in Section 6.3, the computational overhead of tenant placement depends on the value of $NumRepetitions$ – we found experimentally that the choice of 100 repetitions to give a good trade-off between placement quality and overhead, as larger values did not result in substantial reduction in the number of violations.

8.2 Evaluation: Modified PLB energy function

In these experiments, we compare the original Service Fabric PLB component with the modified version described in Section 7. Here, the simulated clusters contain 40 nodes, divided into 10 upgrade domains and 5 fault domains (using the logic described in [2]). For each resource, we create a *custom metric* [5] in SF, which is used to report resource usage to PLB; we assign identical *metric weights* to each resource. In the modified scoring function, we use $w_{failovers}$ = 0.10.[3] Each experiment spans 10 days of simulated time, and is repeated 3x, with the average number of violations being reported. When comparing algorithms across different experimental setups, we average the *ratio* of the numbers of violations between the different algorithms over all experiments; we compute the averages in this way, so that the experiments with high tenant density and many violations do not dominate the comparison.

Results: Figure 8 shows the violations seen for different tenant trace regions and tenant densities. As we can see, the modified PLB component consistently outperforms the original PLB, performing better in 15 of the 16 experiments (where each pair of bars in Figure 8 corresponds to one experiment), and being tied once. Averaging the violations ratios, the original PLB energy function results in 2.6x as many violations as the modified one on average.

Importance of the Probability of Violation Estimates: To evaluate if the probability of violation estimates are necessary for the observed improvements, we conducted two experiments that test if similar gains are possible using simpler changes to PLB:

Varying the initial demand: As the resource demands for new tenants are unknown at placement time, a key decision is which demand estimate to use for them. Thus, we evaluated the effect of varying this initial estimate. For this, we computed the distribution of resource demands for different tenant classes and resources, and evaluated using different percentiles of these distributions as the initial demand estimate reported to PLB; concretely, we use the 50th, 80th, 90th, 95th and 99th percentile, in separate experiments.

Repeating the earlier setup, we found that the new initial demands can reduce violations; however, the use of probability of violation still results in larger improvements. Overall, the best-performing initial demand estimates combined with the original PLB resulted in 1.5x as many violations as the modified PLB.

Prediction of future demand: Some recent approaches (e.g., [24, 30]), have used (time series) prediction techniques to estimate future tenant demand. Therefore, we compare our approach to algorithms that have access to such predictions. A large number of prediction techniques exist, so we – instead of evaluating all of them – use an *oracle* instead, which provides *exact* predictions of resource demand for up to 24 hours into the future; the goal is to study an upper bound on the impact of point predictions on simpler algorithms.

In general, demand predictors require observing tenant demand for some time before issuing predictions (e.g., to identify periodic behavior); for example, in their experimental evaluation [30] requires 3 days of history (see Section 5.3.1) to train models on DBMS demand data. Thus, in our setup, the oracle provides predictions *only* for tenants that have been placed on the cluster for 24 hours already. These predictions are then integrated into PLB by reporting, instead of the current resource demand, the maximum resource demand in the prediction interval to the placement component.

The results when the oracle is used are shown as part of Figure 8. Interestingly, while the oracle predictions improve the performance of the original PLB in 7 (of 16) experiments, they also perform worse in several of them. Overall, performance improves slightly when using the oracle; still, compared to our approach, the combination of PLB + oracle results in 2.5x as many violations. This gain is much less than the improvement seen when varying the initial demand, which supports our intuition that the largest source of uncertainty is the initial growth after a tenant is placed.

8.3 Evaluating *PrV-BestFit* and *PrV-WorstFit*

Next, we compare the *PrV-BestFit* and *PrV-WorstFit* algorithms to heuristics used in industrial cluster managers, namely *BestFit* [39], *WorstFit* [39], and *minimizing the sum of the standard deviations* of resource demands across all cluster nodes. We evaluate *BestFit* and *WorstFit* for both weighting functions described in Section 6. Furthermore, we evaluate techniques proposed in academia, namely placement using the *weighted inner product* used in *cluster scheduling* [17] (Section 3.2) and in *Virtual Machine* placement [15], as well as *online vector packing with re-packing* in [31].

Experimental setup: Each experiment spans 300 hours of simulated time, and is repeated 20x, with average number of violations being reported. In each experiment, we simulate a 40 node cluster. For *PrV-WorstFit*, we use $M = 3$ held out nodes; for both *PrV-WorstFit* and *PrV-BestFit*, we use the W_{Max} weighting function and set $\theta = 0.01$. Since the non-PLB simulator does not enforce constraints on replicas per fault/upgrade domain, the resulting placement problem becomes simpler, with fewer violations compared to the experiments in Section 8.2. As a result, we consider higher tenant densities in these experiments.

Furthermore, we evaluate a 2nd hardware spec, which has 128 CPU cores (corresponding to the maximum vCores available to DB instances by one DBaaS provider [33]), and identical disk/memory capacity. Because tenant density is defined via the number available CPU cores, we limit these experiments to smaller tenant densities.

Results: The results are shown in Figure 9. Because of the large number of approaches compared, we – for clarity – do not report all results, but, for each setting, only report the number of violations from the *best-performing* algorithm not using probability of violation estimates, and compare it to *PrV-BestFit* and *PrV-WorstFit*.

Hardware spec 1: Overall, *PrV-WorstFit* is either equal or better than the best-performing approach that does not use probability of violation estimates (and no oracle) in 50 of the 52 experiments; on

[3] This value was determined experimentally with $w_{failovers}$ varied between 1.0 and 0.1; $w_{failovers}$ = 0.1 resulted in the best performance in terms of violations observed.

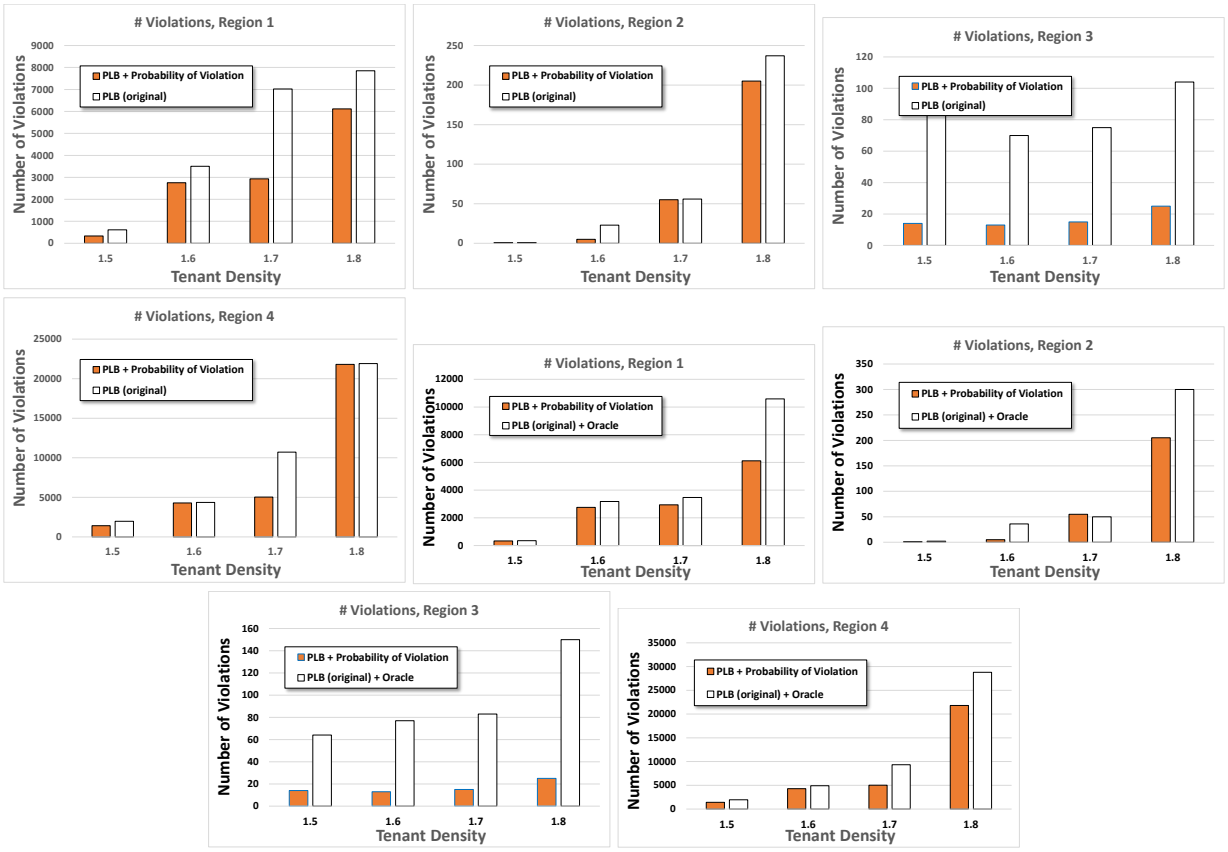

Figure 8: Evaluation: new PLB scoring function (compared to the original PLB and PLB + Oracle predictions)

average, the best results for techniques without probability of violation estimates show 2.57x as many violations as *PrV-WorstFit*. When compared to algorithms using oracle predictions, *PrV-WorstFit* is still equal or better than the best-performing algorithm in 49 of the 52 experiments, with the best results for techniques without probability of violation resulting in 2.27x as many violations.

In contrast, *PrV-BestFit* does not consistently outperform the other algorithms, as it places tenants on fewer nodes than *PrV-WorstFit*; this eliminates resource fragmentation, but increases the risk of violations. Still, on average, the best-performing algorithms, that do not use probability of violation estimates, show 2.6x as many violations as *PrV-BestFit*, and 1.74x as many when the oracle is used.

Hardware spec 2: Here, we observe *PrV-WorstFit* outperforming all algorithms without probability of violation estimates in every experiment. The average improvement ratio is 2.1x (no oracle), and 1.6x (with oracle predictions). *PrV-BestFit* outperforms the best-performing algorithm without probability of violation in 14 of 16 experiments (no oracle), and 10 of 16 experiments (oracle). The average improvement ratio is 2.0x (no oracle), and 1.4x (oracle).

Varying the Initial Demand: Similar to Section 8.2, we varied the initial demand reported for a new tenant (on hardware spec 1 only), comparing the best-performing competing algorithm to *PrV-WorstFit*. We did not see a significant change in the performance: across the different percentiles, the average ratio at which *PrV-WorstFit* outperformed the best-performing algorithm without probability of violation estimates varied between 2.42x and 2.57x.

8.4 Evaluation: Real Cluster Deployment

To assess if the observed gains continue to hold in a real SF cluster deployment, we repeat a subset of the experiments of Section 8.2 on a real 40 node cluster (executing within Microsoft Azure) [3].

Experimental Setup: Each tenant is deployed as a separate application (with the corresponding number of replicas). To obtain realistic resource demand profiles without executing real customer SQL workloads (which is not possible, due to customer IP), each application reports to PLB resource usage corresponding to real customer traces (see Section 8.1), selected at random. As we use the same reporting interfaces used in production clusters, the demand distributions observed by PLBs correspond to the ones in actual production clusters. Because of the cost and time required, we repeat the experimental setup of Section 8.2 for two of the four regions (regions 1 and 4); each experiment covers a week of time.

Results: The overall reduction seen in the number of violations is in line with the earlier, simulation-based experiments, with the unmodified Service Fabric exhibiting, on average, 2.93x as many violations (as before, averaging the ratios of violations) as the variant of PLB incorporating the probability of violation estimates.

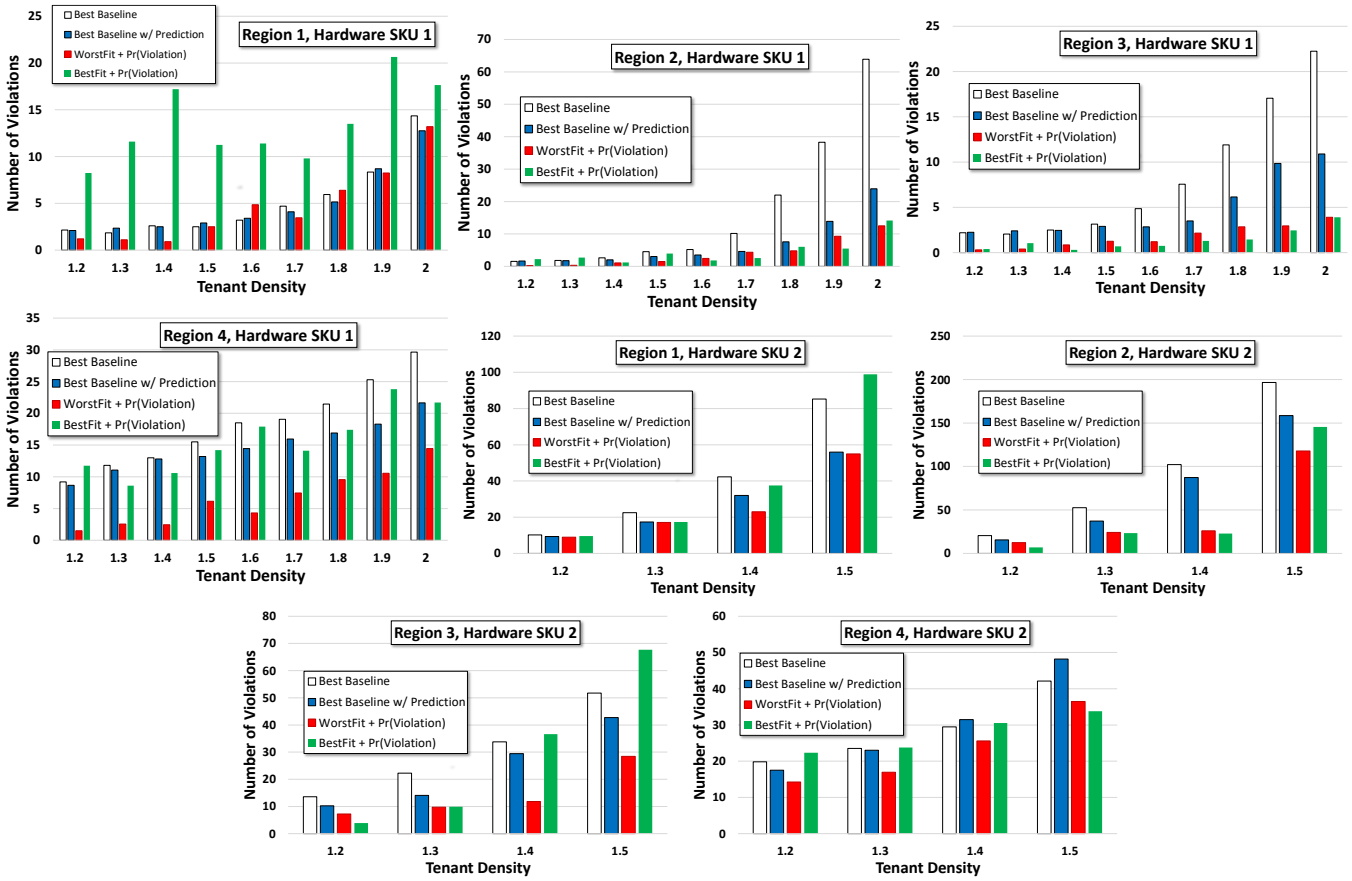

Figure 9: Evaluation of Violations (Hardware specs 1 & 2)

8.5 Evaluation: Overheads

In this section, we describe the overheads required for the algorithms described in Sections 6 and 7. The experiments were conducted on an Intel Xeon CPU E5-2660 v3 (2.60 GHz) wit 192 GB of RAM, running Windows Server 2019 Datacenter.

Memory overhead: All proposed algorithms require a storing the repository $\mathcal{D}$ of (compressed) traces described in Section 5.3 in main memory (including the tenant class), requiring a total of 66 byte per trace. The memory overhead required by PLB to maintain additional aggregates for evaluation of the new scoring function is $H \cdot NumRepetitions \cdot 8$ bytes. For the experimental setup of Sections 8.2 and 8.4, this corresponds to 32KB, which is negligible.

Computational overhead: For the modified PLB component, we measured time to enumerate and score 1K configurations as part of the simulated annealing (for the experiments of Section 8.2 and 8.4): Here, the original PLB component requires about 432ms, whereas the modified component required 890ms on average. However, this increase in time is still a very small fraction of the overall time required for the creation of a new tenant.

For the algorithms described in Section 6, we measured the average time required to place a single tenant (including the required MC simulations, whose overhead vary with the tenant density) for each individual experiment in Section 8.3. For *PrV-WorstFit* these averages (depending on tenant density) were between 8.58 ms and 65.4 ms. For *PrV-BestFit* they range from 9.66 ms to 76.7 ms.

9 CONCLUSION

Over-subscribing resources can significantly increase resource utilization in multi-tenant database clusters, but comes with the risk of potentially disruptive resource violations. In this paper, we describe new tenant placement algorithms that significantly reduce the incidence of such violations. The key insight is the use of a fast estimator of the probability of a future violation, based on historical usage data. We proposed three different algorithms that leverage these estimates, and prototyped one of them in the *Service Fabric* cluster manager. Experiments using production traces from Azure SQL DB, including a cluster deployment of the modified Service Fabric code, show that using the probability of violation estimates results in a significant reduction in the incidence of violations, compared to the current state of the art in tenant placement.

ACKNOWLEDGMENTS

We are very grateful for the feedback and suggestions given by Luke Marshall, Srikanth Kandula, Pankaj Arora, Matthaios Olma, Lukas Maas and Jiaqi Liu, as well as the anonymous reviewers.

REFERENCES

[1] Microsoft Azure. 2020. Configuring and using Service Affinity in Service Fabric. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-advanced-placement-rules-affinity. Last accessed: 2022-07-12.

[2] Microsoft Azure. 2021. Service Fabric Cluster Resource Manager. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-cluster-description. Last accessed: 2022-07-13.

[3] Microsoft Azure. 2022. Create a Service Fabric Cluster. https://docs.microsoft.com/en-us/azure/service-fabric/scripts/service-fabric-powershell-create-secure-cluster-cert. Accessed: 2022-07-13.

[4] Microsoft Azure. 2022. High availability for Azure SQL Database and SQL Managed Instance. https://docs.microsoft.com/en-us/azure/azure-sql/database/high-availability-sla. Last accessed: 2022-07-12.

[5] Microsoft Azure. 2022. 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. Last accessed: 2022-07-13.

[6] Microsoft Azure. 2022. Service Fabric Movement Cost. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-movement-cost. Accessed: 2022-07-13.

[7] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. 2012. "Cut Me Some Slack": Latency-Aware Live Migration for Databases. In *Proceedings of the 15th International Conference on Extending Database Technology*. 432–443.

[8] Sebastian Berndt, Klaus Jansen, and Kim-Manuel Klein. 2020. Fully Dynamic Bin Packing Revisited. *Mathematical Programming* (2020).

[9] Henrik I. Christensen, Arindam Khan, Sebastian Pokutta, and Prasad Tetali. 2017. Approximation and Online Algorithms for Multidimensional Bin Packing: A Survey. *Computer Science Review* 24 (2017), 63–79.

[10] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-Aware Database Monitoring and Consolidation. In *SIGMOD International Conference on Management of Data*.

[11] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow.* 4, 8 (2011).

[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management *(ASPLOS)*.

[13] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 301–312.

[14] Leah Epstein and Meital Levy. 2010. Dynamic Multi-Dimensional Bin Packing. *Journal of Discrete Algorithms* 8, 4 (2010).

[15] W. Fernandez de la Vega and G. S. Lueker. 1981. Bin Packing can be solved within $1 + \epsilon$ in linear time. *Combinatorica* 1, 4 (1981), 349–355.

[16] Github. 2022. Service Fabric Codebase. https://github.com/Microsoft/service-fabric/.

[17] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. In *SIGCOMM*.

[18] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, David Dion, Esaias E Greeff, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *OSDI*. USENIX.

[19] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *22nd Symposium on Operating Systems Principles*. Association for Computing Machinery, 261–276.

[20] 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 *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*.

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by Simulated Annealing. *Science* 220, 4598 (1983), 671–680.

[22] Kubernetes. 2022. Kubernetes Pod Priority and Preemption. https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/. Last accessed: 2022-07-13.

[23] Kubernetes. 2022. What is Kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[24] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems *(SIGMOD)*.

[25] Justin Moeller, Zi Ye, Katherine Lin, and Willis Lang. 2021. Toto - Benchmarking the Efficiency of a Cloud Service. ACM SIGMOD.

[26] Hyun Jin Moon, Hakan Hacıgümüş, Yun Chi, and Wang-Pin Hsiung. 2013. SWAT: A Lightweight Load Balancing Method for Multitenant Databases. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*.

[27] Vivek Narasayya and Surajit Chaudhuri. 2021. Cloud Data Services: Workloads, Architectures and Multi-Tenancy. *Foundations and Trends in Databases* 10, 1 (2021), 1–107.

[28] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder. 2011. *Heuristics for Vector Bin Packing*. Technical Report. Microsoft Research.

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

[30] 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. *VLDB* 14, 2 (2020), 154–162.

[31] Kamali S. 2015. Efficient Bin Packing Algorithms for Resource Provisioning in the Cloud. *ALGOCLOUD* (2015).

[32] Jan Schaffner, Tim Januschowski, Megan Kercher, Tim Kraska, Hasso Plattner, Michael Franklin, and Dean Jacobs. 2013. RTP: Robust Tenant Placement for Elastic In-Memory Database Clusters. In *ACM SIGMOD*.

[33] Amazon Web Services. 2022. DB Instance Classes. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.DBInstanceClass.html. Accessed: 2022-07-13.

[34] Azure SQL. 2022. Azure SQL Database. https://azure.microsoft.com/en-us/products/azure-sql/database/#overview Accessed: 2022-07-10.

[35] Azure SQL. 2022. Azure SQL Database Serverless. https://docs.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview. Accessed: 2022-07-13.

[36] Azure SQL. 2022. Overview of Azure SQL Managed Instance Resource Limits. https://docs.microsoft.com/en-us/azure/azure-sql/managed-instance/resource-limits#hardware-generation-characteristics. Accessed: 2022-07-13.

[37] Rebecca Taft, Willis Lang, Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, and David DeWitt. 2016. STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments. In *ACM Symposium on Cloud Computing*. 388–400.

[38] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *EuroSys*. Association for Computing Machinery, Article 18.

[39] Wikipedia. 2022. Bin packing problem. https://en.wikipedia.org/wiki/Bin_packing_problem. Accessed: 2022-07-13.