Solver-In-The-Loop Cluster Resource Management for Database-as-a-Service

Arnd Christian König
Microsoft Research
chrisko@microsoft.com

Yi Shan
Microsoft Research
shayi@microsoft.com

Karan Newatia
University of Pennsylvania
knewatia@seas.upenn.edu

Luke Marshall
Microsoft Research
lumarsha@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

ABSTRACT

In Database-as-a-Service (DBaaS) clusters, resource management is a complex optimization problem that assigns tenants to nodes, subject to various constraints and objectives. Tenants share resources within a node, however, their resource demands can change over time and exhibit high variance. As tenants may accumulate large state, moving them to a different node becomes disruptive, making intelligent placement decisions crucial to avoid service disruption. Placement decisions need to account for dynamic changes in tenant resource demands, different causes of service disruption, and various placement constraints, giving rise to a complex search space.

In this paper, we show how to bring combinatorial solvers to bear on this problem, formulating the objective of minimizing service disruption as an optimization problem amenable to fast solutions. We implemented our approach in the Service Fabric cluster manager codebase. Experiments show significant reductions in constraint violations and tenant moves, compared to the previous state-of-the-art, including the unmodified Service Fabric cluster manager, as well as recent research on DBaaS tenant placement.

PVLDB Reference Format:
Arnd Christian König, Yi Shan, Karan Newatia, Luke Marshall, and Vivek Narasayya. Solver-In-The-Loop Cluster Resource Management for Database-as-a-Service. PVLDB, 16(13): 4254 - 4267, 2023.
doi:10.14778/3625054.3625062

1 INTRODUCTION

In Database-as-a-Service (DBaaS) settings, the service provider is responsible for maintaining the database software, resource management, backup/restore, point-in-time recovery and high availability of the service [37]. Examples of cloud relational DBaaS include *Amazon Aurora* [1], *Microsoft Azure SQL Database* [44] and *Google Cloud SQL* [19]. The DBaaS market size is expected to double from 12.0 Billion USD in 2020 to 24.8 Billion by 2025 [14]. DBaaS offerings are *multi-tenant*, i.e., multiple databases (aka *tenants*) from different customers share resources such as CPU, memory, disk I/O and storage. Individual tenants are hosted on sets of nodes called *clusters* (see Figure 1); for high availability, a tenant may have multiple *replicas* distributed across these nodes. On a node, one or more database processes, each assigned to a tenant, execute within one or more virtual machines; the node's resources are shared.

Proceedings of the VLDB Endowment, Vol. 16, No. 13 ISSN 2150-8097.
doi:10.14778/3625054.3625062

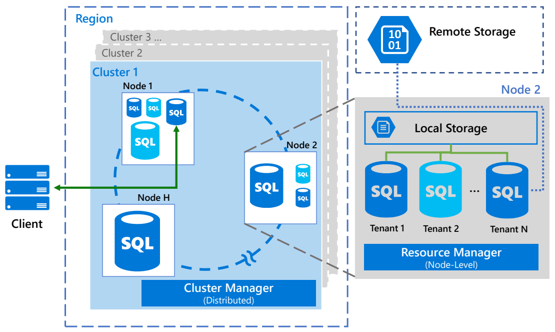

Figure 1: Example *Azure SQL DB* DBaaS cluster (from [31])

DBaaS Cluster Resource Management: Because resources on a node are shared between tenants, the key challenge in DBaaS cluster resource management is which replicas to assign to which nodes. We refer to this assignment as the cluster's *configuration*. Since tenants arrive and depart and their resource demands change over time, this is an *online* problem, closely related to *online vector packing with repacking* [13, 41], but with additional constraints. The goal is to place tenants such that all constraints, including constraints on node resource demands, are maintained for as long as possible, minimizing service disruption due to constraint violations. Resource management for DBaaS is particularly challenging, as DB tenants are long-lived [39], accumulate large state [37] and exhibit significant changes in resource demand over time [31].

Resource Over-Subscription: One challenge for cluster resource management are *over-subscribed* resources, meaning that the sum of resources promised to services on a node exceeds the node's capacity. Over-subscription can significantly improve cluster utilization, thereby reducing service costs for providers and subscribers, and is crucial to mitigating temporary capacity shortages (e.g., due to node failures). However, resource over-subscription makes it possible for resource demand on a node to exceed the node's capacity.

Resource Violations: If the resource demand on a node exceeds a threshold[1], this is considered a *resource violation*. If a resource violation persists, it becomes necessary to move tenant replicas to other cluster nodes. These moves (which we refer to as replica *failovers*) may further disrupt the service. Specifically, (a) queries executing on the failed-over replica may be canceled, (b) the contents of caches critical for query performance may be lost, (c) any database

[1]In practice, thresholds are set below the node capacities, allowing for time to move tenants before capacity is truly exhausted. For clarity of exposition, however, we use the node capacities as violation thresholds in this paper.

state on local storage has to be copied to the new node, and (d) the tenant itself may become temporarily unavailable. Consequently, service disruptions due to failovers are crucial to avoid.

Opportunities for Improvement: Because of the number of tenants in a cluster (DBaaS clusters may host 1000s of replicas), complex combinations of constraints (e.g., [5, 26]) and changing resource demands, cluster management techniques need to traverse a large search space. Because of this, and the need to make placement decisions quickly, industrial cluster managers use (greedy or randomized) heuristics that base decisions on *snapshots* of current resource usage. As we show, these fail to explore the space of candidate placements in an effective manner, and may miss optimal solutions. We describe this in detail for two widely used cluster managers, *Kubernetes* [30] and *Service Fabric* [24], in Section 2.1.

While this paper focuses on *relational* DBaaS settings, our approach is likely applicable to other resource management scenarios.

1.1 The Proposed Approach

This paper formulates DBaaS Resource Management as a combinatorial optimization task that leverages a *mixed-integer programming* solver [52] to quickly generate a solution with provable bounds on the objective function. It supports the full breadth of placement constraints [4, 5, 8] and cluster management features (e.g., *move costs* [11]) required for DBaaS settings, and has been implemented inside an industrial-strength cluster-manager (Service Fabric).

Challenges and Key Ideas: The use of constraint solvers for cluster resource management is far from being a panacea: e.g., both [24] (in the context of Service Fabric) and [20] (in the context of Cluster Scheduling) argue that combinatorial optimization results in unacceptable overheads (or poor solution quality) and proposed simpler heuristics. According to [47], no open-source cluster managers use constraint solvers and, anecdotally, nor do widely used enterprise cluster managers. This begs the question: why should our approach succeed when previous attempts have not?

There are several reasons why solvers are viable for our scenario. First, our optimization objective is specified as a sum of (expected) failovers. This makes it an effective way to characterize service disruptions, as we require resources to always be made available when needed, and any sustained violation entails failovers. Hence, the objective is highly effective at reducing service disruptions.

Because of this specific formulation, the objective value for intermediate solutions can be used to prune the search space: a candidate configuration with objective value X implies that no configurations requiring more than X failovers can be optimal; thus, only configurations that differ from the current one in the placement of $\leq X$ replicas need to be considered. Since most violations can be corrected with very few failovers, such pruning quickly reduces the search space, and, in turn, speeds up the optimization significantly.

Second, the formulation allows the solver to quickly derive bounds on the objective itself, and abort the search when a sufficiently good solution (e.g., within 1% of optimal) has been found. As we will explain, this is not the case for the objective used in Service Fabric, making it less suitable for our specific scenario.

Third, for scalability, nearly all decision variables in our formulation are binary, and we use only *linear combinations* of them in the objective and constraints. In contrast, a number of the techniques proposed in research used formulations that are significantly more expensive to solve. For example, [20] incorporates non-linear constraints, which result in more expensive optimization tasks [32].

Finally, because of the large state of a typical DBaaS tenant, which makes failovers disruptive, the consequences of "bad" placement in DBaaS settings can be severe. Consequently, even industrial cluster managers use seconds of wall-clock time for tenant placement/movement, allowing for some time for the optimization itself.

Results: In experiments using resource traces from real DBaaS tenants, and clusters mirroring DBaaS production settings, our approach reduces violations by 71% and failovers by 79% (on average) compared to Service Fabric, and violations by 54% and failovers by 68% (on average) compared to [31]. Moreover, it has significantly lower latency for small-to-medium clusters sizes, due to being able to abort the search when (near-) optimal solutions are found.

2 BACKGROUND

Cluster managers, such as *Kubernetes* [30], *Service Fabric* [24], *OpenShift* [22], *DRS* [50], etc. are ubiquitous in modern data centers where they are essential for numerous orchestration tasks. In this paper, we primarily consider two industrial cluster managers, *Kubernetes* [30], which is deployed by e.g., Amazon, Google, IBM and various Fortune 100 companies, and *Service Fabric*, which is used as the cluster manager behind one of the largest DBaaS offerings [9], *Azure SQL Database* [44], *Azure Cosmos DB* and *Dynamics 365*.

The role of the cluster manager is to compute a suitable *target configuration* in response to tenant arrivals/departures, resource shortages, node failures, etc., and implement these via replica movement. Target configurations need to satisfy a number of *hard constraints* such as *anti-affinity constraints*, which e.g., prohibit co-locating replicas within the same fault domain [5], *affinity constraints* [26], which co-locate dependent services, etc. We refer to configurations that satisfy *all* such constraints as *valid*. Among all valid configurations, the target configuration is chosen such that it optimizes a number of different metrics, such as the number of tenants moves required to reach the target configuration, etc. These are combined into a single criterion, called the configuration *scoring function*.

It is crucial that the cluster manager (1) efficiently generates valid configurations reachable from a (potentially invalid) cluster state, and (2) efficiently identifies target configurations that optimize the scoring function. Unfortunately, as we will discuss, industrial cluster managers often fail to provide either of these properties.

2.1 Limitations in the State of the Art

Kubernetes: In Kubernetes, the search space of target configurations is limited by the existing scheduling logic, which only allows moves[2] of replicas either (1) located on nodes that are experiencing resource shortages [29] or (2) in response to a higher-priority replica needing to be scheduled on the node of the replica to be moved [29]. In particular, Kubernetes does not support *cross node preemption* [28] where pods on a node N are preempted, in order to enable scheduling of a pod on a node K (with $K \neq N$).

[2]In Kubernetes, replica moves correspond to the eviction [29] or preemption [28] and subsequent re-scheduling of a pod.

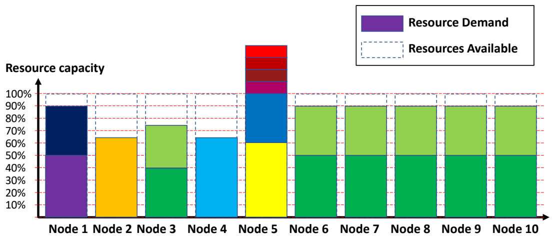

Figure 2: Constraint Violation on node 5

Since the Kubernetes scheduler places replicas one-at-a-time, this means that the configuration search is prone to local optima. As a result, Kubernetes may not be efficient at optimizing scoring functions and may miss valid configurations, or introduce unnecessary failovers when "fixing" invalid configurations.

Service Fabric: Service Fabric uses multiple algorithms in concert to fix constraint violations, which use a combination of random and greedy heuristics that fix one constraint at a time. This scheme does not systematically explore the space of valid configurations reachable with only few moves, even when given large amounts of time, which can mean unnecessary failovers or persistent constraint violations in practice. We will show that Service Fabric can fail to find valid configurations even for few (< 25) total replicas (see Section 7.1) and its constraint-fixing logic yields configurations which require significantly more failovers than needed (see Section 2.2).

The scoring function of Service Fabric (see [18], Score.cpp), is the (minimal) product of (a) the weighted sum of the standard deviations of all metrics [3][3] across all nodes and (b) the sum of *move costs* [11] of all replicas moved to reach the target configuration[4]. This makes it difficult to bound the number of replicas moved to reach the *optimal* target configuration, thereby constraining the search space, as additional replica moves may make the (a) term arbitrarily small.

To navigate the resulting search space, the configuration enumeration of Service Fabric uses *Simulated Annealing* [25] (SA). When exploring the state space, SA generates a random move (e.g., moving a replica) and computes the resulting scoring function. Depending on the new score, the new configuration is adopted with a certain probability (see [24], Section 5.3) and used for further exploration. Similar to the case of fixing constraint violations, this approach does not allow systematic exploration of parts of the search space.

As a result, Service Fabric may miss an optimal target configuration, and has no effective way to assess how far the current best candidate target configuration is from the theoretical optimum, and e.g., use this to bound the exploration time.

2.2 Experiment: Finding Valid Configurations

To demonstrate the limitations discussed above, we experimentally compare the failovers required to return the cluster to a valid state for different cluster managers and a specific instance of a constraint violation. For fairness of comparison, we use the same scoring function in all approaches: *minimizing the total failovers required.*

We consider the simple example of a 10-node cluster, with each node providing only a single resource. There are 22 single-replica

[3]These metrics typically correspond to demands for individual resources [3].
[4]The scoring function also includes penalties associated with insufficient free nodes and the total replica count, which we omit for ease of exposition.

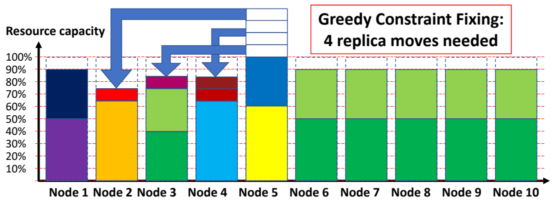

Figure 3: Service Fabric greedily moves 4 small tenant replicas

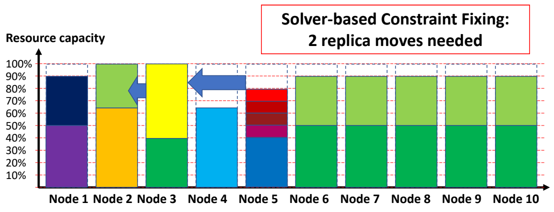

Figure 4: The solver is able to fix the violation using only 2 moves

tenants placed in the cluster, which are distributed in the cluster as shown in Figure 2: due to an increase in resource demand of a tenants on node 5, the resource demand exceeds the node's capacity.

When using Service Fabric to resolve this violation, it moves 4 tenants to three different nodes (see Figure 3). This is due to the constraint check logic, which *greedily* searches for moves that reduce the number of violations. However, there exists a solution requiring only 2 failovers (see Figure 4) which is found by the techniques of this paper, but not Service Fabric, despite the small number of tenants and the extensive search time (20 seconds) given to Service Fabric (whereas our approach requires < 1 second). Because Kubernetes only allows preemption of tenants on nodes experiencing a constraint violation, the 4-failover solution of Figure 3 is the best possible outcome for a Kubernetes-based scheduler as well.

Weighted Scoring: To illustrate that this issue extends to other optimization criteria, we consider a variant of our scenario, in which the failover of a tenant using many resources is more significant than the failover of a "small" tenant. Concretely, we use the failed-over tenants' resource demands as the optimization criterion.

Again, node 5 requires failovers to fix a capacity violation (see Figure 5). This node hosts 3 tenants that demand 33% of the node resource capacity (colored yellow) and one tenant that demands 35% (colored red). None of the remaining cluster nodes has more than 33% capacity available. A greedy approach to solving this violation moves 2 yellow tenants to nodes 2 and 4, thereby moving resources corresponding to 66% of a node's capacity (see Figure 6). In contrast, the solver moves only 39% of a node's capacity, by first moving

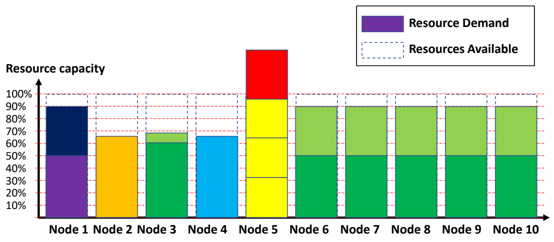

Figure 5: Replicas need to be moved off of node 5

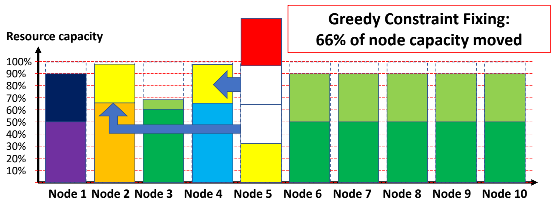

Figure 6: A greedy solution moves replicas using 66% of node capacity

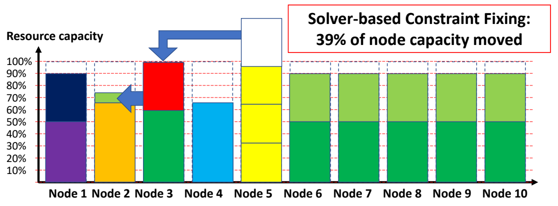

Figure 7: The solver moves only 39% of node capacity in total

a small tenant (requiring 4% of capacity) off of node 3, thereby creating space to move the red tenant there (see Figure 7).

3 RELATED WORK

Resource Management based on Vector Packing: [13, 41] have focused on optimizing the packing density, with only loose bounds on the incidence of failovers; for example, [41] was found to perform much worse in terms of violations and failovers than simple heuristics for realistic tenant distributions [31].
Resource Management in Database Clusters: [31] introduced an estimator of the *probability of a future capacity violations* for co-located tenants. We leverage this estimator in our optimization formulation, but go beyond [31] in that (a) we change the configuration search algorithm, (b) propose a new scoring function, allowing for efficient pruning of the search space, and non-trivial bounds on the optimality of an intermediate score. [33] proposes a system for resource optimization of cloud database services, with resources allocated statically to replicas (while in our scenario they are shared dynamically among co-located replicas), including a variant of the *Best Fit* heuristic with logic to avoid skew/fragmentation.

Other techniques have studied *different* optimization tasks: e.g., [42] minimizes the number of used servers, subject to constraints on service-level objectives and node load. In contrast, in our scenario, cluster sizes are fixed and service disruption is the optimization criterion. [16] manages resources based on the impact shortages have on workload, allocating the *minimum resources needed to satisfy a performance target*. Again, this differs from our scenario, where violations must be resolved, regardless of their performance impact. Finally, [35] considers a placement approach that protects against the failure of multiple servers without requiring tenant failovers. However, [35] models the resource demand as a single fraction of node capacity (i.e., no differentiation between resources) and does not account for changes to tenant resource demands.
Cluster Scheduling: Cluster resource management is related to *cluster scheduling* (e.g., [23, 48]). However, cluster scheduling scenarios differ in a number of respects: first, in many of these approaches, resources are *reserved statically* and remain assigned independently of whether they are used (e.g., see [48], Section 2.4). Moreover, the key metrics optimized in scheduling (e.g., *job completion times, fairness*) are very different from minimizing failovers.
Database Migration: How to migrate database replicas within a cluster to alleviate resource contention has been an active area of research as well [12, 15, 17, 36]. [36] studies *swap-removable* resource contention, which is resolved by swapping primary and secondary replicas. This approach is applicable to CPU over-subscription, but fails for other resources, such as disk space, as swaps do not necessarily reduce the disk footprint; moreover, it is not applicable to non-replicated tenants. [12] proposes *SWAT*, an end-to-end tenant migration framework, with the objective being to *minimize tenant downtime* during the migration itself, and minimizing the impact of the migration operations on workload throughput. However, which tenants to co-locate, in order to minimize disruptions is not studied. Thus, [12] complements this paper. The same holds for [15, 17], which study live migration of database replicas within a cluster.
Optimization in Cluster Management: [47] proposes a framework to translate constraints specified as SQL statements into a combinatorial optimization task. However, [47] does not specify which objective function (or constraints) to use in DBaaS settings, which is one of our key contributions; moreover, it is not clear if the objective we propose can be handled by their framework. [40] uses query optimization techniques to scale [47] to larger clusters.

4 PLACEMENT AS AN OPTIMIZATION TASK

In the following, we formulate the task of tenant placement and movement as an optimization problem. This formulation takes as input the current state of the cluster, and a (possibly empty) set of new tenants to be placed/removed. Assuming the placement problem is not over-constrained, it outputs a *target configuration* in which all active replicas are assigned a node, such that (a) all *hard constraints* are satisfied and (b) the *scoring function* is minimized (or – in the relaxed problem variant – within ϵ of the optimal score).
Constraints: For ease of explanation, we focus, in the following, on 4 constraints typical for DBaaS settings. We describe how to encode other constraints supported by Service Fabric in Appendix A.
Capacity Constraints: Aggregate resource demand for all replicas on a node may not exceed the node capacities for any resource.
Replica Anti-affinitys: To preserve tenant availability after node failures, no two replicas of a tenant may be mapped to one node.
Fault Domain Constraints: Nodes in DBaaS clusters are partitioned into *fault domains* (FD) [5], where a single fault domain contains nodes which share a single point of failure. To preserve availability, no two replicas of one tenant may be placed in one fault domain.
Upgrade Domain Constraints: Similarly, nodes are partitioned into *upgrade domains* (UD) [5], where an upgrade domain describes a set of nodes that are upgraded (and restarted) in parallel. No two replicas of one tenant may be placed to the same upgrade domain.
Configuration Scoring: The purpose of the scoring function is to minimize degradation in performance and availability of the DBaaS service. Such degradation can result from a number of causes, with the most important one being resource violations that affect service performance and necessitate replica failovers. Any scoring function in the DBaaS context ideally minimizes the likelihood of

such shortages, and resolves them with minimal failovers. This is equally desirable for any other type of constraint violation.

Resource fragmentation: Similarly, resource fragmentation can be a cause of failovers when "large" replicas need to be placed. Consider the example of two classes of tenants; class A, which consumes 25% of a node's capacity, and class B, which 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, placing a tenant of class B will require an additional failover.

Scoring Functions in Enterprise Cluster Managers: For example, the scoring function used in Service Fabric has three components: (i) the (weighted [11]) number of failovers required to reach the target configuration, (ii) the imbalance in resource demands across nodes, and (iii) a penalty term that is assessed when insufficiently many nodes with sufficient unused capacity exist [2].

Given the above, a key challenge is to combine these vastly different components into a single score, especially given that they all use different units (failovers, resources and penalties), and to reason about them during configuration enumeration (see Section 2.1).

Proposed Scoring Function: In contrast, we propose a scoring function that is the sum of terms that are *all* specified in terms of (expected) failovers; this makes the score more easily interpretable and allows effective pruning of the search space. This scoring function has three components, which we define in detail in Section 5:

moves: the actual number of failovers required to reach the target configuration from the current one.

$moves_{cap}$: The *expected* number of failovers required as a result of capacity violations.

$moves_{frag}$: The *expected* number of failovers for placing a full-node tenant into a randomly chosen fault domain and upgrade domain.

Formulating the score as a sum enables the *branch-and-bound* optimizations [51] discussed in Section 1.1, since it allows us to bound the number of replicas whose placement changes.

Weighted Failovers: There are numerous incentives to differentiate failovers for different tenants by how impactful they are on the overall service. For example, we may want to distinguish tenants by resource usage, by tenant type (e.g., Serverless vs Provisioned offerings) or distinguish between 1st- and 3rd-party users.

For this purpose, Service Fabric offers the *move cost* [11] abstraction, which exposes weights which can be associated with a tenant. The optimization task then becomes to minimize the *weighted* failover count. Assigning such weights is possible in our formulation as well. For clarity of exposition, we first describe a version of our formulation that treats all failovers as equal; and subsequently, in Appendix B, extend it to incorporate weights.

5 OPTIMIZATION FORMULATION

Notation: We consider a cluster to be a set of nodes $\mathcal{N}$, each of which offers a set of *resources* $\mathcal{R}$. For each resource $r \in \mathcal{R}$, every node has *capacity* c_r. The set of nodes is partitioned into *df fault domains* [5], where sets of nodes sharing a common point of failure are grouped in the same domain, and *du upgrade domains* [5], with nodes being upgraded in parallel grouped into one upgrade domain. We denote the set of nodes in fault domain d as $\mathcal{F}_d \subseteq \mathcal{N}$. Similarly, we denote the set of nodes in one *upgrade domain* d as $\mathcal{U}_d \subseteq \mathcal{N}$. We use $D_f = \{1, \ldots, df\}$ to enumerate the fault domains, and $D_u = \{1, \ldots, du\}$ to enumerate the upgrade domains. Finally, we denote the set of all nodes with a current resource violation as $\mathcal{N}_V$.

Database tenants: we consider the set $\mathcal{T}$ of database tenants; each tenant has 1 or more replicas. For a tenant $t \in \mathcal{T}$, we denote the set of replicas $\mathcal{K}_t \subset \mathbb{N}$, with replica $1 \in \mathcal{K}_t$ denoting the *primary* replica and all other *secondary* replicas. We use $\mathcal{K}_t^{\bar{V}}$ to specifically refer to all replicas not placed on nodes with a current resource violation. We include new replicas that have not yet been placed on the cluster, but are to be placed during the next invocation of the optimization in $\mathcal{T}$; we use the notation $\mathcal{T}^{new}$ to refer to these new tenants specifically. For every tenant $t \in \mathcal{T}$ not in $\mathcal{T}^{new}$, we encode the current placement replica $k \in \mathcal{K}_t$ as $p^{t,k} \in \mathcal{N}$.

We denote the (current) demand for a resource r by the k-th replica of tenant t as $d_r^{t,k}$. In addition, each tenant replica has a *maximum* demand $d\_max_r^{t,k}$, which is the maximum of resource r the replica can use. Based on these, we capture the degree a resource is over-subscribed as the *oversubscription-ratio* of a node n for a resource r as $OR_n^r := \left(\sum_{t \in \mathcal{T}} \sum_{k \in \mathcal{K}_t : p^{t,k} = n} d\_max_r^{t,k} \right) / c_r$ and for the cluster as OR^r, corresponding to the average of OR_n^r over all nodes.

Target Configuration: The output of the optimization is encoded in the form of binary decision variables $x_n^{t,k}$, which are set to 1, if the k-th replica of tenant $t \in \mathcal{T}$ should reside on node $n \in N$ in the target configuration (and 0 otherwise). The set of all decision variables is shown in Table 1 and they will be explained subsequently.

Furthermore, our formulation uses 2 constants: M, which is an upper bound number of tenants on a single node, and C, which is an upper bound on the total resource demand on a node.

This gives us the optimization objective *Score* defined below, which corresponds to the sum of failovers described in Section 4 and is minimized subject to constraints on node capacity (Equation (1)), upgrade and fault domains ((3) and (4)), and a constraint ensuring that every replica is assigned to one node only (2).

Optimization Formulation

$$Score = \min \left(moves + w_f \cdot moves_{frag} + w_c \cdot moves_{cap} \right)$$

such that,

$$\sum_{t \in \mathcal{T}} \sum_{k \in \mathcal{K}_t} d_r^{t,k} x_n^{t,k} \leq c_r \quad \forall n \in \mathcal{N}, r \in \mathcal{R} \qquad (1)$$

$$\sum_{n \in N} x_n^{t,k} = 1 \quad \forall t \in \mathcal{T}, k \in \mathcal{K}_t \qquad (2)$$

$$\sum_{n \in U_d} \sum_{k \in \mathcal{K}_t} x_n^{t,k} \leq 1 \quad \forall t \in \mathcal{T}, d \in D_u \qquad (3)$$

$$\sum_{n \in F_d} \sum_{k \in \mathcal{K}_t} x_n^{t,k} \leq 1 \quad \forall t \in \mathcal{T}, d \in D_f \qquad (4)$$

Constraints 1-4 ensure that no node in the target configuration is in resource violation (constraint (1)), each replica is placed on exactly 1 node (constraint (2)), and there is at most 1 replica per upgrade and fault domain (constraints (3)-(4)). This also implicitly ensures anti-affinity.

Table 1: Decision Variables used in the Optimization Formulation

$x_n^{t,k}$	Binary	1 if the kth replica of tenant $t \in \mathcal{T}$ should reside on node $n \in N$ in target configuration
π_n^l	Binary	1 if for Monte-Carlo iteration $l \in L$, the node $n \in \mathcal{N}$ exceeds capacity for a resource at some time
δ_d	Integer	The minimum number of tenants on a node for upgrade domain $d \in D_u$
$\delta_{d,n}$	Binary	1 if node $n \in U_d$, $d \in D_u$ has the smallest number of placed tenants
γ_d	Integer	The minimum number of tenants on a node for fault domain $d \in D_f$
$\gamma_{d,n}$	Binary	1 if node $n \in F_d$, $d \in D_f$ has the smallest number of placed tenants

Components of the scoring function: The first component of the scoring function, *moves*, quantifies the number of failovers required to reach the target configuration. For this, we count the number of replicas whose assignment in the target configuration is different from the current one (Equation (5)). We do not count new tenants entering the cluster in this equation.

Number of Failovers needed for target configuration

$$moves := \sum_{t \in \mathcal{T} \setminus \mathcal{T}^{new}} \sum_{k \in \mathcal{K}_t} \left(1 - x_{p^{t,k}}^{t,k}\right) \tag{5}$$

Here, the term $x_{p^{t,k}}^{t,k}$ is equal to 1 if the replica k was not moved between current and target configuration, and 0 otherwise. Summing $1 - x_{p^{t,k}}^{t,k}$ over all replicas therefore corresponds to the number of failed-over replicas.
Note that this above formula is simplified in that it treats a primary/secondary swap as two separate failovers. To model swaps separately, we can compute swaps as $swaps := \sum_{t \in \mathcal{T} \setminus \mathcal{T}^{new}} \sum_{k \in \mathcal{K}_t \setminus \{1\}} x_{p^{t,1}}^{t,k}$ and adjust the *moves* value by subtracting $2 \cdot swaps$.

Fragmentation failovers: Resource fragmentation can lead to failovers when "large" replicas need to be placed or moved. It is possible to address this is by measuring the amount of resource fragmentation directly as part of the scoring function, or requiring some nodes to have low or no demand. The drawback of this type of modeling is that it does not quantify the number of failovers that may result from fragmentation when placing/moving "large" replicas. Moreover, when a large replica is placed, we do not know upfront in which fault/upgrade domain it will be placed; hence, ensuring that *some* nodes have resources available may not prevent failovers.

Instead, we compute the minimum number of failovers required to place a replica requiring the *entire* resource capacity of a node (i.e., assuming the worst case) into each fault domain; we then use the average over all fault domains in scoring. We compute the same average over all upgrade domains, and use the sum of both averages as the $moves_{frag}$ component of the scoring function (Equation (12)).

The optimization criterion itself ensures that space required for this type of placement (for a domain) is only given up, if this prevents an (expected) failover of a different type. In short, the $moves_{frag}$ term ensures the cluster is maintained in a state that allows for the placement of large replicas in arbitrary domains as long this does not entail additional failovers due to other causes.

Modeling Failovers due to Fragmentation

$\forall d \in D_u,\ n \in \mathcal{U}_d$:

$$\delta_d \leq \sum_{t \in \mathcal{T}} \sum_{k \in \mathcal{K}_t} x_n^{t,k} \tag{6}$$

$$\delta_d \geq \sum_{t \in \mathcal{T}} \sum_{k \in \mathcal{K}_t} x_n^{t,k} - M\left(1 - \delta_{d,n}\right) \tag{7}$$

$\forall d \in D_f,\ n \in \mathcal{F}_d$:

$$\gamma_d \leq \sum_{t \in \mathcal{T}} \sum_{k \in \mathcal{K}_t} x_n^{t,k} \tag{8}$$

$$\gamma_d \geq \sum_{t \in \mathcal{T}} \sum_{k \in \mathcal{K}_t} x_n^{t,k} - M\left(1 - \gamma_{d,n}\right) \tag{9}$$

$$1 = \sum_{n \in U_d} \delta_{d,n} \qquad \forall d \in D_u \tag{10}$$

$$1 = \sum_{n \in F_d} \gamma_{d,n} \qquad \forall d \in D_f \tag{11}$$

$$moves_{frag} := \sum_{d \in D_u} \frac{\delta_d}{|D_u|} + \sum_{d \in D_f} \frac{\gamma_d}{|D_f|} \tag{12}$$

The terms δ_d and γ_d correspond to the minimum number of replicas in the corresponding upgrade and fault domains, respectively, with the indicator variables $\delta_{d,n}$ and $\gamma_{d,n}$ being set to 1 if n corresponds to the node that has this minimum number of replicas (and 0 otherwise). Equation (6) ensures that δ_d is never larger than the number of tenants on a node in upgrade domain d, whereas Equation (7) ensures that it is at least as large as the minimum. Equations (8)-(9) serve the same roles for fault domains.

Capacity Failovers: To model the *expected number of failovers* due to (future) capacity violations, we leverage the technique from [31]. The key idea is shown in Figure 8: the future resource demand of a set of co-located replicas is simulated using the resource demands of *similar* replicas that were observed previously. These demands are aggregated for time points $O = \{o_1, o_2, \ldots\}$, and tested for resource violations. This computation is repeated, with the past resource demands used chosen at random from all collected *similar* tenants, yielding a Monte-Carlo (MC) simulation. The probability of a resource violation is then estimated as the fraction of iterations that had one (or more) resource violations.

There are 3 key reasons why we use the MC simulation to quantify the risk of a future failover: (a) the approach effectively models the uncertainty of future demands (as opposed to giving a point

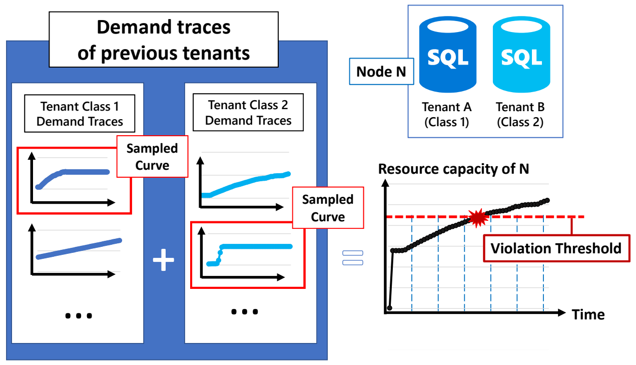

Figure 8: One MC iteration of the estimation of the probability of violation, for a node N containing 2 tenants (of class 1 and class 2).

estimate only), which is crucial for new tenants for which there is little information to base demand predictions on, (b) it implicitly captures correlations across resources and (c) it does not make assumptions about the shape of the demand curves. While the probability estimates are heuristic in nature, this is a consequence of tenant placement being an *online* problem, in which solution quality is dependent on (unknown) future resource demands. More principled characterizations of placement quality have been attempted in *online vector packing* [13, 41], which offer (loose) bounds on the number of failovers required in response tenant arrivals or load changes, but do *not* asses the likelihood of such events, either.

To realize the MC simulation as part of optimization, Equation (13) realizes a single MC iteration l (for a given node $n \in \mathcal{N}$ and time point $o \in \mathcal{O}$), resulting in an indicator variable π_n^l being set if there is a violation. Based on the indicators, the expected number of resulting failovers $moves_{cap}$ is the sum of the violation probabilities (Equation (14)). Similar to [31], we make the assumption that each violation can be resolved using one failover.

Modeling Failovers due to Capacity Violations

$$\forall n \in \mathcal{N}, o \in \mathcal{O}, l \in L, r \in \mathcal{R}:$$

$$\sum_{t \in \mathcal{T}} \sum_{k \in \mathcal{K}_t} d_{r,o,l}^{t,k} x_n^{t,k} \leq c_r + C\pi_n^l \tag{13}$$

$$moves_{cap} := \frac{1}{|L|} \sum_{n \in \mathcal{N}} \sum_{l \in L} \pi_n^l \tag{14}$$

In Equation (13), the term $d_{r,o,l}^{t,k}$ corresponds to the future resource demand of resource r for replica k of tenant t, at time o, with the past resource demand trace selected during the l-th Monte-Carlo invocation. For each node n, all of these demand terms are added up, and if the aggregate (predicted) demand exceeds the node capacity c_r, the indicator variable π_n^l is set to 1 (0 otherwise).

5.1 Scheduling failovers

When transitioning a cluster to a new configuration we need to ensure that all failovers can be executed without introducing temporary capacity violations or acerbating existing ones. For this, it is not sufficient to ensure that the new configuration has no violations (i.e., constraint (1)): when moving a replica, locally stored data needs to be copied to the target node, and memory and CPU demand for the database process are incurred even before the new replica becomes active. Therefore, we need to ensure that (in the worst case) the resource demand for moved replica can be accommodated on *both* the source and the target node during the transition.

There is an inherent trade-off here: assuming sufficient network bandwidth, scheduling failovers in parallel allows for them to complete quickly, but also ties up resources on more nodes. Moreover, in some cases, it becomes necessary to schedule failovers in multiple *phases*: consider an example cluster in which every node contains replicas that require 30% of its resource capacity. Now, placing a new replica that requires 80% of a node's capacity means that *first* some replica(s) need to be moved to create space, and only *after* these moves can the placement of the large replica proceed.

Consequently, we consider both 1-phase and 2-phase failover schedules, where all failovers in a *phase* are executed in parallel, with the next phase starting after all failovers have completed. In 2-phase schedules, during phase 1 replica moves are executed that create sufficient space (a) for new tenants and (b) to allow subsequent (i.e., during phase 2) moves of replicas located on nodes in violation. As explained above, we account for the demand of failed over replicas on both the source and target nodes, and require all nodes not in violation at the start of phase 1 to satisfy resource capacity constraints at the end of it. For nodes initially violating capacity constraints, we do not allow any tenant moves onto these nodes, in order to not exacerbate the violation further.

To realize this logic, we formulate an additional constraint ensuring that the cluster state after the 1st phase does not introduce any additional capacity violations (constraint (15)), and disallow placing tenants on nodes in capacity violations (constraint (16)).

Scheduling Failovers

$$\forall n \in \mathcal{N} \setminus \mathcal{N}_V, r \in \mathcal{R}:$$

$$\sum_{t \in \mathcal{T} \setminus \mathcal{T}^{new}} \sum_{k \in \mathcal{K}_t^{\bar{V}}} \left(d_r^{t,k} \left(x_n^{t,k} + I[n = p^{t,k}](1 - x_n^{t,k}) \right) \right) \leq c_r \tag{15}$$

$$\forall n \in \mathcal{N}_V, \forall t \in \mathcal{T}, k \in \mathcal{K}_t, \text{ with } p^{k,t} \neq n : x_n^{t,k} = 0 \tag{16}$$

Here, the left side of (15) sums up the resource demands of all replicas, counting the demand of failed-over replicas on both source and destination. This condition then ensures that there are no additional constraint violations introduced at the end of phase 1. Constraint (16) disallows adding replicas to nodes that are in resource violation.

6 SERVICE FABRIC INTEGRATION

Background: In Service Fabric clusters, each Azure SQL database runs as a stateful service and is scheduled by the *Reliability Subsystem* [10], which consists of (a) the *Failover Manager* (FM), which keeps track of database creation/deletion requests and the latest cluster configuration and periodically reports them to (b) the *Placement & Load Balancing* component (PLB). PLB detects if the configuration is invalid or requires new placement/removal of replicas. It then executes a search to find a new valid cluster configuration, generates replica moves and sends these FM for execution.

Invoking the solver: The input to PLB includes database creation/-deletion requests and the cluster configuration. PLB's state already contains all the data needed in the optimization formulation except for the MC samples $d^{t,k}_{r,o,l}$ required by Equation (13). Now, similar to the resource usage of a tenant, its MC samples can change over time. So we associate the MC samples with each tenant as its *custom metrics* [3], which are reported periodically. In total, we create an additional $|\mathcal{R}| \times |L| \times |O|$ metrics cluster-wise. Using the settings in our experiments (Section 7.2), this means $3 \times 10 \times 6 = 180$ additional metrics stored per tenant, which is less than 1KB in memory.

The *constraint constructor* and *objective constructor* consume the input and together construct the full optimization model. Specifically, the constraint constructor covers Constraints (1)–(4) and (15)–(16), while the objective constructor covers the remainder. Logically, the constraint constructor is abstracted to construct *user-facing* constraints, such as tenant affinity, while objective constructor builds the *internal* failover objective and constraints. The decoupled design aims to improve code clarity and maintainability.

For example, we observed that the majority of the scheduler invocations result new tenant placements only, with no failovers of existing tenants. This leads to room for optimization in the objective constructor without affecting the constraint constructor. Specifically, when there is no failover, the left side of Equation (13) is upper bounded by the sum of $d^{t,k}_{r,o,l}$ for all existing replicas on node n plus the largest value of $d^{t',k}_{r,o,l}$ for the newly placed tenant t'. If this upper bound is smaller than c_r for all offsets o, then we know $\pi^l_n = 0$ and the corresponding instance of Equation (13) can be omitted from the encoding. In experiments, this optimization reduced the number of constraints generated by Equation (13) by up to 90%.

The MIP solver solves the resulting optimization problem and is bounded by a configurable time limit (e.g., 2 sec.) and an approximation limit (e.g., stop when within 2% of optimal) for high efficiency. Then the FM move generator compares all optimized placement variables to the input configuration to determine which replica moves are needed. All such moves are sent to FM for execution.

7 EXPERIMENTS

In this section, we compare the proposed approach to existing industrial cluster managers as well as [31]. Please note that [31] already compared its approach to many simpler placement heuristics (such as *BestFit*, *WorstFit*, etc.) used in cluster managers, as well as [41], [20], [38] and outperformed them significantly in terms of capacity violations and failovers. Moreover, none of these approaches directly supports the placement constraints required for industrial DBaaS clusters, or has logic to resolve constraint violations, so we omit a comparison to them.

7.1 Experiment: Finding valid configurations

In this experiment, we evaluate the ability of different cluster managers (Service Fabric, Kubernetes, and our approach) to identify valid configurations for small, but challenging placement tasks. We use the *Gurobi* solver [21] in our approach.

Experimental setup: We consider a 10 node cluster with a single resource r and node capacity $c_r = 100$. Now, we generate a set of 55 single-replica tenants $\mathcal{T}$; to set their resource demands $d^{t,1}_r$, we first chose a value F ($\leq c_r$), which denotes the average per-node resource demand once all tenants in $\mathcal{T}$ are placed. Then we generate tenant demands as follows: 1 tenant with demand F, 2 tenants with demand $F/2, \ldots$, and 10 tenants with demand $F/10$. Obviously, these tenants can be placed in the cluster without capacity violations by placing the 1st tenant on node 1, the next 2 tenants on node 2, etc…

We then shuffle the set of tenants randomly and place them in the shuffled order on the cluster, testing if the cluster manager is able to generate a valid configuration after every placement request. Since we only want to exercise the configuration search component, we make the simplifying assumption that all tenant moves required to place a large tenant are instantaneous (i.e., the resource demand does *not* accumulate on both source and target nodes during a move). In particular, this means that a final configuration placing *all* tenants is always reachable, no matter where tenants had been placed previously. In the experiments, we vary F between 90 and 99 (in steps of 1) and repeat the experiment 10× for each value of F.

Cluster Manager Configuration: We initially set the *PlacementSearchTimeout* in Service Fabric (see [7]) and the *Timeout* of Gurobi to 2 seconds (both execute in a single thread in these experiments).

Results: Service Fabric is able to place all tenants in 71% of all experiments, whereas the solver is able to do so in 100% of all experiments. If we increase the *PlacementSearchTimeout* for Service Fabric by 5x to 10 seconds, Service Fabric is still only able to place all tenants in 80% of all experiments; increasing the *PlacementSearchTimeout* to 1 minute results in Service Fabric placing all tenants in 91% of the experiments. In contrast, if we reduce the *Timeout* value of Gurobi to 0.5 sec., the solver continues to place all tenants 100% of the time.

Kubernetes: We repeated the same experimental setup for the Kubernetes cluster manager. Here, we deploy a local 10-node Kubernetes (V1.25) cluster on a Windows Server 2019 machine using minikube (V1.28). The Kubernetes scheduler component, *kube-scheduler*, is deployed with the widely used *NodeResourceFit* (V1Beta2) plugin which supports three scoring strategies: *Most Allocated*, *Least Allocated*, and *RequestToCapacityRatio*. As there is only one metric r in the experiments, RequestToCapacityRatio reduces to either Most Allocated or Least Allocated depending on the pre-defined scores of node utilization ratios. Therefore, we only evaluate the Most Allocated and Least Allocated scoring strategies in the experiments.

Placement is carried out by deploying a dummy nginx pod with the specified resource demand. An *extended resource* [27] is created to represent the demand for r. The resource demand is set accordingly in the 'requests' and 'limits' of the pod YAML file.

Kubernetes handles placement failure differently than Service Fabric: if a pod (of higher priority) can not be placed, an existing lower-priority pod is preempted and placed into the scheduling queue [28]. Thus, finding a valid configuration may involve multiple iterations of preemption and re-scheduling and hence we cannot apply a timeout threshold (as before) to asses if a placement fails.

Instead, each tenant is (initially) placed with *low* priority. If the initial placement of a tenant t fails, its priority level is temporarily increased to *high* so that Kubernetes may preempt existing tenants to place t. Once placed, t's priority is restored to *low*. All preempted tenants are rescheduled for placement. This may trigger cascading preemptions, which are not guaranteed to converge to a valid cluster state. Consequently, we set a stopping criterion: if the number of tenants preempted due to a single new placement exceeds half of the existing tenants in a cluster, we consider the placement failed.

Repeating the experimental setup above, we see that rate of successfully placing all tenants depends significantly on the scoring strategy: for *Most Allocated*, Kubernetes was able to place all tenants 44% of the time, and for *Least Allocated* only 17%. Both policies consistently result in placement failures if F exceeds 97.

7.2 Experiment: Cluster Simulation

In this section, we consider realistic tenant (demand) distributions and cluster setups and evaluate the effect of the different techniques on the overall quality of the DBaaS service, measured via the number of violations, failovers as well as the overhead of tenant placement itself. Our experiments cover a period of 3 (simulated) weeks of cluster time, and we stress the various techniques by systematically varying the degree to which resources are over-subscribed. Because Kubernetes handles violations and placement failures by iterative re-scheduling tenant using priorities (as discussed in Section 7.1) thereby making it difficult to minimize failovers, we focus these experiments on comparing to Service Fabric.

Experimental setup: Due to the scope (40 nodes) and duration (21 days) of the experiments required for one data point, executing them on real hardware is prohibitively expensive. Instead, 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. Because the simulation covers all relevant input interfaces to PLB (including all SF configuration settings [7]), 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 runs on a Windows Server 2019 machine with Intel Xeon E5-2660 v3 CPU, 384 GB memory and 745 GB SSD. The use of simulators in cluster placement is common in research, e.g., papers on the *Omega* scheduler [43], the *Borg* cluster manager [49], or learning-based cluster scheduling [34] all evaluate based on (trace-driven) simulations.

For the simulated clusters, we use the Azure SQL DB *Gen5* hardware specs [6]. The clusters contain 40 nodes, divided into 10 upgrade domains and 5 fault domains (using the logic in [5]).

Compared Techniques: We compare Service Fabric's existing *Placement-and-Load-Balancing* component (*PLB*), the *Probability-of-Violation* estimator [31] integrated with PLB's scoring function (*PLB+PrV*), which is the best-performing combination in the cluster experiments in [31], as well as the optimization formulation described in Section 5 (*MIP*). As before, we run the Gurobi solver with a single thread, since Service Fabric's PLB component is single-threaded. To give comparable processing time to each approach, we set a timeout of 2 seconds for the configuration optimization itself (using *Constraintchecktimeout* and *PlacementSearchTimeout* in Service Fabric and *TimeLimit* for Gurobi); we also allow Gurobi to exit the optimization if the current best score is within 5% of the lower bound computed by Gurobi. Finally, to demonstrate that our approach can achieve significant improvements independently of the probability of violation estimates of [31], we also evaluate a version of our approach that does *not* include the $moves_{cap}$ term defined in (14), which is based on the violation probabilities. We refer to this approach as *MIP w/o PrV*.

Demand traces: To simulate the resource demands of DBaaS tenants, we use two distinct samples of resource usage traces from Azure SQL DB, containing several million distinct tenants; both samples were taken in different geographical regions, which differ significantly in terms of the distribution of resource demands and tenant sizes. Each trace contains the resource usage at 10-minute granularity, for 3 resources: *CPU*, *main memory* and *local disk* usage, all of which are used in the following experiments. We hold out 115K of these traces (sampled at random) for the Monte Carlo simulations used to estimate the probability of violations, requiring 7.19 MB of storage after *trace compression* (see [31]).

Optimization Formulation: The placement algorithm described in Section 5 uses the weights $w_f = 0.05$ (for failovers due to fragmentation) and $w_v = 0.1$ (for capacity failovers) in the computation of the objective function *Score*. We use $|L| = 10$ iterations of the Monte Carlo simulations and 5 time points $O = \{5, 30, 60, 1000, 5000\}$ (with each value corresponding to a number of minutes into the future). Finally, we set C to 10× the node capacity, and M to the total number of replicas in the cluster.

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 technique placing the same sequence of tenants. In this experiment, we assign identical weight to all tenants – weighted failovers are considered in Section 7.3. All techniques use the same set of constraints.

Results – Service Quality (Region 1): Figure 9 shows total number of violations seen, as we vary the cluster-level over-subscription ratio OR^r from 1.8 to 2.6 (using CPU cores as r). Here, the use of *probability of violation* estimates significantly reduces the number of violations seen. However, *MIP* reduces the number of violations even further, especially at large OR^r, where *PLB+PrV* has more than twice the violations of *MIP*. Moreover, *MIP w/o PrV* performs almost on par with *MIP*. This demonstrates that the improvements seen from *MIP* are not mainly due to the *probability of violation* estimates, but the modified scoring function and configuration search.

We observe the same trends when comparing the numbers of failovers, with the improvements due to *MIP* being more pronounced: whereas for *MIP* the number of failovers and violations are nearly

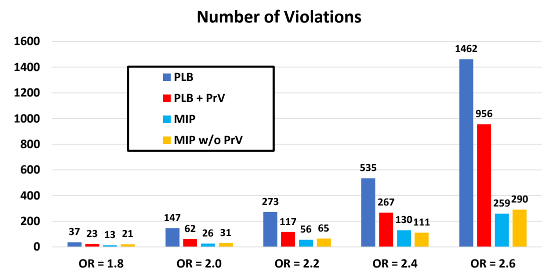

Figure 9: No. Violations during the Simulation (Region 1)

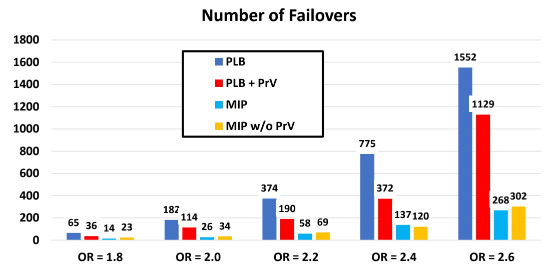

Figure 10: No. Failovers during the Simulation (Region 1)

identical in every experiment (thus validating the assumption made when computing the expected number of failovers $moves_{cap}$ in Section 5), both competing techniques, which use the PLB configuration enumeration, exhibit significantly more failovers than violations. This is likely due to the effects described in Section 2.2.

Figure 11 shows the number of instances for which the cluster manager is *not* able to generate a valid target configuration in response to a resource violation. As we can see, both Service Fabric as well as our proposed approach (almost) never run into this issue unless the OR is 2.6 (*PLB* has 2 such instances, *PLB+PrV* 1, and MIP none for $OR < 2.6$). However, at $OR = 2.6$, the fact that our approach (a) can iterate through the search space of configurations in a principled manner and (b) is able to use intermediate results to prune the search space result gives an improvement in the number instances for which no valid configuration could be found by more than 30x over the PLB-based searches. Even if this level of over-subscription is unlikely to be used in production clusters, the results demonstrate that the solver-based placement component is able to handle emergency situations in which (e.g., due to correlated node failures) cluster capacity is significantly reduced. We did not call out the variant of MIP without PrV estimates in Figure 11, as these do not affect the solver's ability to generate a valid configuration.

Results – Service Quality (Region 2): We repeated the experiment for region 2, with one exception: the over-subscription ratio was only increased up to 2.4, as tenants from this region consume more resources on average, leading to instances where the cluster capacity was insufficient to accommodate all tenants at $OR = 2.6$. This also results in more resource violations, compared to region 1.

Otherwise, the overall trends seen in these experiments are similar to the ones observed in region 1; MIP significantly reduces the

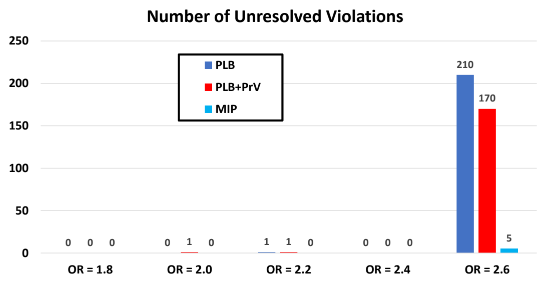

Figure 11: Instances without a valid configuration (Region 1)

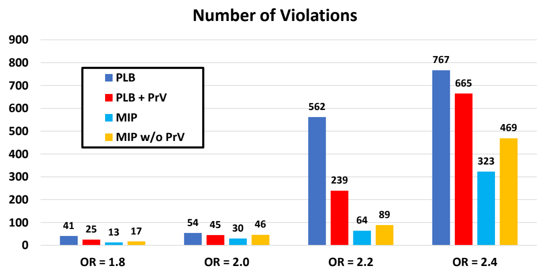

Figure 12: No. Violations during the Simulation (Region 2)

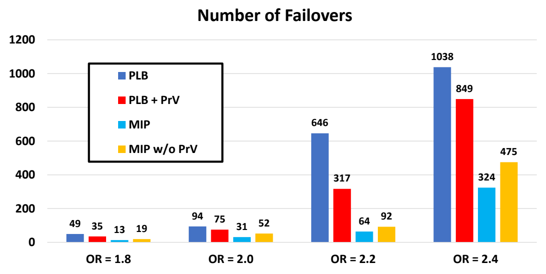

Figure 13: No. Failovers during the Simulation (Region 2)

incidence of violations compared to the alternatives and exhibits an even larger reduction in the number of failovers.

Results – Overheads: For each invocation of the placement and constraint violation fixing logic, we measured the latency from the invocation to the output of the target configuration. We show the measurements in Figure 14, showing the 50th, 90th and 99th percentile for each algorithm; we are showing only the measurements for region 1, as the values for region 2 are very similar.

Interestingly, the latency for *MIP* is significantly less than for *PLB* and *PLB+Prv*, despite the fact that (a) all approaches use identical timeouts, and (b) while PLB-based approaches incrementally maintain all data structures inside PLB as replicas report new resource demands, *MIP* has to encode the cluster state from scratch for every invocation. The main reason for this improvement is that *MIP* can stop the optimization when sufficiently close to the optimal objective value, which is something generally not possible for *PLB* and *PLB+Prv*, for the reasons described in Section 2.1.

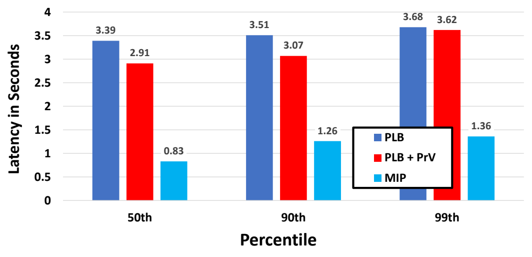

Figure 14: Placement Latency for OR=1.8 (Region 1)

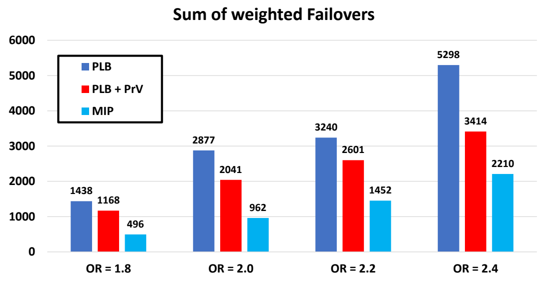

Figure 15: Sum of weighted Failovers (Region 1)

7.3 Experiment: Weighted Failovers

One important consideration is to differentiate failovers by how impactful they are for the service. For this, we repeated the experiment of Section 7.2, but assigned different weights to tenants, and used the modified formulation from Appendix B. To compare to the original PLB, we used the weight assignment logic used in production in Azure SQL DB, and limited the number of distinct weights to 3, as Service Fabric currently supports only 3 weight classes (Low/Medium/High, which correspond to the weights 1/15/40).

Here, we compare the sum of the *weighted* failover counts, as these are the metric each approach seeks to optimize. In addition, we measured the aggregate main memory consumption of all failed-over tenants (as a proxy for the costs to re-hydrate the corresponding caches) as well as aggregate local disk space (as a a proxy for the time required for a move to complete). The results can be seen in Figures 15, 16, and 17. As we can see, MIP continues to outperform the alternative approaches, on *all* of the considered measures.

7.4 Experiment: Scalability

Next, we measure the scalability of our approach as we scale up the cluster sizes, repeating the setup (for $OB = 1.8$) of Section 7.2 and measuring the (a) placement latency and (b) the peak memory overhead of the solver (including both solver code and working memory). The results are shown in Figures 18 and 19.

As we can see, the latency numbers increase with cluster size (which means scaling up the number of tenants as well), but they do so *sub-linearly*. In fact, given the 2 sec. timeouts used for both cluster managers, *MIP* on a 200 node cluster still has lower latencies than *PLB* on 40 nodes. Only at large cluster sizes does MIP exceed the 2 sec. timeout, due to the encoding of the cluster state becoming more

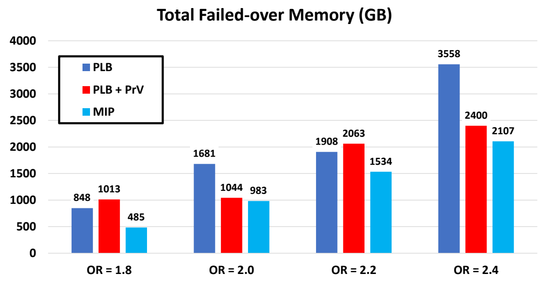

Figure 16: Total Memory failed over (Region 1)

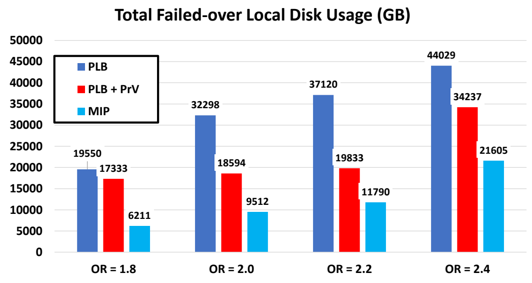

Figure 17: Total Disk-resident Data failed over (Region 1)

expensive. Note that this overhead is required independently of any timeout settings. An incremental encoding scheme modifying a previous optimization task is something we are investigating as future work. Memory overhead (as shown in Figure 19) is not a significant concern for the practicality of our approach.

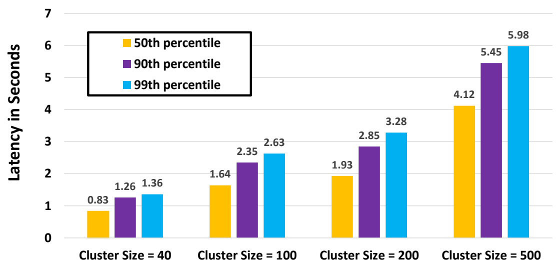

Figure 18: Placement latency (*MIP*) for larger clusters

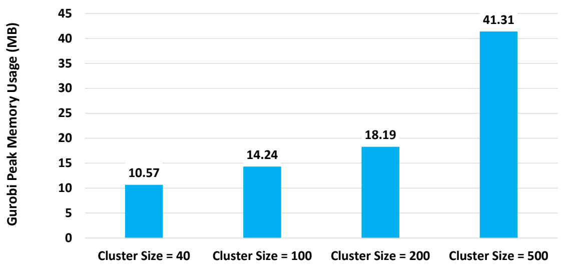

Figure 19: Peak Memory Requirements for Optimization

8 CONCLUSION

In this paper, we show how to leverage solvers in DBaaS resource management. Our contributions include the formulation of resource management as an optimization problem in which (a) constraints and objective are realized as linear combinations of (binary) decision variables and (b) the objective captures the key causes of service disruption, while enabling effective pruning of the search space.

We implemented the approach in the Service Fabric codebase. Using realistic (simulated) cluster setups and tenant distributions, we show significant improvements in violations/failovers when compared to existing cluster managers. We also demonstrated improvements in the ability to find valid placements and resolve violations. Our approach has a small resource footprint, requiring 1 thread and $\leq$ 42MB memory. Our work also substantially improves upon [31], which targeted the same scenario, and itself had been shown to outperform many simpler heuristics used in enterprise cluster managers, and techniques proposed in research ([41], [20], [38]).

A ENCODING ADDITIONAL CONSTRAINTS

Here, we outline how to extend the formulation to the other constraints supported by Service Fabric:

Node Block List and Placement Constraints These constraints can be enforced by setting $x_n^{t,k} = 0$ for all nodes n that are either *blocked* or do not satisfy all *placement constraints* [46].

Affinity (aligned) To satisfy this constraint, the primary replicas of two services t_1, t_2 need to be co-located on a node. Assuming the primaries are always the 1st replica, this can be encoded as $\forall n \in \mathcal{N} : x_n^{t_1,1} = x_n^{t_2,1}$

Affinity (non-aligned) For this constraint to hold, replicas of a *child service* t_1 must be co-located with replicas of a parent Service t_2, with the replica role (i.e., *primary* vs *secondary*) being irrelevant. For this, we define the number of child replicas on a node n as $SR_n = \sum_{k\in\mathcal{K}_{t_1}} x_n^{t_1,k}$, and the number of parent replicas as $PR_n = \sum_{k\in\mathcal{K}_{t_2}} x_n^{t_2,k}$ and enforce the constraint $\forall n \in \mathcal{N} : SR_n \leq PR_n$.

Scale Out Constraints These constraints enforce a minimum or maximum on the total number of nodes containing replicas for a given tenant t. We can encode this constraint using additional (binary) decision variables μ_n, which encode if an instance of t is placed on a node n, and are set using the additional constraints:

$$\forall n \in \mathcal{N} : \textstyle\sum_{k\in\mathcal{K}_t} x_n^{t,k} \quad \leq M\mu_n$$
$$\forall n \in \mathcal{N} : \textstyle\sum_{k\in\mathcal{K}_t} x_n^{t,k} \quad \geq \mu_n$$

The Scale Out constraints can then be expressed as $\sum_{n\in\mathcal{N}} \mu_n \leq$ *max_ScaleOut* and $\sum_{n\in\mathcal{N}} \mu_n \geq$ *min_ScaleOut*.

Preferred Location Constraint Unlike the other constraints, this is a *soft* constraint, meaning it is used in the scoring function that is minimized as part of configuration selection. This constraint is used to ensure that – during a cluster upgrade – primary replicas on nodes about to be upgraded are *swapped* with secondary replicas located on nodes that have been upgraded already [45], meaning no further swaps will be required. Denoting the set of upgraded nodes as $\mathcal{N}_U$, we can specify the number of such swaps as

$$U\_swaps := \left|\{t \in \mathcal{T} \mid \exists k \in \mathcal{K}_t\backslash\{1\} : x^{t,k} = p^{t,1} \wedge p^{t,1} \in \mathcal{N}_U\}\right|.$$

We can then make a weighted combination of (a) *swaps* – *U_swaps* and (b) *U_swaps* part of the optimization criterion.

B ASSIGNING WEIGHTS TO FAILOVERS

In the following, we describe how to modify the formulation of Section 5 to incorporate weighted failovers, where each tenant t is associated with a weight w_t, with the possible weights chosen from a set $\mathcal{W} = \{v_1, \ldots, v_h\} \subset \mathbb{R}$. Modifying the computation of the number of failovers needed to reach a target configuration (i.e., the value of *moves*) and expected failovers due to resource fragmentation ($moves_{frag}$) is straight-forward, as all that is required is to add the corresponding w_t terms to Equations (5)–(9); we show the modified versions (Equations (5')–(9')) below.

Computing expected failovers due to capacity violations ($moves_{cap}$) is more involved: here, we assume that – in case of a capacity violation – we resolve it by moving the replica with the lowest weight off of the target node. To model this, we, instead of using π_n^l to indicate a violation on node n in Monte-Carlo iteration l, use an indicator $\pi_{n,j}^l$, which indicates violation on node n in Monte-Carlo iteration l that is resolved by moving a tenant of weight v_j (Equation (13')). Based on these indicators, we can then compute the weighted failovers due to capacity constraints by multiplying the $\pi_{n,j}^l$ with the corresponding weights v_j (Equation (14'))

For correctness, we need to ensure that (a) at most one of the $\pi_{n,j}^l$ indicators is set (i.e., for only one value of l) and (b) that the indicator is only set if a tenant with the corresponding weight exists on the node, leading to new constraints ((17) and (18)).

Assigning Weights to Failovers

Weighted failovers to reach a new configuration:

$$moves \quad := \sum_{t\in\mathcal{T}\setminus\mathcal{T}^{new}} w_t \left(\sum_{k\in\mathcal{K}_t} \left(1 - x_{p^{t,k}}^{t,k}\right)\right) \qquad (5')$$

(Expected) weighted failovers due to fragmentation:

$$\delta_d \quad \leq \sum_{t\in\mathcal{T}} \sum_{k\in\mathcal{K}_t} w_t\, x_n^{t,k} \qquad (6')$$
$$\delta_d \quad \geq \sum_{t\in\mathcal{T}} \sum_{k\in\mathcal{K}_t} w_t\, x_n^{t,k} - M\left(1 - \delta_{d,n}\right) \qquad (7')$$
$$\gamma_d \quad \leq \sum_{t\in\mathcal{T}} \sum_{k\in\mathcal{K}_t} w_t\, x_n^{t,k} \qquad (8')$$
$$\gamma_d \quad \geq \sum_{t\in\mathcal{T}} \sum_{k\in\mathcal{K}_t} w_t\, x_n^{t,k} - M\left(1 - \gamma_{d,n}\right) \qquad (9')$$

Note that M needs to account for weights, i.e. multiplied with $\max_i v_i$.
(Expected) weighted failovers due to capacity violations:

$$\sum_{t\in\mathcal{T}} \sum_{k\in\mathcal{K}_t} d_{r,o,l}^{t,k} x_n^{t,k} \leq c_r + C\big(\textstyle\sum_{j\in\{1,\ldots|\mathcal{W}|\}} \pi_{n,j}^l\big) \qquad (13')$$
$$moves_{cap} \quad := \frac{1}{|L|} \sum_{n\in N} \sum_{l\in L} \sum_{j\in\{1,\ldots|\mathcal{W}|\}} v_j \cdot \pi_{n,j}^l \qquad (14')$$
$$\sum_{j\in\{1,\ldots|\mathcal{W}|\}} \pi_{n,j}^l \qquad \leq 1 \qquad (17)$$
$$\forall n \in N, j \in \{1, \ldots, |\mathcal{W}|\}, l \in L : \sum_{t\in\mathcal{T}:w_t=v_j} \sum_{k\in\mathcal{K}_t} x_n^{t,k} \geq \pi_{n,j}^l \quad (18)$$

REFERENCES

[1] AWS. Amazon Aurora. https://aws.amazon.com/rds/aurora/. Last accessed: 2023-09-21.

[2] Azure, M. Defragmentation of metrics and load in Service Fabric. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-defragmentation-metrics. Last accessed: 2023-09-21.

[3] Azure, M. 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: 2023-09-21.

[4] Azure, M. 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, 2020. Last accessed: 2022-07-12.

[5] Azure, M. Service Fabric Cluster Resource Manager. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-cluster-description, 2021. Last accessed: 2022-07-12.

[6] Azure, M. Azure sql database pricing. https://azure.microsoft.com/en-us/pricing/details/azure-sql-database/single/, 2022. Last accessed: 2022-07-12.

[7] Azure, M. Customize service fabric cluster settings. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-fabric-settings, 2022. Last accessed: 2022-07-12.

[8] Azure, M. Describe a Service Fabric cluster by using Cluster Resource Manager. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-cluster-description#node-properties-and-placement-constraints, 2022. Last accessed: 2023-09-21.

[9] Azure, M. Overview of Azure Service Fabric. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-overview/, 2022. Last accessed: 2022-07-12.

[10] Azure, M. Service Fabric Architecture. https://github.com/uglide/azure-content/blob/master/articles/service-fabric/service-fabric-architecture.md, 2022. Last accessed: 2022-07-12.

[11] Azure, M. Service Fabric Movement Cost. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-movement-cost, 2022. Last accessed: 2022-07-12.

[12] Barker, S., Chi, Y., Moon, H. J., Hacigümüş, H., and Shenoy, P. "Cut me some slack": Latency-Aware Live Migration for Databases. In *Proceedings of the 15th International Conference on Extending Database Technology* (2012), p. 432–443.

[13] Berndt, S., Jansen, K., and Klein, K.-M. Fully Dynamic Bin Packing Revisited. *Mathematical Programming* (2020). https://link.springer.com/article/10.1007/s10107-018-1325-x, Last accessed: 2022-07-12.

[14] BusinessWire. Global cloud database and dbaas market (2020 to 2025). https://www.businesswire.com/news/home/20200317005638/en/Global-Cloud-Database-and-DBaaS-Market-2020-to-2025, 2022. Last accessed: 2022-07-12.

[15] Das, S., Nishimura, S., Agrawal, D., and El Abbadi, A. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow. 4*, 8 (2011).

[16] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS* (2014).

[17] Elmore, A. J., Das, S., Agrawal, D., and El Abbadi, A. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (2011), p. 301–312.

[18] Github. Service fabric codebase. https://github.com/Microsoft/service-fabric/, 2022. Last accessed: 2022-07-12.

[19] Google. Google Cloud SQL. https://cloud.google.com/sql/, 2021. Last accessed: 2022-07-12.

[20] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-Resource Packing for Cluster Schedulers. In *SIGCOMM* (2014). Last accessed: 2022-07-12.

[21] Gurobi. Gurobi optimizer. https://www.gurobi.com/, 2022. Accessed: 2022-07-13.

[22] Hat, R. Red hat openshift. https://www.redhat.com/en/technologies/cloud-computing/openshift, 2022. Last accessed: 2022-07-12.

[23] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: Fair Scheduling for Distributed Computing Clusters. In *22nd Symposium on Operating Systems Principles* (2009), Association for Computing Machinery, p. 261–276.

[24] Kakivaya, G., Xun, L., Hasha, R., Ahsan, S. B., Pfleiger, T., Sinha, R., Gupta, A., Tarta, M., Fussell, M., Modi, V., Mohsin, M., Kong, R., Ahuja, A., Platon, O., Wun, A., Snider, M., Daniel, C., Mastrian, D., Li, Y., Rao, A., Kidambi, V., Wang, R., Ram, A., Shivaprakash, S., Nair, R., Warwick, A., Narasimman, B. S., Lin, M., Chen, J., Mhatre, A. B., Subbarayalu, P., Coskun, M., and Gupta, I. Service Fabric: A Distributed Platform for Building Microservices in the Cloud. In *Proceedings of the Thirteenth EuroSys Conference* (2018), EuroSys. https://dl.acm.org/doi/10.1145/3190508.3190546, Last accessed: 2022-07-12.

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

[26] Kubernetes. Assign pods to nodes using node affinity. https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/. Last accessed: 2023-09-21.

[27] Kubernetes. Assign Extended Resources to a Container. https://kubernetes.io/docs/tasks/configure-pod-container/extended-resource/, 2022. Last accessed: 2022-07-12.

[28] Kubernetes. Kubernetes Pod Priority and Preemption. https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/, Last accessed: 2023-09-21, 2022.

[29] Kubernetes. Node-pressure Eviction, 2022. https://kubernetes.io/docs/concepts/scheduling-eviction/node-pressure-eviction/#pod-selection-for-kubelet-eviction, Last accessed: 2022-07-12.

[30] Kubernetes. What is Kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, Last accessed: 2022-07-12, 2022.

[31] König, A. C., Shan, Y., Ziegler, T., Kakaraparthy, A., Lang, W., Moeller, J., Kalhan, A., and Narasayya, V. Tenant Placement in Over-subscribed Database-as-a-Service Clusters. In *48th International Conference on Very Large Databases* (September 2022). Last accessed: 2022-07-12.

[32] Lee, J. Mixed-integer nonlinear programming: Some modeling and solution issues. *IBM Journal of Research and Development 51*, 3.4 (2007), 489–497.

[33] Li, J. Y., Zhang, J., Zhou, W., Liu, Y., Zhang, S., Xue, Z., Xu, D., Fan, H., Zhou, F., and Li, F. Eigen: End-to-End Resource Optimization for Large-Scale Databases on the Cloud. *Proc. VLDB Endow. 16*, 12 (Sep 2023), 3795–3807.

[34] Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., and Alizadeh, M. Learning Scheduling Algorithms for Data Processing Clusters. In *SIGCOMM* (2019), p. 270–288.

[35] Mate, J., Daudjee, K., and Kamali, S. Robust multi-tenant server consolidation in the cloud for data analytics workloads. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (2017), pp. 2111–2118.

[36] Moon, H. J., Hacigümüş, H., Chi, Y., and Hsiung, W.-P. SWAT: A Lightweight Load Balancing Method for Multitenant Databases. In *Proceedings of the 16th International Conference on Extending Database Technology* (2013), EDBT '13.

[37] Narasayya, V., and Chaudhuri, S. Cloud Data Services: Workloads, Architectures and Multi-tenancy. *Foundations and Trends in Databases 10*, 1 (2021), 1–107. Last accessed: 2022-07-12.

[38] Panigrahy, R., Talwar, K., Uyeda, L., and Wieder, U. Heuristics for Vector Bin Packing. Tech. rep., Microsoft Research, 2011. https://www.microsoft.com/en-us/research/publication/heuristics-for-vector-bin-packing/, Last accessed: 2022-07-12.

[39] Picado, J., Lang, W., and Thayer, E. C. Survivability of Cloud Databases - Factors and Prediction. In *ACM SIGMOD* (2018), p. 811–823. https://www.microsoft.com/en-us/research/publication/survivability-of-cloud-databases-factors-and-prediction/, Last accessed: 2022-07-12.

[40] Rong, K., Budiu, M., Skiadopoulos, A., Suresh, L., and Tai, A. Scaling a Declarative Cluster Manager Architecture with Query Optimization Techniques. *Proceedings of the VLDB Endowment* (2023).

[41] S., K. Efficient Bin Packing Algorithms for Resource Provisioning in the Cloud. *ALGOCLOUD* (2015). https://link.springer.com/chapter/10.1007/978-3-319-29919-8_7, Last accessed: 2022-07-12.

[42] Schaffner, J., Januschowski, T., Kercher, M., Kraska, T., Plattner, H., Franklin, M., and Jacobs, D. RTP: Robust Tenant Placement for Elastic In-Memory Database Clusters. In *ACM SIGMOD* (2013).

[43] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), p. 351–364.

[44] SQL, A. Azure SQL Database. https://azure.microsoft.com/en-us/products/azure-sql/database/#overview, 2022. Last accessed: 2022-07-12.

[45] SQL, A. Cluster resource manager integration with Service Fabric cluster management, 2022. https://github.com/MicrosoftDocs/azure-docs/blob/main/articles/service-fabric/service-fabric-cluster-resource-manager-management-integration.md, Accessed: 2022-07-13.

[46] SQL, A. Describe a service fabric cluster by using cluster resource manager. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-cluster-description#node-properties-and-placement-constraints, 2022. Last accessed: 2023-09-21.

[47] Suresh, L., Loff, J. a., Kalim, F., Jyothi, S. A., Narodytska, N., Ryzhyk, L., Gamage, S., Oki, B., Jain, P., and Gasch, M. Building Scalable and Flexible Cluster Managers using Declarative Programming. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (USA, 2020), OSDI'20.

[48] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale Cluster Management at Google with Borg. In *EuroSys* (2015), Association for Computing Machinery.

[49] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2015).

[50] VMWare. Distributed Resource Scheduler. https://www.vmware.com/products/vsphere/drs-dpm.html, 2022. Last accessed: 2022-07-12.

[51] Wikipedia. Branch and bound. https://en.wikipedia.org/wiki/Branch_and_bound, 2022. Last accessed: 2022-07-12.

[52] Wolsey, L. *Integer Programming, Second Edition.* John Wiley & Sons, Inc., 2020.