Vertically Autoscaling Monolithic Applications with CaaSPER:

Scalable Container-as-a-Service Performance Enhanced Resizing Algorithm for the Cloud

Anna Pavlenko, Joyce Cahoon, Yiwen Zhu, Brian Kroth, Michael Nelson,
Andrew Carter, David Liao, Travis Wright, Jesús Camacho-Rodríguez, Karla Saur
{firstname}.{lastname}@microsoft.com
Microsoft

ABSTRACT

Kubernetes has emerged as a prominent open-source platform for managing cloud applications, including stateful databases. These monolithic applications rely on vertical scaling, adjusting CPU cores based on load fluctuations. However, our analysis of Kubernetes-based Database-as-a-Service (DBaaS) offerings at Microsoft revealed that many customers consistently over-provision resources for peak workloads, neglecting cost-saving opportunities through resource scale-down. We found that there is a gap in the ability of existing vertical autoscaling tools to minimize resource slack and respond promptly to throttling, leading to increased costs and impacting crucial metrics such as throughput and availability.

To address this challenge, we propose CaaSPER, a vertical autoscaling algorithm that blends reactive and proactive strategies. By dynamically adjusting CPU resources, CaaSPER minimizes resource slack, maintains optimal CPU utilization, and reduces throttling. Importantly, customers have the flexibility to prioritize either cost savings or high performance based on their preferences. Extensive testing demonstrates that CaaSPER effectively reduces throttling and keeps CPU utilization within target levels. CaaSPER is designed to be application-agnostic and platform-agnostic, with potential for extension to other applications requiring vertical autoscaling.

1 INTRODUCTION

Cloud computing [6, 27, 53] has transformed the landscape of application development, deployment, and management by providing organizations with access to on-demand resources and scalability. However, during provisioning, users are often required to specify the amount of resources they will initially require among a large number of cloud offerings (e.g., VM configuration and size). It is challenging to estimate resource requirements upfront, and the initial settings can become irrelevant with the dynamic nature of the workloads. One of the most common scaling approaches to address some of these issues has been *horizontal autoscaling*, in which additional service replicas are added and removed based on utilization, thus adjusting overall system resource usage in fixed-sized quantities. Although this has worked for some services [59], this approach is not well suited for stateful monolithic systems (e.g., traditional RDBMS) that either have a fixed number of total instances (e.g., single writable primary) or cannot quickly scale horizontally due to size of data copy operations inherent to creating new replicas. In such cases, *vertical scaling* capabilities become crucial, allowing expanding or contracting resources of existing replicas. Additional benefits to vertical scaling include simplicity, performance, and reliability [41].

Modern platforms such as Kubernetes (K8s) [45], which has become a popular platform choice for implementing Database-as-a-Service (DBaaS) and other stateful service offerings [20, 39, 54, 66], facilitate vertical resource scaling. K8s provides two essential mechanisms, `requests` and `limits`, to define *guaranteed* and *burstable* CPU resource allocation for applications. At Microsoft and elsewhere, applications needing predictable performance, such as databases [24, 37], often set `requests` and `limits` to the same value [13, 40, 48] to ensure that the application will be scheduled on a node with enough resources to always provide the `limits`.[1] Despite resizing support, we find that users in our database offerings at Microsoft rarely scale their deployments and usually over-provision for the worst-case. In fact, during a sampling of first-party DBaaS deployments, we found cases of CPU *over-provisioning* that exceeded peak load by a significant factor, up to 20x in certain workloads, resulting in under-utilized resources (i.e., increased costs for idle resources), as well as instances of *under-provisioning*, which leads to performance impacts due to "throttling" (i.e., when an application lacks enough resources to meet its load demands).

To automate this process, the Vertical Pod Autoscaler (VPA) [33] in K8s can dynamically adjust the `requests` and `limits` values according to a pluggable algorithm. However, when tested, the default VPA algorithm and other existing approaches proved inadequate in our scenario for effectively addressing cases involving throttling detection and scaling down when over-provisioned. Additionally, these methods were oblivious of the billing model in use, leading to suboptimal cost-performance tradeoffs during scaling decisions, an important consideration for customers. Other recent works in K8s [73] leverage machine learning to support predictive autoscale. However, there is a significant drawback in purely relying on a machine learning algorithm in predictive autoscaling especially using time-series forecasting, as it lacks the ability to effectively detect throttling and instead assumes that future usage will remain consistent until retrained. Moreover, for throttled workloads, the usage forecast does not align with the true amount of resources required for the workload, leading to under-estimated `limits` (see §3.3). When aiming for optimal performance, there is a strong need to quickly identify and respond to throttling to meet SLA objectives.

SIGMOD'24, June 2024, Santiago, Chile

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Note that service level agreements (SLAs) provided by DBaaS further emphasize the importance of predictability and may have penalties for violations [57, 58].

To overcome these limitations, we propose a hybrid autoscaling algorithm called CaaSPER, focused on predictable performance and enabling optimal scaling for our pay-as-you-go billing model. CaaSPER adopts a clean-slate, history-independent reactive algorithm and combines it with a predictive proactive approach based on historical time-series data to make informed decisions regarding CPU requirements. CaaSPER ensures smooth operation without causing throttling while also minimizing excessive resource slack to improve cost-efficiency. Additionally, CaaSPER offers customizable parameters, allowing users to prioritize availability, cost-efficiency, or performance based on their specific workload requirements. To aid in understanding different trade-offs and fine-tuning CaaSPER's parameters, we have also developed an accompanying simulator.

We have conducted extensive testing of CaaSPER using a diverse range of real and synthetic data, focusing on two widely used databases, Database A and Database B, and compared its performance against existing approaches. Our results demonstrate the effectiveness of CaaSPER, maintaining 90-100% of the original throughput at a reduced cost of 49%-74% in our live traces. This cost reduction is configurable based on preferences. In contrast, the other evaluated schemes exhibited higher levels of throttling, restricting throughput to as low as 27%. Moreover, as we show in our results, CaaSPER can adapt to the unique characteristics of the pay-as-you-go billing model employed by DBaaS providers by tuning its parameters accordingly, ensuring cost-effectiveness for users. While we evaluate CaaSPER specifically in the context of DBaaS, it is worth noting that CaaSPER is designed to be application-agnostic, meaning its benefits can be extrapolated to other applications with similar characteristics that require vertical autoscaling.

Contributions. This paper makes the following core contributions:

- We develop a vertical scaling process aligned with service invariants such as a pay-as-you-go billing model and performance constraints, with a focus on ensuring predictable performance and support for stateful service sets with asymmetric workload metrics (i.e., primary-secondary systems).
- We propose an algorithm that approximates the severity of throttling by using the slope function and determines the optimal capacity, significantly reducing throttling in a single step.
- We introduce an interpretable method for customers to consider trade-offs between price and performance in decision-making, enabling customization of the algorithm through configuration parameter mapping.
- We develop a simulator aimed at accelerating the evaluation and optimization of algorithms.
- We conduct experiments across real and synthetic workloads across two DBaaS offerings at Microsoft.

2 BACKGROUND

In this section, we provide an overview of the key components of K8s (§2.1), then we delve into the implementation of vertical autoscaling within K8s (§2.2).

2.1 Kubernetes Resource Management

Containers are lightweight, executable packages that simplify the deployment of applications. In K8s, containers are grouped into *pods*, which serve as the fundamental unit for scheduling on K8s

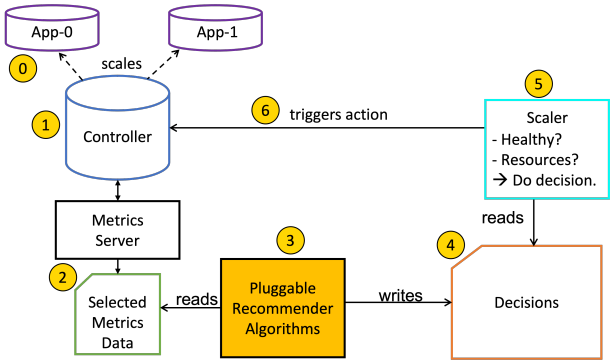


Figure 1: Generic vertical autoscaling (end-to-end).

cluster *nodes*[2]. Additionally, pods can be part of a *stateful set* for stateful applications that require *persistent volumes* for storage [3]. This ensures that a specified number of identical pod instances, referred to as *replicas*, are running at any given time. These replicas can be distributed across multiple nodes to provide high-availability (HA) [35]. To coordinate state transitions for replicas within a set, an automated controller called an *operator* is used.

The K8s *scheduler* uses `requests` specifications on various dimensions (e.g., CPU core time and memory) to define minimum guaranteed resource allocations for scheduling pods onto nodes. In contrast, `limits` are employed to prevent a single pod from monopolizing resources, thereby preserving the performance and stability of other pods on the same node [34]. Both `limits` and `requests` are specified at the container level within a pod, and they are applied to all replicas in a stateful set. Once the containers are running on the nodes, their specifications are enforced using the Linux cgroups subsystem [1]. This ensures that containers are allocated the specified minimum resources (`requests`) and do not exceed the specified maximum resources (`limits`). For CPU resources, which is the main focus of this work, allocation typically refers to CPU *time* rather than specific cores [31].

2.2 Vertical Autoscaling in K8s

Since K8s configurations are declarative, users have the flexibility to modify the specifications of pod resources at any time. To enable automatic vertical scaling of pod resources based on monitored usage, K8s offers a built-in feature called the Vertical Pod Autoscaler (VPA) [33]. VPA includes a default algorithm based on a decayed histogram to determine future `limits` and `requests` for both memory and CPU (explained in §3.3). However, it also allows users to plug their own scaling algorithm policy if desired.

VPA end-to-end workflow. Figure 1 illustrates the end-to-end flow of a vertical autoscaling system. In the top left corner, we see the target application stateful set that requires vertical scaling (0). Although the figure shows two pod replicas, the number of instances can vary, including a single instance. These instances are managed by a *controller* (1), which performs tasks such as load balancing or HA management, if applicable. The controller also

[2]K8s cluster nodes can be either bare metal or virtual machines.
[3]While our focus is primarily on *stateful sets* due to their relevance to database services, the techniques discussed in this paper are generally applicable to K8s *replica sets* for stateless applications as well.

Figure 2: Resizing the pods of a stateful set in a DB service.

publishes metrics, such as the current CPU usage and allocation for the application, which are stored in a metrics server (2). These metrics can be accessed by the *recommender* algorithm (3), which publishes a decision (4) regarding the optimal resource allocation based on the available metrics information. Finally, a *scaler* entity (5) polls or subscribes to the decision information, performs health and resource safety checks, and enacts the decision (6) by instructing the controller to adjust the resource allocation of the application. This adjustment is done by modifying the `requests` and `limits` settings in the stateful set's specification.

VPA scaling modes. After recognizing the required resource adjustments, the K8s scheduler is tasked with applying the new configuration. Currently, this process involves adjusting one pod in the stateful set at a time by *deallocating* the pod and *rescheduling* it, which we refer to as *rolling updates with restart*[4]. Although the scheduler may assign the pod to the same node, this method results in a process restart and may lead to service interruptions (e.g., dropped connections or transactions) or performance degradation (e.g., due to replica re-synchronization). By performing a rolling update in a stateful set, the operator is given the opportunity to effectively manage connection failovers, replica re-synchronization, and other control plane operations in a more orderly manner, minimizing perceived service downtime for users.

3 MOTIVATION

In this section, we present our selected case study for vertical autoscaling, which involves a real HA DBaaS architecture implemented at Microsoft using traditional RDBMSs deployed on K8s (§3.1). We then outline the key requirements of our autoscaler (§3.2), which are influenced by the characteristics of the service. Finally, we review existing solutions and discuss their limitations in meeting the aforementioned requirements (§3.3).

3.1 Case Study: High Availability DBaaS

In managed offerings of Database A and Database B at Microsoft, the databases run in n pods as part of a K8s stateful set. The stateful set consists of a single writable primary instance that handles most user requests and $n-1$ optional readable secondary replicas. Fig. 2 illustrates this configuration for a common scenario where $n = 3$. While this scheme is common for RDBMSs [28, 43, 47, 55], it differs slightly from traditional horizontal scaling scenarios. We can add replicas, but they cannot serve write-transaction load, as only the primary instance can handle such traffic. Moreover, adding new replicas often involves a "size of data copy" operation to seed the new replica from existing ones. Both of these constraints motivate the need for vertical scaling to accommodate additional load.

Vertical scaling. When adjusting the resource allocation for Database A and Database B, K8s performs rolling updates with restarts, as described in §2.2. During the scaling process, user connections are interrupted when a pod instance restarts. Since most connections are assumed to be directed to the primary replica, the operator policy prioritizes updating the initial primary replica last to avoid additional client failovers if other replicas assuming the primary role are restarted. However, deferring the update of the initial primary replica may result in a delay before users experience the new resource allocations. Throughout our efforts, we have made several improvements to the scaling process. Despite these enhancements, the process can last around 10 minutes, depending on the number of replicas and workload. This duration affects SLAs and influences how frequently scaling algorithms should adjust resources.

Predictability. In our target DBaaS offerings, predictable performance is a key consideration [18, 37]. Therefore, similar to other deployments with comparable requirements on K8s [13, 40, 48], `limits` are set equal to `requests` to ensure predictability. Setting `limits` larger would allow burstability, but then the resources may not be there when needed.

Resource-based pay-as-you-go model. The billing model for users in the DBaaS is based on the *peak* CPU provisioned resources within a certain time period [5]. This means that users are charged according to the maximum value of core `limits` assigned during that time period (ex: $\$x * numcores$). Memory usage, however, is not billed. Additionally, the service rounds up the billing to whole cores. From this billing model, most customers prefer to conservatively allocate redundant resources to have a buffer in case of a burst in workload, preferring high performance over cost. Other customers prefer to err on the side of saving money and may under-provision.

3.2 Requirements

The goal of CaaSPER is to efficiently allocate the appropriate number of cores for the managed RDBMS offering at Microsoft, taking into account users' expected *performance*, *cost*, and *availability*. The architecture and invariants play a crucial role in defining the requirements of our vertical autoscaler, which we present next.

R1 : *Alignment with the service invariants.* Our algorithm should be designed to align with the service's requirement for predictability and its pay-as-you-go billing model. To satisfy this, we enforce that `limits` and `requests` (1) are equal, ensuring that the node has sufficient resources to meet the defined limits, and (2) are integers, conforming to the whole-core billing model.

R2 : *Flexible behavior based on user preferences.* Our design should be flexible enough to accommodate various user preferences, such as prioritizing *cost-savings* or prioritizing *availability* and *performance* for mission-critical workloads. This requires mapping user preferences into parameters that tailor the behavior of the autoscaling algorithm. This also requires that we are able to quickly simulate different scenarios based on those preferences.

R3 : *Optimization based on target function.* Once user preferences and the associated SLA are defined, our objective is to minimize costs and meet the required SLA by fine-tuning the CPU `limits` to align resource allocation with workload demands. Three key metrics are used: (1) average CPU *slack* (over-allocated CPU), which represents the difference between the allocated resource `limits`

[4]We are currently evaluating the stability of a new scaling feature in K8s that does not require restart [32], which we plan to explore further in future work.

[5]This time period may be minutely or hourly depending on configuration.

and the actual usage and directly impacts cost efficiency, (2) the percentage of observations and amount of *throttling* (insufficient CPU), which affect performance metrics such as throughput and latency, and (3) the *frequency of scalings*, which also impact the throughput of the system. Note that maintaining a small slack buffer in the system can help prevent outages and ensure that bursty workloads meet their SLAs, and a system that swiftly reacts to throttling ensures performant services.

R4 : *Application-agnostic.* K8s can be used with a wide range of applications. Therefore, we aim for our solution to be applicable to any application that requires and supports vertical scaling, without being tied to a specific application such as RDBMSs. To that end, it should rely on generic metrics such as CPU usage and provide CPU settings as output. However, in certain types of applications, such as RDBMSs, service load metrics have been shown to be more useful than system metrics for autoscaling [25]. The integration of application-specific extensions into our algorithms and a comparison with our current proposal are left for future work.

R5 : *Support for diverse workloads.* Our solution should be designed to effectively scale resources regardless of the specific workload. Concretely, we identify two key scenarios that our solution needs to address. (1) *Low predictability:* In this scenario, there may be insufficient CPU history data available (e.g., for new workloads) or the data may exhibit unpredictable patterns, rendering accurate ML prediction impossible. (2) *Predictable workloads:* Certain workloads exhibit some predictable patterns, allowing us to leverage the metrics history to more accurately forecast future resource requirements. Examples include cyclical patterns during work-days/weeks, periodic spikes in usage for quarterly reporting, etc.

R6 : *Interpretable.* When using ML to help customers select optimal configurations, it is important that the model is interpretable so that they understand trade-offs and can make an informed decision [15]. To achieve this, our solution should leverage transparent models that clearly highlight performance options.

3.3 Limitations of Existing Solutions

In our investigation of automatic vertical scaling solutions, we examined off-the-shelf options such as the built-in algorithm for VPA [33] and OpenShift's VPA solution [2]. Our goal was to assess how effectively these algorithms fulfilled our specific requirements.

To evaluate them, we set up K8s clusters running three instances of Database A, with `limits` initially set to a maximum of 14 cores. To avoid disruption to the deployment, we implemented logic to prevent autoscaling below 2 cores. Additionally, we modified the existing algorithms to target the primary instance only since its metrics patterns differentiate from secondary replicas[6]. In Figure 3a, we present a control experiment that approximates a real over-provisioned customer scenario. The CPU `limits` are fixed at 14 cores (indicated in red on the right y-axis), while the current CPU usage is depicted in blue on the left y-axis. The experiment covers a 62-hour trace period with 8 hours of usage at approximately ~2-3 cores, followed by 8 hours at ~7 cores, and another 8 hours at ~2-3 cores, repeating throughout the period. As shown, the

[6]This adjustment was necessary because the evaluated VPA algorithms are designed for stateless K8s replica sets, where each replica receives a roughly equal workload distribution (https://github.com/kubernetes/autoscaler/issues/3015).

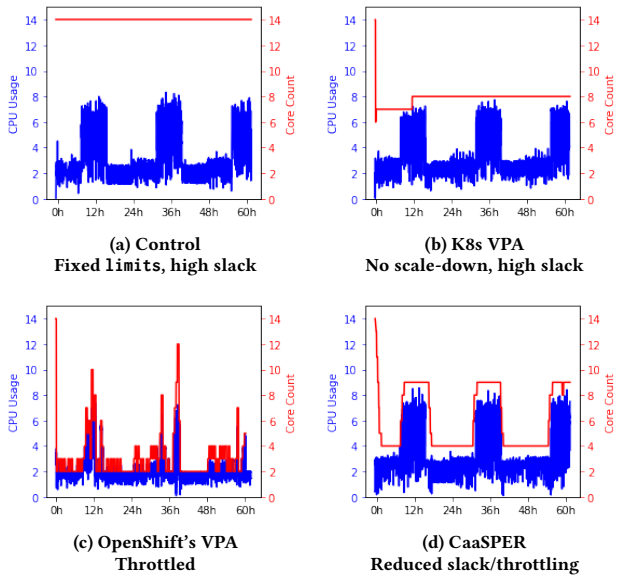


Figure 3: A comparison of existing VPA Recommenders.

fixed CPU `limits` provide a buffer of nearly double the maximum observed resource usage for this workload, resulting in excessive slack (idle resources).

Default VPA algorithm. This algorithm uses a decaying histogram of weighted CPU samples collected at one-minute intervals to determine the new `requests` target based on the 90th percentile (P90) of observed usage within the configured history length. `limits` are usually increased proportionally to `requests`, but not explicitly used in calculations. This means that scaling up is only enabled when the `limits` are strictly greater than the `requests`. However, this conflicts with our original requirement of having `limits` equal to `requests`, as specified in **R1**. To address this discrepancy, we maintained the invariant `limits:=requests+1` so that `limits` is greater than `requests` (to allow VPA to detect the need for scaling) yet as close as possible and aligned with our billing model (**R1**(2)).

Figure 3b shows that the algorithm scales up to 8 cores after traffic increases around the 8th hour. However, during the period of low utilization, it does not scale back down since the P90 usage values within the available history window remain high. Adjusting the safety margin (slack) and history duration in VPA's configuration can encourage more aggressive scaling down, but this comes at the expense of decreased scale-up accuracy. Overall, the default VPA algorithm focuses on scheduling and *responding* to resource usage to meet SLAs rather than providing predictable performance, accounting for future usage, or optimizing costs.

OpenShift's VPA. OpenShift's VPA uses the pluggable nature of the default VPA recommender component to use a different, predictive algorithm. Figure 3c shows the results of evaluating OpenShift's VPA for the same workload. Initially, the recommender component predicts low CPU utilization, resulting in the scaler component setting low `limits`. Consequently, container throttling occurs when setting the `limits` according to the prediction. Moreover, due to the ongoing low CPU metrics resulting from the previous `limits`

setting, the recommender continues to forecast low CPU usage in the future, exacerbating the throttling issue. As a result, the usage is severely capped (compared with Figure 3a where sufficient resources were provisioned), leading to a 73% reduction in throughput, while the `limits` oscillate between 2 and 3 cores (where 2 is our minimum guardrail). In addition to the throttling issue, we found its current implementation, where multiple competing models were retrained at prediction time based on the outcome of a decision tree evaluation, was costly and led to high recommendation latency for our production workloads. Similar to the default VPA algorithm, OpenShift's algorithm focuses on scheduling and optimizing resources based on `requests` and the heuristic of setting limits was not sufficient. Although we believe the algorithm could be modified to consider our emphasis on CPU `limits`, it is currently unsuitable for our workloads or billing model.

Summary. In conclusion, while both solutions are application agnostic (**R4**), they provide limited support for parameterizing the slack (**R3**) and do not consider throttling, as they focus primarily on `requests` rather than `limits`. OpenShift's VPA was particularly ineffective when dealing with existing throttling, leading to inaccurate predictions (**R5**). Furthermore, neither solution offers support for optimizing customer cost-performance preferences (**R2**), and they do not adequately meet the service invariants or interpretability requirements (**R1**, **R6**).

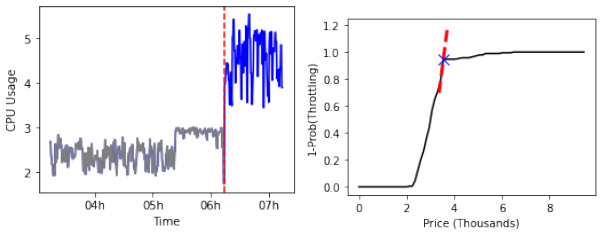
4 THE CAASPER ALGORITHM

In this section, we present the CaaSPER algorithm and how it addresses the previously mentioned requirements. We start by providing background information on Doppler and price-performance curves in §4.1, which is crucial to understand our proposal. Following that, we introduce the *reactive* and *proactive* approaches that together constitute CaaSPER in §4.2 and §4.3, respectively.

4.1 Background on Doppler

In the context of Doppler [11], price-performance curves (*PvP-curves*) are used to aid customers migrating their on-premises databases to the cloud. These curves visually display the monthly prices for various relevant SKUs (*Stock Keeping Units*, e.g., VMs with specific core counts, memory, IOPS, etc.) available for migration, along with the corresponding expected performance for each customer's workload. Typically, these curves show diminishing returns on performance as costs increase. During migration, customers can use the *PvP-curves* to evaluate the potential decrease in performance they may experience in exchange for increased monthly savings, and vice versa.

In order to create these personalized *PvP-curves*, Doppler introduces a proxy for estimating performance, called the *probability of resource throttling*. This metric provides a sufficient approximation[7] of the true throttling, the extent of lower throughput, higher latency, and increased performance bottlenecks. This probability is used to approximate how well each SKU meets customer workload performance requirements. The following equation outlines how



(a) CPU usage (b) *PvP-curve*

Figure 4: CaaSPER appropriately scales up based on the slope inflection point (highlighted by the x and the associated slope at that core count) of the price-performance curve (right) estimated based on traces highlighted on the left in gray.

Doppler estimates performance, defined in this case as the *throttling probability*: $P_n(\cdot)$ of each SKU_i, where $i = 1, \ldots, m$ represents some SKU among m possible SKUs:

$$P_n(\text{SKU}_i) = P(r_{\text{CPU}_n} > R_{\text{CPU}_i} \cup r_{\text{RAM}_n} > R_{\text{RAM}_i} \cup \ldots \cup r_{\text{IOPS}_n} > R_{\text{IOPS}_i}). \quad (1)$$

where $r_{\{\text{CPU}_n,\text{RAM}_n,\ldots,\text{IOPS}_n\}}$ denotes the vector of random variables corresponding to the resource usage for customer n, and $R_{\{\text{CPU}_i,\text{RAM}_i,\ldots,\text{IOPS}_i\}}$ denotes the maximum capacity for each performance dimension as fixed by a specific SKU_i.[8] The advantage of this mathematical formulation is that it can be easily extended to handle any combination of performance dimensions.

Figure 4b provides an example of the *PvP-curves*. Customers are presented with a wide array of SKUs, each with varying monthly prices and the corresponding probability of resource throttling. Ideally, a customer aims to avoid any performance bottlenecks entirely, i.e., a `1-Prob(Throttling)` value of 1; however, they might be reluctant to pay the high monthly price for such assurance. These *PvP-curves* thus serve as a valuable and personalized tool in assisting customers in selecting the optimal database SKU, showcasing the balance between choosing higher-performing and more cost-effective options.

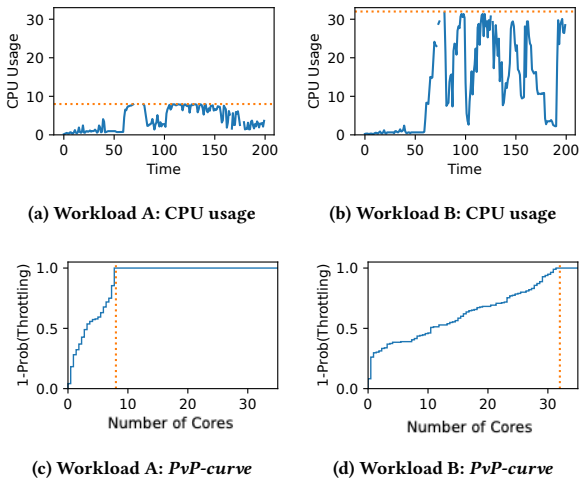
4.2 Reactive Scaling with CaaSPER

In the absence of historical data (e.g. beginning of workload), it is difficult to rely on existing predictive autoscaling tools. Therefore, our focus is on first developing responsive reactive approaches to handle workloads without history data (**R5**). This section outlines our approach, which integrates a hybrid strategy utilizing the previously discussed *PvP-curves* and *protective guardrails*. This approach provides customers with flexibility in adjusting their cluster's size based on the trade-offs between cost and performance (**R2**).

Autoscaling Based on Price-Performance Curves. Considering the limited interpretability of the baseline VPA algorithm and competing techniques (e.g., Holt-Winters and LSTM [73]), our approach builds upon *PvP-curves* to ensure that autoscaling (data-driven) decisions are *interpretable*, in accordance with **R6**. In practice, we refurbish the *PvP-curves* introduced in Doppler to guide our autoscaling decisions by examining the *PvP-curves*' slopes.

[7] In Doppler, workload replay helped to confirm how well this proxy metric approximates performance. Specific customers provided full workloads (e.g. queries and data), and the migration team was able to replay each workload on a wide range of SKUs to examine subsequent performance bottlenecks.

[8] There are certain performance dimensions that require small transformations; for example, IO latency is taken as the *inverse* of the actual IO latency in order to calculate the effect of this performance dimension.



(a) Workload A: CPU usage (b) Workload B: CPU usage

(c) Workload A: *PvP-curve* (d) Workload B: *PvP-curve*

Figure 5: CPU usage (blue line) and price-performance curves for two workload samples of Database A (throttled and not). CPU `limits` are marked with dashed orange lines. A throttled workload is usually associated with a steep slope (lower left).

Applying Price-Performance Curves in CaaSPER. While the original algorithm in [11] depended on multiple performance dimensions (e.g., IOPs, latency, memory, CPU, etc) to select an appropriate SKU, when scaling applications on top of platforms like K8s, each resource can be scaled *independently* and we can treat each resource scaling problem separately. Thus, we refactored the original algorithm to only require CPU utilization as input and output the new number of cores as the scaling decision[9]:

$$a(t) = \text{AUTOSCALE}\left(\text{CoreCount}_{\text{cur}}, \{X_t\}\right) \quad (2)$$

where $\text{CoreCount}_{\text{cur}}$ is the current number of CPUs allocated, X_t is the observed CPU usage pattern for some available time period $t = \{0, 1, ...T-1\}$, and a is the subsequent autoscaling action at time T. This generic input satisfies **R4**.

By examining the position of the current chosen number of cores in the PvP-curve, a clear signal can be captured indicating if the current choice is under- or over-provisioned. Figure 5 shows two examples of CPU traces from Database A customers where the `limits` is marked by an orange dashed line. For a workload that runs close to the limit of 8 cores (Figure 5a), the associated *PvP-curve* in Figure 5c has a relatively steep slope at the point of the selected CPU `limits`. This makes sense considering Equation 1: Given that the majority of the CPU time series (r_{CPU_n}) in Figure 5a reaches the `limits`, opting for fewer than 8 cores would significantly raise the throttling probability. A steep slope on the *PvP-curves* thus signals a customer might be throttled and should consider increasing the number of cores to meet their performance needs. Figure 5b is representative of a customer that is using an appropriate CPU `limits` value since the *slope* of their *PvP-curve* (Figure 5d) at their $\text{CoreCount}_{\text{cur}}$ allocation (i.e., 32 cores) is neither excessively steep nor too flat.

[9]As noted in §2.2, we leave for future work the extension of our autoscaling algorithm to consume other performance input, like memory and IOPs, given appropriate scaling mechanisms exist for these resource dimensions. Our focus on CPU is also driven by the fact that product billing is only dependent on the number of cores.

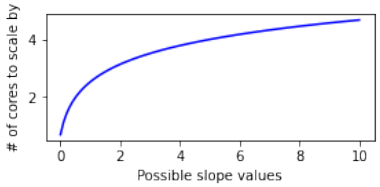


Figure 6: Example shape of scaling-factor function $SF(s)$ of *PvP-curve* slope s. Scale-ups happen more aggressively for large s (more throttling), than small s (less throttling).

Moreover, the slopes *give a good measurement of the severity of throttling and can be used to determine the number of cores to scale up by.* To leverage the *slopes* of *PvP-curves* for autoscaling decisions in CaaSPER and respond to potential throttling, we examined the evolving behavior of *PvP-curves* over time for a set of sophisticated customers that actively optimize the number of cores for their Database A on a daily basis. This gives us a baseline to understand how to correlate slope values to the number of cores by which customers should increase (or decrease) their $\text{CoreCount}_{\text{cur}}$ allocation by, which effectively addresses **R3**. By capturing static snapshots of personalized *PvP-curves* just before a customer makes an autoscaling decision, assessing the slopes at their current core allocation, and monitoring the subsequent increase (or decrease) in cores resulting from the decision, we can precisely determine the extent to which a customer should scale up (or down) by based on the slope of their respective *PvP-curves*. Our analysis revealed that a simple logarithmic decay function suffices for automatically determining the *scaling factor* (SF), representing the appropriate number of cores to scale up (or down) by. Specifically we define:

$$\text{SF}(s, \text{skew}) = \log\left(\text{skew} \times s + c_{min}\right) \quad (3)$$

where s represents the slope at the customer's $\text{CoreCount}_{\text{cur}}$ allocation of their *PvP-curve* (estimated from the usage data $\{X_t\}$), skew captures the asymmetry of the distribution of existing slopes of the *PvP-curve*, and c_{min} is a guardrail for the minimum number of cores required to operate the pod since we want to prevent nonsensical autoscaling decisions that could lead to a service outage. When the distribution has a higher skew, indicating concentration towards lower/higher end of the usage, we scale up/down more aggressively. By letting the SF multiplicative factor in our logarithmic function be driven by the skew estimate, we are able to dynamically capture the desired number of cores we want to scale by. We also have additional adjustable guardrails in place to prevent system failure (ex: minimums based on machine size). Figure 6 shows an example of the logarithmic decay function for a range of slope values.

By using this logarithmic function, the scaling-up decisions tend to be associated with a high number of cores when the derived slope s is high. This is ideal, as high s values are typically associated with a customer getting highly throttled (e.g., their $\text{CoreCount}_{\text{cur}}$ allocation lands them at an inflection point of their curve as in Figure 4). Alternatively, when the derived s values are small (e.g., between 0-2), the associated scaling factor is correspondingly small, leading to better micro-adjustments in the price vs. performance trade-off. Our full autoscaling strategy is depicted in Algorithm 1.

Effectiveness of CaaSPER. Figure 4 presents an instance where a customer's utilization was limited to 3 cores before 06h. Upon

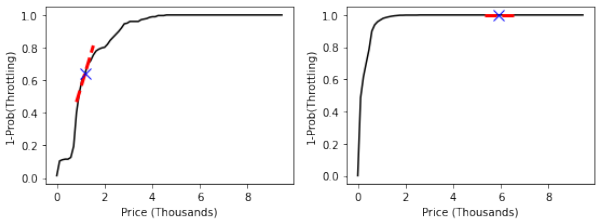
Algorithm 1 CaaSPER autoscaling decision algorithm.

Require: x_c: $\text{CoreCount}_{\text{cur}}$
Require: $\{X_t\}$: Vector of workload CPU usage indexed by time (observed and/or predicted)
Require: R: System inputs (e.g., resource limit such as max CPU, price per core, granularity per core)
Require: s_h: High slope threshold
Require: s_l: Low slope threshold
Require: m_h: High slack threshold as percentage of capacity
Require: m_l: Low slack threshold as percentage of capacity
Require: SF_h: Maximum single step scale-up amount
Require: SF_l: Maximum single step scale-down amount
Require: c_{min}: Minimum resource requirements (scale-down lower bound)

1: **function** AUTOSCALE(x_c, $\{X_t\}$)
2: normalized cpu ← PREPROCESS CPU($\{X_t\}$)
3: PvP curve ← SKU RECOMMENDATION TOOL(normalized CPU, R)
4: PvP slopes ← CALCULATE SLOPES(PvP curve)
5: skew ← CALCULATE SKEW(PvP slopes)
6: s ← GET CURRENT SLOPE(PvP slopes, x_c)
7: SF ← CALCULATE SCALING FACTOR, SF(s, skew)
8: **if** $s \geq s_h$ or Quantile($\{X_t\}$) $\geq (1 - m_h) * x_c$ **then**
9: SF ← min (SF, SF_h)
10: **else if** $s \leq s_l$ or Quantile($\{X_t\}$) $\leq m_l * x_c$ **then**
11: SF ← max ($-$SF, $-\text{SF}_l$)
12: **else if** $s == 0$ and x_c at top of PvP curve **then**
13: SF ← UPDATE SCALING FACTOR(PvP curve, x_c)
14: SF ← APPLY GUARDRAILS(SF, SF_h, SF_l, c_{min}, R)
15: **return** SF

detecting throttling, CaaSPER generated the corresponding *PvP-curve* on the right. Considering the skewness of the slopes in this specific *PvP-curve* and the current slope of 1.38 at the customer's existing allocation of 3 cores, our mathematical model recommends scaling up by 3.73 cores. However, since our system is not yet built to scale by a fractional number of cores, the result is rounded down (configurable) to comply with **R1**. (Although our system could be adapted for sub-core usage in the future.) Continuing the analysis of the customers' CPU utilization post-decision, as indicated by the blue line on the right of Figure 4a, we observe that the algorithm appropriately right-sized the customer pod to 6 cores.

Our autoscaling methodology is highly sensitive to the shape of the *PvP-curve*. As shown in Figure 7a, following our previous design discussion, we would only make relevant autoscale-up decisions when the customer's $\text{CoreCount}_{\text{cur}}$ allocation places them on their respective *PvP-curve* such that the derived slope s is positive. However, we encounter instances where customers land on the far right of the curve, as indicated by the red line snippet in Figure 7b. In this situation, the customer's $\text{CoreCount}_{\text{cur}}$ falls within a consistently flat portion of the curve to the right, resulting in a derived slope s of 0 that indicates significant over-provisioning. To address such cases, we introduce the mechanism to scale down as shown in line 12 of Algorithm 1. This approach prevents situations where the customer is paying for significantly more compute than their workload requires. Here, we walk down the curve (to the left) to identify the cheapest $\text{CoreCount}_{\text{next}}$ that can meet the workload requirements at 100% utilization. For scenarios like Figure 7b where the customer is highly over-provisioned, our algorithm recommends scaling down by almost 8 cores, leading to substantial cost savings.



(a) Typical *PvP-curve* (underprovisioned) **(b) Flat *PvP-curve* (overprovisioned)**

Figure 7: Two instances of *PvP-curves*. The blue X indicates $\text{CoreCount}_{\text{cur}}$ and the red dashed line the derived slope s.

Given the mechanisms described, the algorithm provides a relatively small slack buffer by selecting the $\text{CoreCount}_{\text{next}}$ that most closely aligns with the present workload requirements. However, since reactive scaling mechanisms may require time for resizing (§3.1), many of our customers express the need for additional headroom as guardrails to mitigate potential throttling when workloads' CPU utilization unexpectedly increases. This is addressed in Algorithm 1 with the inclusion of m_h and m_l parameters, allowing us to set a desired buffer amount. In general, our algorithm provides flexibility for trade-offs between price and performance to cater to various customer preferences. As we show in §6.2, our autoscaling algorithm not only adeptly responds to workload changes but also effectively introduces adequate buffer to meet customer requirements, even without prior performance history.

4.3 Proactive Scaling with CaaSPER

The reactive component of CaaSPER adeptly adjusts to workload changes, yet many workload patterns are recurrent. Next we discuss the integration of established techniques related to time series and machine learning algorithms to proactively address scaling needs.

The main idea underpinning the integration of forecasting in CaaSPER is to preprocess input data for Algorithm 1 by combining actual resource utilization data $\{X_t\}$ with predictions (refer to Figure 8). On the first day (`period1`), the algorithm operates reactively due to insufficient historical data. A complete *seasonality period* (depicted by the green region in Figure 8) is awaited before transitioning to proactive mode. Starting from `period2`, there is enough history in the `forecasting window`, with length o_f, to generate a valuable prediction for this cyclical workload. We then combine the original window (e.g., the last 40 minutes of CPU usage typically processed by our reactive autoscaling algorithm) with the predicted data (`forecasting horizon`) to form the combined `new window`, with length o_n, serving as the input for Algorithm 1.

By combining the current data (e.g., CPU usage) with the predicted future data, Equation 2 becomes:

$$a(t) = \text{AUTOSCALE}(\text{CoreCount}_{\text{cur}}, \{X_{t=T-(o_n-o_f),\ldots,T-1}\}, \{\hat{X}_{t=T,T+1,\ldots,T+o_f-1}\}) \quad (4)$$

where $\hat{X}_t$ is the predicted usage for CPU. We can tail the new combined window to give less weight to historical data and rely more on predictions (new window length is a tunable parameter).

The predictive component is pluggable, allowing us to choose different ML algorithms as needed. We experimented with various

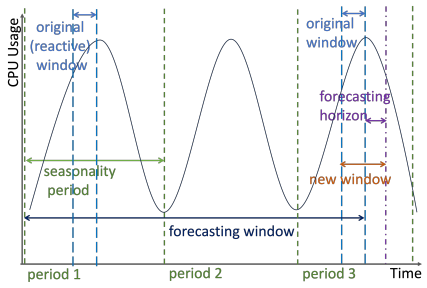


Figure 8: Input data preprocessing for Algorithm 1 for proactive CaaSPER.

algorithms, including those in OpenShift's VPA [2], as well as the naïve and ARIMA forecasters from `sktime` [3], and Prophet [70]. Although we tested them on a limited number of simple workloads, we found the naïve algorithm to be the most lightweight and explainable (details in §6). Exploring additional forecasting algorithms (e.g. LSTM) and the trade-offs between cost of training (which contributes to CPU load) and model accuracy, is beyond the scope of this paper and planned for future work. Currently, our system does not consider the confidence values of model predictions, but we plan to incorporate them as a prefilter in future iterations to improve the balance between predictive and reactive components.

4.4 Discussion

Returning to the example introduced in §3.3, Figure 3d illustrates the scaling behavior of CaaSPER, showing how it (1) utilizes *PvP-curves* and slack to address resource exhaustion when it occurs, and (2) predicts future CPU spikes to enable early scaling up and avoid throttling. The VPA algorithm reduced slack by 61% compared to the control run with 14 cores, while CaaSPER was able to reduce it by 78.3%. Also note that in Figure 3d, at ~8h, there is a small amount of throttling, that is avoided on the subsequent two periods (32h, 56h) shown by a small white vertical gap.

According to Algorithm 1, when dealing with a severely throttled workload that is bottlenecked, our policy identifies the insufficient slack and applies the logarithmic decay function (Eq. 3) to scale the capacity up to the maximum allowed SF_h. By incorporating the *forecasted* CPU usage in the input for generating the *PvP-curves* (Eq. 4), CaaSPER can proactively scale up or down since the curves project the utilization levels a configurable number of minutes ahead of time. This combined approach allows CaaSPER to set the CPU `limits` at an appropriate level to accommodate the workload while also minimizing costs during periods of low demand.

Summary: Requirements met. CaaSPER effectively fulfills the requirements outlined in §3.2. It satisfies the invariants set by our motivating case study (**R1**). CaaSPER offers customers interpretable price-performance curves (**R6**) that are tailored to their specific needs, enabling them to make decisions considering the trade-offs between cost and performance (**R2**). CaaSPER achieves application agnosticism (**R4**) by focusing solely on CPU utilization as input and outputting the core count as the scaling decision. To handle diverse workloads (**R5**), it adopts a combined reactive-proactive approach that leverages the slopes of price-performance curves to make autoscaling decisions based on slack and potential throttling (**R3**).

5 SIMULATOR AND PARAMETER TUNING

As it is evident from the previous section, the development of a comprehensive autoscaling algorithm with adequate flexibility is a complex and time-consuming task. To enhance the explainability of our system and expedite the development process, we created a simulator that allows us to evaluate various autoscaling algorithm policies for CaaSPER using only a CPU trace. The simulator replicates key components outlined in Figure 1, specifically targeting the tuning and testing of pluggable recommender algorithms. This simulator enables us to achieve the following objectives: (1) Simulate autoscaling in scenarios where the live workload (e.g., a user's queries and data) is inaccessible. (2) Evaluate our system's performance against standard workload traces such as the Alibaba dataset [5]. (3) Conduct rapid parameter tuning of our algorithm using existing workload history (**R3**). (4) Adjust parameter combinations based on desired slack, throttling, and scaling frequency (**R2**).

By leveraging the simulator, we enable users to easily choose the parameter settings that align with their desired cost-performance preferences. Specifically, for each experimental run with a specific parameter combination, we capture three metrics:

- $K(\cdot)$: Sum of all slack values, representing the total unused capacity necessary to prevent resource shortage for bursty workloads.
- $C(\cdot)$: Sum of insufficient CPU occurrences, reflecting the total throttling that can result in performance degradation.
- $N(\cdot)$: Total number of scalings, considering that not all systems can scale without downtime; frequent scaling is penalized to prevent performance degradation.

Our tuning primarily focuses on the reactive parameters indicated as **Required** inputs to Algorithm 1 (from s_h to c_{min}) as well as the forecasting window sizes shown in Figure 8. For example, for workloads demanding higher performance, a larger single-step core scale-up count (SF_h) allows the system to scale more rapidly, while a lower minimum core count (c_{min}) reduces the likelihood of throttling during bursts. The opposite holds true for a cost-oriented tuning approach. Furthermore, larger window sizes make CaaSPER less responsive to minor bursts, potentially saving costs, and reduce scaling frequency, thereby improving availability.

In order to identify the Pareto frontier [75] based on the observations in Figure 12, where both slack and throttling are minimized, we introduce the following objective function G:

$$\mathrm{G}(\alpha, p) = \alpha \cdot K(p) + C(p) \tag{5}$$

where α is a scalar coefficient that represents the penalty of having slack (which can be tailored based on preferences of having higher slack versus throttling), p represents the parameter combination, and $K(p)/C(p)$ denotes the observed (simulated) total slack and insufficient CPU, respectively. This function computes an objective value by combining the two observed metrics, assigning a weight of α to the slack (and the weight of throttling is 1).

The optimal parameter combination set (based on preferences) can be estimated by iterating over all possible values of α:

$$\hat{p} = \{\arg\min_{p} G(\alpha, p) | \forall \alpha \in D\} \tag{6}$$

where we sample random numbers from a log-uniform (reciprocal) distribution with $\ln(D) \sim \mathcal{U}(-100, 100)$ to encompass a broad range of values. These parameters can be continually updated as data is accumulated, and the newly tuned parameters can be fed back into CaaSPER at a configurable interval. This flexibility allows users to modify their preferences over time, such as prioritizing higher performance over cost savings during known busy seasons.

Simulator Correctness. To verify that the decisions generated by the simulator are statistically similar to those from actual runs, we use the pairwise t-test [4]. In multiple iterations of our simulator, we observe that the decision value distributions have equal variances. Based on this assumption, we consistently obtain the same conclusion from pairwise t-tests: the decision values produced by the simulator and the real runs (at each time point) are statistically equivalent on average. We maintain an alpha value of 0.05 for statistical significance across all scenarios considered. The consistency in our findings across all tested workloads gives us confidence that the simulator accurately replicates the autoscaling decisions we would obtain when running the same workload on actual deployments.

6 EXPERIMENTS

In this section, we present the performance of CaaSPER in diverse scenarios. We begin by explaining the constraints and scenario (§6.1). We demonstrate the autoscaling capabilities of CaaSPER on both Database A and Database B for both synthetic data and real workloads (§6.2). Additionally, for situations where live execution is not feasible, such as when working with static CPU traces, we reproduce the scaling outcomes using our simulator (§6.3).

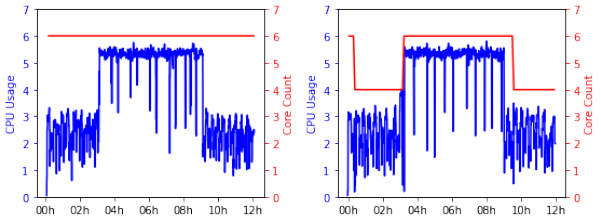
6.1 Scenario

In our experiments, we aimed to align with the billing model and system constraints. To ensure consistency, we followed a set of rules: (1) Our target system bills based on peak `limits` at integer granularity rounded up (**R1**). (2) Since existing VPA implementations require `limits` and `requests` to be *unequal*, we cannot use an exact comparison and rely on an approximation (§3.3). (3) We selected the "control" `limits` to be approximately equal to the total number of utilized cores in the expected peak workload. This mimics an ideal oracle where no throttling or scaling occurs. (4) For Database A, which had 3 replicas, resizing operations are subject to a 5-15 minute window due to the rolling update and HA constraints. For Database B, we set it up read-only across the 2 replicas; therefore resizing takes approximately 3-5 minutes to complete.

6.2 Live System Evaluation

In this section, we demonstrate the scaling capabilities of CaaSPER on managed K8s clusters in the public cloud at Microsoft using two different-sized clusters and various workloads. The "small cluster" has 6 VMs each with 8 CPUs and 32GB and the "large cluster" has 6 VMs each with 16 CPUs and 56GB RAM. For the resizing, we use the rolling update and failover system outlined in §3.1. For the workloads, we generate load using a selection of queries across the TPC-H, TPC-C, and YCSB benchmarks, using BenchBase [19] to drive the client's workload across many terminals.

Right-sizing without history. In §4.2, we described CaaSPER's ability to scale without prior history. To demonstrate this, we started



(a) Workday - Control. **(b) Workday - CaaSPER**

Figure 9: Synthetic workload on Database A.

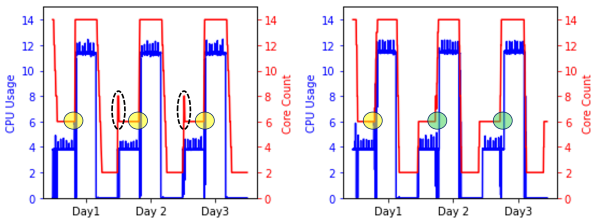
	Non-Cyclical on Database A (12 hrs, 1.2M txns)		Cyclical on Database B (3 days, 3M txns)		
	Ctrl (No Resize)	CaaSP. (React. Only)	Ctrl (No Resize)	CaaSP. (React. Only)	CaaSP. (+Proact.)
Avg Lat (ms)	141±4	138±4	22±1	22±1	22±1
Med Lat (ms)	62±2	61±2	87±2	87±2	86±2
Price ($)	x	$.85x$	y	$.57y$	$.56y$

Table 1: CaaSPER has a large improvement on price with negligible latency impact.

with a 12-hour workload that we ran on Database A in the small cluster. The first 3 and last 3 hours are a mix of read and write transactions with a CPU usage of ~1-3.3 cores, and the middle 6 hours are batches of read-only queries requiring ~5.5 cores. Figure 9a shows a control run fixed at 6 cores shown on the right y-axis in red. Figure 9b shows the workload autoscaled with CaaSPER. As mentioned, we do not use prediction, only the reactive mode, due to lack of historical data. At ~3h, the workload shifts to the heavier portion. Although there is some throttling initially (due to the 10-15 minute scale-up period required by Database A while the secondaries are updated before updating the primary), CaaSPER is able to reactively scale up to adequately accommodate the workload. During each of the 3 resizings (~0h, ~3h, ~9h), one transaction is dropped and retried.[10] The second and third columns of Table 1 show the throughput, latency, and price per transaction for the control and the CaaSPER-scaled workloads for Figure 9. With CaaSPER, total slack was reduced by 39.6%, the cost is 85% of the original cost, and the impact to latency and throughput is within the margin of error, measured by multiple runs in the same cluster.

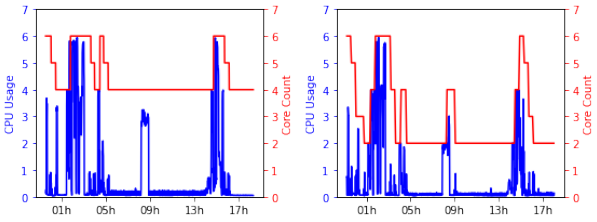
Cyclical workloads. Next we show the impact of adding prior history, the proactive portion of CaaSPER. First, in Figure 10a, we show a 3-day, synthetic cyclical load with 3 million transactions on Database B that is purely reactive with some over-corrections (dashed ovals) and throttling (yellow circles). Then we show that same workload with the addition of forecasting as described in § 4.3 in Figure 10b. Both graphs of Figure 10 show that Day 1 is nearly identical in terms of the number of cores that CaaSPER chooses. On Day 2, CaaSPER is able to correctly scale up to 6 cores in the proactive system rather than overshooting to 8 cores like in the reactive only. Then, during the large 12-core spike on Day 2, in Figure 10b there is no throttling as the `limits` jump to 14 cores, as shown by the green circle and the small (configurable) gap between the red (top) and blue (lower) lines. For this experiment, we set the

[10] In our initial tests with the new in-place resize feature for K8s (§3.1), neither the scale-up lag nor failed transactions occur.



(a) Reactive only. (b) With Proactive.

Figure 10: Reactive and Proactive scaling on Database B. With Proactive, CaaSPER is able to preemptively scale up avoiding the mistakes in the dashed circles, and not throttle (like in yellow circles) on Days 2 and 3 (as seen in the green circles).



(a) Prefer high-performance. (b) Prefer low-cost.

Figure 11: Autoscaling a recreated customer workload on Database A for two different tuning preferences. (Note that Database A has a mandatory 2-core minimum to run.)

	Total Thrpt (#Txns)	Avg Lat (ms)	Med Lat (ms)	¢/Txn	Total $
Control	300K	~60	~35	x	y
CaaSPER: Prefer Perf	300K	~70	~35	$.74x$	$.74y$
CaaSPER: Prefer Savings	270K	~100	~35	$.54x$	$.49y$

Table 2: Using CaaSPER to balance customer preferences. Saving half the cost shows only a 10% throughput impact.

scale-ahead window gap to 1 hour to display on the graph more clearly; but in practice we set this smaller to increase savings. To the far right of Figure 10b, CaaSPER proactively scales up for the beginning of the fourth day. If the traffic does not arrive as predicted, CaaSPER would scale back down as the reactive component takes over. Table 1 shows the performance impact to throughput and latency for this cyclical experiment. In the control run, the core count is held at 14 cores for the duration. For the 3M transactions, the average and median latency are within the margin of error across all experiments. For the price per transaction, we see that the proactive version is lowest overall because it is able to quickly scale from 14 to 2 cores rather than slowly scaling back down over the course of an hour. Additionally, total slack was reduced by 66.5% in 10a and 68.2% in 10b compared to the control run.

Customer CPU Trace. We also evaluated CaaSPER using a sample of a real Database A workload CPU trace, synthesized to mimic real customer workload traces. To do this, we used Stitcher [72], an open-source tool published by Microsoft which recreates customer CPU and I/O traces using a mix of public benchmarks to mimic the real workload (matching the same resource utilization characteristics) rather than proprietary data and queries. The customer used Database A, which takes 5-15 minutes to scale up due to strict HA policies. For this experiment, we used the small K8s cluster which had other customer-required services running, bounding the `limits` to a max of 6 cores, and Database A mandates a minimum of 2 cores. To understand the impact of throttling for this experiment, we did not retry throttled transactions after a timeout window.[11]

We started with a control run with `limits` fixed at 6 cores (not pictured), which completed 300k transactions as shown in the first row of Table 2. We followed with two customer scenarios tuned based on preferences as described in §5: one preferring high performance (Figure 11a), and one preferring cost savings (Figure 11b). In particular, the cost-saving scenario was tuned to allow a minimum of only 2 cores (c_{min}), where the high-performance scenario required 4 cores minimum. Table 2 shows that the performance-preferred run with CaaSPER completes the same number of transactions as the control for only 74% of the original price. The savings-focused run completed 10% fewer total transactions, but only cost approximately *half* of the original price. The impact to average latency in the two CaaSPER runs is due to Database A taking 10-15 minutes to scale up during throttling scenarios; as mentioned in §2.2, future improvements can reduce this number further.

6.3 Simulated System Evaluation

Evaluation using the live system can be time-consuming, especially for workloads that run over many days. To better evaluate CaaSPER's performance, we built and leverage a simulator to mimic the system behavior as described in Section 5. In this section, we present results for (1) parameter tuning using the simulator to pick the optimal sets of configurations that are both high-performance and cost-efficient; and (2) evaluation of CaaSPER using the workload trace from Figure 10 and Alibaba's workload traces [5].

Parameter tuning. To find the best parameter setting for autoscaling, we did a random search over the parameters described in §5, with a total of 5000 combinations per CPU trace. Achieving an optimal balance among availability, performance, and cost often involves trade-offs between the different dimensions, creating a Pareto frontier of potential solutions. A good combination of parameters should minimize the slack K to be cost-efficient, total throttling C to be performant, and total scalings N to avoid impacting availability, forming the Pareto frontier obtained by Equation (5).

Figure 12 shows a scatter plot for the workload in Figure 10 with slack vs. throttling[12] where each dot corresponds to one run of the experiment with one parameter combination. In the figure, blue dots are the combinations that included the proactive approach, while green shows those only using the reactive algorithm. Different Pareto optimal points (red ×) on the frontier in Figure 12 represent different combinations and balance between throttling and slack. We can see a clear trade-off between the two metrics—higher slack reduces the likelihood of throttling, and vice versa. The results also show that predictive runs have higher slack, as expected, as they allow for upfront scaling and lower throttling values. A drill-down

[11] In practice, customer applications would typically retry transactions.
[12] Total scalings (availability) dimension omitted for visualization purposes.

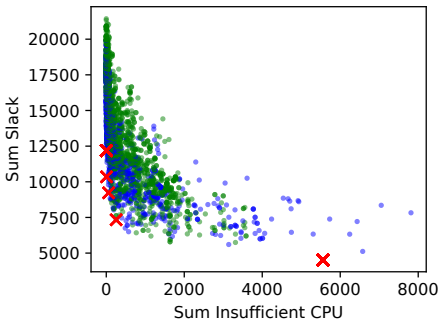
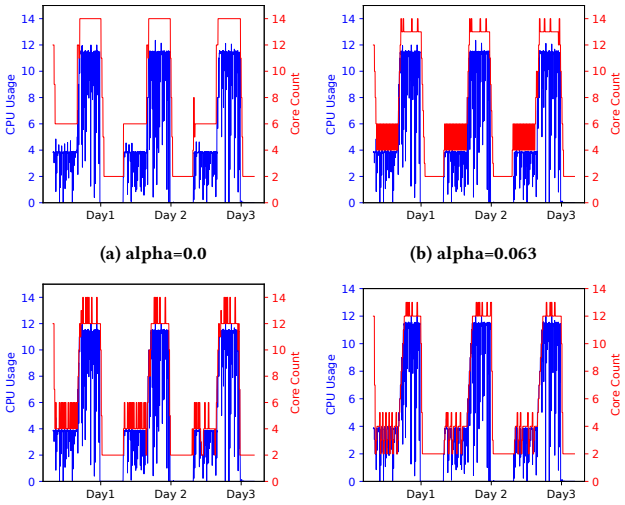


Figure 12: Total slack vs throttling.[12] Green areas are reactive runs, blue - predictive. Red × are Pareto frontier points.



(a) alpha=0.0 **(b) alpha=0.063**

(c) alpha=0.447 **(d) alpha=2.28**

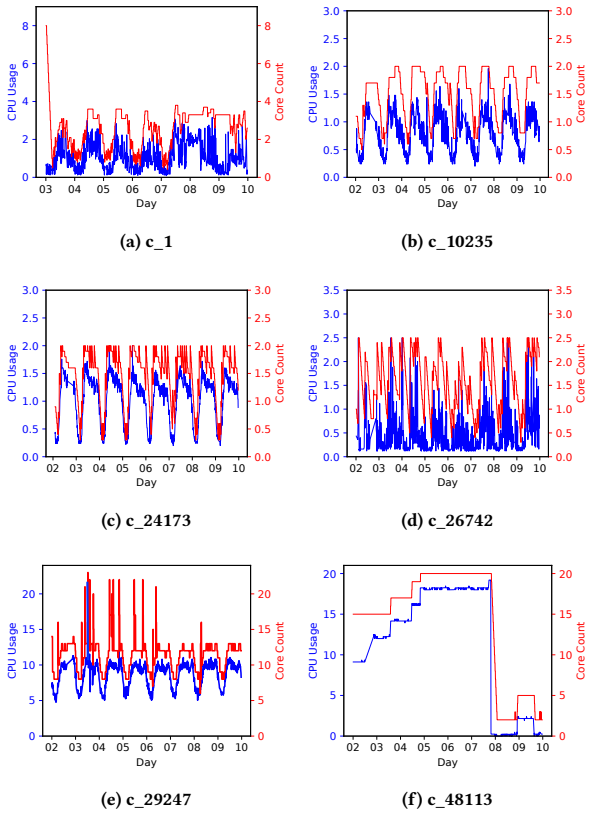
Figure 13: Sampling of α values (weight of slack vs throttling) for Figures 10 and 12.

sampling of runs for different α values (weight of slack parameter) is in Figure 13. As α increases, slack diminishes, and throttling rises.

Alibaba cluster trace tests. We selected 9 representative runs from the Alibaba data set [5] using k-means clustering [10] (4043, 29759, 23544, 24173, 26742, 48113, 29247, 12104, 29345) and chose an additional 2 runs (1, 10235) from [73]. We tuned the parameters with the simulator as described above. Every workload was resampled to have regular data points for every minute (so each workload has around 11k data points to simulate). Because the number of cores in the traces was often millicores and our prototype is designed for full cores, we scaled the number of cores in the trace to integer values in range of our instance max sizes instead of millicores. (Ex: for a range of 0.000-3.000 cores in a trace, we scaled to 0-30 cores by multiplying the millicores by 10 and rounding to the nearest integer.) To visualize, we converted the values back to the original. Based on desired slack and throttling we chose parameters that resulted in the scaling decisions in Figure 14 and metrics in Table 3. Note that compared with works like [73], we focus on setting `limits` instead of just `requests`. Therefore, we would have to set their `limits` to a scalar factor N of `requests` to account

Workload	Average Slack	Num Scalings	Average Insuff. CPU	Throttling Obvsns. %
c_1	1.54	259	0.002	0.90
c_4043	0.15	163	0.000	0.16
c_10235	0.70	173	0.000	0
c_12104	3.94	110	0.005	0.49
c_23544	0.80	115	0.001	0.49
c_24173	0.44	373	0.000	0.52
c_26742	1.14	443	0.004	1.21
c_29247	2.8	155	0.005	0.27
c_29345	2.81	382	0.001	0.17
c_29759	0.64	218	0.000	0.04
c_48113	2.85	38	0.000	0

Table 3: Performance summary for Alibaba workload traces.



(a) c_1 **(b) c_10235**

(c) c_24173 **(d) c_26742**

(e) c_29247 **(f) c_48113**

Figure 14: Alibaba Workloads: CPU and CaaSPER decisions.

for the "insufficient CPU" (relative to `requests` that the system actually used in their case) before computing average slack and "insufficient CPU" (throttling experienced in our case). As that data is not available, a direct comparison is not possible.

The c_29247 trace (Figure 14e) has a significant CPU spike on Day 3 that is not consistent with the other days. The lower accuracy of the naïve forecasting for this workload caused by the huge outlier spike is then projected onto future days and a higher CPU slack relative to actual CPU usage can be seen on Days 4-6. Choosing the right forecasting model or using its confidence intervals to filter predictions fed to our algorithm is future work and out of scope for this paper. Even so, the reactive portion of the model is quick to adjust and reduce unnecessary slack while still avoiding throttling.

7 RELATED WORK

Researchers have extensively studied various aspects of resource management in prior works [7, 29, 30, 42, 46, 50, 52]. Due to space limitations, we focus primarily on works closely related to our own.

Database-Tailored Vertical Autoscaling. [17] aligns closely with CaaSPER's objectives as it aims to automate the scaling of resource container sizes based on signals derived from database engine telemetry. In contrast, CaaSPER follows a more generic approach that is suitable for a broader range of applications, while still being applicable to database systems. Nonetheless, we recognize the potential value in incorporating application-specific telemetry into CaaSPER to improve decision-making, especially as we extend the system to support other resource types like memory. Additionally, research in workload prediction [38, 51], resource usage estimation [49, 56, 68], and proactive scaling [64, 65] could be leveraged to propose alternative prediction algorithms for CaaSPER. While these approaches present interesting directions, they lack the generality of CaaSPER, making them complementary to our proposal.

Other prior works [16, 21] have also addressed scenarios in which servers exhibit significant underutilization. They proposed consolidation strategies to reduce the number of physical servers needed for database infrastructure. Our work represents an important step in this direction, as the optimization of pod instance sizes is critical in enabling K8s to make adequate decisions about pod placement.

General-Purpose Vertical Autoscaling. There are several published works related to vertical autoscaling in cloud workloads. In [73], the authors show that the default VPA is not sufficient, and apply the Long Short-Term Memory (LSTM) and Holt-Winters algorithms to set the `requests` (minimal) bounds for scheduling and scaling. As per §4.3, we utilize a lightweight naïve time series forecasting algorithm in our approach (keeping it simple is shown to work well[14]), while enabling pluggability for more advanced algorithms like LSTM as future work. This work also differs from our setup because we focus on `limits` (maximal) bounds for billing and scaling, and we combine the predictive approach with a novel reactive approach. In [76], the authors show how the VPA default algorithm is not suitable for serverless short-running jobs and propose lightweight rightsizing and autoscaling mechanisms, such as simple and exponential moving averages. In Autopilot [67], they use vertical scaling to reduce slack and prevent throttling in their workloads. We also focus on reducing slack, but our approach to throttling is a bit more cautious because we set these `limits` from a customer-facing perspective. In this work, they focus on optimizing multi-tenant scheduling rather than applying customer preferences.

Additional Scaling Approaches. There is a broad body of work focused on scaling VMs or on scaling K8s pods horizontally [59–61], which has commonality with our vertical pod autoscaling. In [26], the authors present an approach for predictive scaling focusing on optimizing resource estimation for elastic VMs. However, their prediction approaches work for repetitive workloads with consistent patterns that can be readily detected through stationary Markov transition probabilities or Fourier transformation, which might not be universally applicable. Additionally, many history-based approaches break down when the workload history contains throttling (§4.3), as the predicted `limits` may still throttle the workload.

Database Workloads on K8s. Several studies [13, 20, 66] have investigated the effective deployment of database workloads in K8s. [12] evaluates the performance of SQL scalable systems on K8s using the TPC-H benchmark, with an emphasis on helping tenants to reason about monetary budget and query latency.

Workload Simulation for Autoscaling. There are several K8s load-based simulators and generators adopted in industry [69, 71] that focus on helping customers an under how autoscaling behaves during different types of loads. These simulators currently only deal with reactive autoscalers, whereas ours is also predictive. Works like [62, 63] take a look at predictive autoscaling as well as the different levels of autoscaling (e.g., K8s cluster nodes as well as Pods), but do not address availability or cost. Additionally, many autoscaling cloud simulators assists with load forecasting, planning [8], or resource optimization [9]. Our CPU simulator focuses on optimizing `limits` with respect to performance vs cost, allowing for customization and adjustments based on customer preference.

Performance-Cost Trade-offs in Cloud Systems. Relying on more transparent, explainable methodologies to drive autoscalers is not new. [23] is one example that deploys an adaptive search metaheuristic to identify the best autoscaling outcome based on an objective that balances performance and cost trade-offs. A similar suite of techniques exists that leverage sophisticated machine learning, like Q-learning [74], rule-based [22], and queuing [36] algorithms, to optimize cost and performance requirements in making autoscaling decisions. [44] highlights the need for explainable strategies in autoscaling, however their approach is even more customized, requiring the need to conduct extensive interviews to define metrics outside of the traditional scope of performance.

8 CONCLUSION

In this paper, we introduced CaaSPER, a novel reactive-proactive algorithm that allows users to vertically scale their DBaaS workloads according to their preferences and performance requirements. Our approach, leveraging price-performance curves and a slope function, efficiently identifies and responds to throttling, ensuring excellent performance. Our results demonstrate that CaaSPER effectively minimizes throttling across diverse workloads while reducing the required amount of slack buffer.

For future research, we plan to integrate CaaSPER with other VM and cloud fabrics within Microsoft to expand its applicability. As we continue to analyze customer workloads, we plan to explore alternative ML prediction algorithms beyond the initial approach presented in this paper. By incorporating ML predictors that provide confidence intervals rather than point estimators, we can guide scaling actions with greater precision and adjust our decision-making to be more conservative or aggressive based on prediction quality. Further, we aim to investigate automatic scaling of other resource types, e.g., memory, disk. The integration of application-specific metrics, such as buffer pool utilization in RDBMSs, could further enhance performance. Lastly, we plan to integrate the *in-place update without restart* [32] feature of K8s with CaaSPER, eliminating potential downtime or disconnections during resizing operations, ensuring seamless transitions and uninterrupted service. These steps will contribute to CaaSPER's continuous evolution, providing users with an effective solution for automatic vertical autoscaling.

REFERENCES

[1] 2015. Cgroups v2. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html Accessed on May 23, 2023.

[2] 2023. Nodes, pods, and the vertical autoscaler. https://docs.openshift.com/container-platform/4.9/nodes/pods/nodes-pods-vertical-autoscaler.html Accessed May 2023.

[3] 2023. *sktime*. https://github.com/sktime/sktime/tree/main Accessed May 2023.

[4] 2023. *Student's t-test: Paired samples*. https://en.wikipedia.org/wiki/Student%27s_t-test#Paired_samples Accessed on June 02, 2023.

[5] Alibaba Inc. 2018. Alibaba Open Cluster Trace. https://github.com/alibaba/clusterdata.

[6] Amazon. 2023. AWS. https://aws.amazon.com/.

[7] Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, Jiaqi Liu, Lukas Maas, Akshay Mata, Ishai Menache, Justin Moeller, Vivek Narasayya, Matthaios Olma, Morgan Oslake, Elnaz Rezai, Yi Shan, Manoj Syamala, Shize Xu, and Vasileios Zois. 2023. Flexible Resource Allocation for Relational Database-as-a-Service. In *49th International Conference on Very Large Data Bases*. VLDB Endowment, VLDB, 4202–4215.

[8] Mohammad S. Aslanpour, Adel N. Toosi, Javid Taheri, and Raj Gaire. 2021. AutoScaleSim: A simulation toolkit for auto-scaling Web applications in clouds. *Simulation Modelling Practice and Theory* 108 (April 2021), 102245. https://doi.org/10.1016/j.simpat.2020.102245

[9] Luciano Baresi and Giovanni Quattrocchi. 2020. A simulation-based comparison between industrial autoscaling solutions and cocos for cloud applications. In *2020 IEEE International Conference on Web Services (ICWS)*. IEEE, 94–101.

[10] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Vol. 4. Springer.

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

[12] Cristian Cardas, José F Aldana-Martín, Antonio M Burgueno-Romero, Antonio J Nebro, Jose M Mateos, and Juan J Sánchez. 2022. On the performance of SQL scalable systems on Kubernetes: a comparative study. *Cluster Computing* (2022), 1–13.

[13] Mesut Celik. 2019. *Hazelcast: High Performance Cloud-Native Microservices With Distributed Caching*. https://files.devnetwork.cloud/DeveloperWeekAustin/presentations/2019/Mesut-Celik.pdf

[14] Georgia Christofidi, Konstantinos Papaioannou, and Thaleia Dimitra Doudali. 2023. Is Machine Learning Necessary for Cloud Resource Usage Forecasting?. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 544–554. https://doi.org/10.1145/3620678.3624790

[15] Kristof Coussement and Dries F Benoit. 2021. Interpretable data science for decision making. , 113664 pages.

[16] Carlo Curino, Evan P. C. Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-aware database monitoring and consolidation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. ACM, 313–324. https://doi.org/10.1145/1989323.1989357

[17] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. 2016. Automated Demand-Driven Resource Scaling in Relational Database-as-a-Service. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1923–1934. https://doi.org/10.1145/2882903.2903733

[18] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[19] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[20] Charlotte Dillon. 2020. *How to run a software-as-a-service on Kubernetes*. https://www.cockroachlabs.com/blog/kubernetes-saas-implementation/

[21] Aaron J. Elmore, Sudipto Das, Alexander Pucher, Divyakant Agrawal, Amr El Abbadi, and Xifeng Yan. 2013. Characterizing tenant behavior for placement and crisis mitigation in multitenant DBMSs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 517–528. https://doi.org/10.1145/2463676.2465308

[22] Alexandros Evangelidis, David Parker, and Rami Bahsoon. 2018. Performance modelling and verification of cloud-based auto-scaling policies. *Future Generation Computer Systems* 87 (2018), 629–638.

[23] Iure Fé, Rubens Matos, Jamilson Dantas, Carlos Melo, Tuan Anh Nguyen, Dugki Min, Eunmi Choi, Francisco Airton Silva, and Paulo Romero Martins Maciel. 2022. Performance-Cost Trade-Off in Auto-Scaling Mechanisms for Cloud Computing. *Sensors* 22, 3 (Feb. 2022), 1221. https://doi.org/10.3390/s22031221

[24] Daniela Florescu and Donald Kossmann. 2009. Rethinking Cost and Performance of Database Systems. *SIGMOD Rec.* 38, 1 (jun 2009), 43–48. https://doi.org/10.1145/1558334.1558339

[25] Michael A. Georgiou, Aristodemos Paphitis, Michael Sirivianos, and Herodotos Herodotou. 2019. Towards Auto-Scaling Existing Transactional Databases with Strong Consistency. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. 107–112. https://doi.org/10.1109/ICDEW.2019.00-26

[26] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. 2010. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *2010 International Conference on Network and Service Management*. 9–16. https://doi.org/10.1109/CNSM.2010.5691343

[27] Google. 2023. Google Cloud Platform. https://console.cloud.google.com.

[28] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Quebec, Canada) *(SIGMOD '96)*. Association for Computing Machinery, New York, NY, USA, 173–182. https://doi.org/10.1145/233269.233330

[29] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/3135974.3135993

[30] Hamed Hamzeh, Sofia Meacham, and Kashaf Khan. 2019. A New Approach to Calculate Resource Limits with Fairness in Kubernetes. In *2019 First International Conference on Digital Data Processing (DDP)*. 51–58. https://doi.org/10.1109/DDP.2019.00020

[31] https://kubernetes.io/. 2023. Assign CPU Resources to Containers and Pods. https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/

[32] https://kubernetes.io/. 2023. *In-Place Update of Pod Resources*. https://github.com/kubernetes/enhancements/issues/1287

[33] https://kubernetes.io/. 2023. *Kubernetes' Vertical Pod Autoscaler*. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler

[34] https://kubernetes.io/. 2023. *Resource Management for Pods and Containers*. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

[35] https://kubernetes.io/. 2023. *StatefulSets*. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

[36] Gaopan Huang, Songyun Wang, Mingming Zhang, Yefei Li, Zhuzhong Qian, Yuan Chen, and Sheng Zhang. 2016. Auto scaling virtual machines for web applications with queueing theory. In *2016 3rd International conference on systems and informatics (ICSAI)*. IEEE, 433–438.

[37] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas F. Wenisch. 2017. A Top-Down Approach to Achieving Performance Predictability in Database Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 745–758. https://doi.org/10.1145/3035918.3064016

[38] Xiuqi Huang, Yunlong Cheng, Xiaofeng Gao, and Guihai Chen. 2022. TEALED: A Multi-Step Workload Forecasting Approach Using Time-Sensitive EMD and Auto LSTM Encoder-Decoder. In *Database Systems for Advanced Applications*. 706–713.

[39] IBM. 2023. *IBM Cloud Databases for PostgreSQL*. https://www.ibm.com/cloud/databases-for-postgresql

[40] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. 2019. Performance Modeling for Cloud Microservice Applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (Mumbai, India) *(ICPE '19)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/3297663.3310309

[41] MotherDuck Jordan Tigani. 2023. *The Simple Joys of Scaling Up*. https://motherduck.com/blog/the-simple-joys-of-scaling-up/

[42] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 783–798.

[43] Bettina Kemme and Gustavo Alonso. 2010. Database Replication: A Tale of Research across Communities. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 5–12. https://doi.org/10.14778/1920841.1920847

[44] Floriment Klinaku, Sandro Speth, Markus Zilch, and Steffen Becker. 2023. Hitchhiker's Guide for Explainability in Autoscaling. In *Companion of the 2023 ACM SPEC International Conference on Performance Engineering*. 277–282.

[45] Kubernetes. 2023. *Kubernetes*. https://kubernetes.io/

[46] Arnd Christian König, Yi Shan, Karan Newatia, Luke Marshall, and Vivek Narasayya. 2023. Solver-In-The-Loop Cluster Resource Management for Database-as-a-Service. In *50th International Conference on Very Large Databases*. VLDB Endowment, Inc.

[47] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. 2013. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Eng. Bull.* 36, 2 (2013), 6–13. http://sites.computer.org/debull/A13june/TimesTen1.pdf

[48] Noam Levy. 2022. *Groundcover: Between predictable and practical - on kubernetes limits.* https://www.groundcover.com/blog/kubernetes-limits

[49] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *Proc. VLDB Endow.* 5, 11 (2012), 1555–1566. https://doi.org/10.14778/2350229.2350269

[50] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. 2014. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing* 12 (2014), 559–592.

[51] Venkata Vamsikrishna Meduri, Kanchan Chowdhury, and Mohamed Sarwat. 2021. Evaluation of machine learning algorithms in predicting the next SQL query from the future. *ACM Transactions on Database Systems (TODS)* 46, 1 (2021), 1–46.

[52] Themis Melissaris, Kunal Nabar, Rares Radut, Samir Rehmtulla, Arthur Shi, Samartha Chandrashekar, and Ioannis Papapanagiotou. 2022. Elastic Cloud Services: Scaling Snowflake's Control Plane. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 142–157. https://doi.org/10.1145/3542929.3563483

[53] Microsoft. 2023. Azure. https://azure.microsoft.com/en-us.

[54] Microsoft. 2023. *Azure Arc-enabled SQL Managed Instance.* https://learn.microsoft.com/en-us/azure/azure-arc/data/managed-instance-overview

[55] Ross Mistry and Stacia Misner. 2014. *Introducing Microsoft SQL Server 2014.* Microsoft Press.

[56] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* ACM, 301–312. https://doi.org/10.1145/2463676.2467800

[57] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. 2013. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *CIDR 2013* (cidr 2013 ed.). 6th Biennial Conference on Innovative Data Systems Research. https://www.microsoft.com/en-us/research/publication/sqlvm-performance-isolation-in-multi-tenant-relational-database-as-a-service/

[58] Vivek Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. 2015. Sharing Buffer Pool Memory in Multi-Tenant Relational Database-as-a-Service. *Proc. VLDB Endow.* 8, 7 (feb 2015), 726–737. https://doi.org/10.14778/2752939.2752942

[59] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. 2020. Horizontal pod autoscaling in Kubernetes for elastic container orchestration. *Sensors* 20, 16 (2020), 4621.

[60] Zhicheng Pan, Yihang Wang, Yingying Zhang, Sean Bin Yang, Yunyao Cheng, Peng Chen, Chenjuan Guo, Qingsong Wen, Xiduo Tian, Yunliang Dou, Zhiqiang Zhou, Chengcheng Yang, Aoying Zhou, and Bin Yang. 2023. MagicScaler: Uncertainty-Aware, Predictive Autoscaling. *Proc. VLDB Endow.* 16, 12 (aug 2023), 3808–3821. https://doi.org/10.14778/3611540.3611566

[61] Linh-An Phan, Taehong Kim, et al. 2022. Traffic-aware horizontal pod autoscaler in Kubernetes-based edge computing infrastructure. *IEEE Access* 10 (2022), 18966–18977.

[62] Vladimir Podolskiy. 2021. *Multilayered Autoscaling Policies Simulation Toolbox.* https://github.com/Remit/autoscaling-simulator

[63] Vladimir Podolskiy. 2021. Predictive Autoscaling for Multilayered Cloud Deployments. , 183 pages.

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

[65] Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. 2022. Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless. In *VLDB 2022.* ACM, 1279–1287.

[66] Sergey Pronin. 2023. *DBaaS on Kubernetes: Under the Hood.* https://www.percona.com/blog/dbaas-on-kubernetes-under-the-hood/

[67] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: Workload Autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 16, 16 pages. https://doi.org/10.1145/3342195.3387524

[68] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. 2008. Automatic virtual machine configuration for database workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008.* ACM, 953–966. https://doi.org/10.1145/1376616.1376711

[69] SpeedScale. [n.d.]. *How to Test Autoscaling in Kubernetes.* https://speedscale.com/blog/how-to-test-kubernetes-autoscaling/

[70] Sean J Taylor and Benjamin Letham. 2018. Forecasting at scale. *The American Statistician* 72, 1 (2018), 37–45.

[71] VMware. 2022. *Use the Autoscaler in Simulation Mode to Learn Service Behavior Without Scaling the Target Service.* https://docs.vmware.com/en/VMware-Tanzu-Service-Mesh/services/service-autoscaling-with-tsm-user-guide/GUID-54E2BDEC-283F-4BB7-A786-5E260E806EF5.html

[72] Chengcheng Wan, Yiwen Zhu, Joyce Cahoon, Wenjing Wang, Katherine Lin, Sean Liu, Raymond Truong, Neetu Singh, Alexandra Ciortea, Konstantinos Karanasos, et al. 2023. Stitcher: Learned Workload Synthesis from Historical Performance Footprints. (2023).

[73] Thomas Wang, Simone Ferlin, and Marco Chiesa. 2021. Predicting CPU Usage for Proactive Autoscaling. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (Online, United Kingdom) *(EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 31–38. https://doi.org/10.1145/3437984.3458831

[74] Yi Wei, Daniel Kudenko, Shijun Liu, Li Pan, Lei Wu, and Xiangxu Meng. 2019. A reinforcement learning based auto-scaling approach for SaaS providers in dynamic cloud environment. *Mathematical Problems in Engineering* 2019 (2019).

[75] Wikipedia. 2023. Pareto front. https://en.wikipedia.org/wiki/Pareto_front.

[76] Yuxuan Zhao and Alexandru Uta. 2022. Tiny Autoscalers for Tiny Workloads: Dynamic CPU Allocation for Serverless Functions. In *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022, Taormina, Italy, May 16-19, 2022.* IEEE, 170–179. https://doi.org/10.1109/CCGrid54584.2022.00026