Kerveros: Efficient and Scalable Cloud Admission Control

Sultan Mahmud Sajal, *Microsoft Research and Pennsylvania State University;*
Luke Marshall and Beibin Li, *Microsoft Research;* Shandan Zhou and Abhisek Pan,
Microsoft Azure; Konstantina Mellou and Deepak Narayanan, *Microsoft Research;*
Timothy Zhu, *Pennsylvania State University;* David Dion and Thomas Moscibroda,
Microsoft Azure; Ishai Menache, *Microsoft Research*

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the
17th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Kerveros: Efficient and Scalable Cloud Admission Control

Sultan Mahmud Sajal[1,3]   Luke Marshall[1]   Beibin Li[1]   Shandan Zhou[2]   Abhisek Pan[2]
Konstantina Mellou[1]   Deepak Narayanan[1]   Timothy Zhu[3]   David Dion[2]
Thomas Moscibroda[2]   Ishai Menache[1]

[1]*Microsoft Research*   [2]*Microsoft Azure*   [3]*Pennsylvania State University*

Abstract

The infinite capacity of cloud computing is an illusion: in reality, cloud providers cannot always have enough capacity of the right type, in the right place, at the right time to meet all demand. Consequently, cloud providers need to implement admission-control policies to ensure accepted capacity requests experience high availability. However, admission control in the public cloud is hard due to dynamic changes in both supply and demand: hardware might become unavailable, and actual VM consumption could vary for a variety of reasons including tenant scale-outs and fulfillment of VM *reservations* made by customers ahead of time. In this paper, we design and implement *Kerveros*, a flexible admission-control system that has three desired properties: i) high computational scalability to handle a large inventory, ii) accurate capacity provisioning for high VM availability, and iii) good packing efficiency to optimize resource usage. To achieve this, *Kerveros* uses novel bookkeeping techniques to quickly estimate the capacity available for incoming VM requests. Our system has been deployed in Microsoft Azure. Results from both simulations and production confirm that *Kerveros* achieves more than four nines of availability while sustaining request processing latencies of a few milliseconds.

1   Introduction

Cloud capacity appears to be limitless. However, in reality, cloud providers need to deal with the limitations of datacenters with a finite number of machines while respecting contractual service-level agreements (SLAs) that provide availability guarantees to customer VMs. These SLAs might be severely compromised if the provider runs out of resources. Additionally, users expect predictability: not being able to launch new VMs when required can critically impact a customer's business [31]. An admission-control system is thus necessary to ensure cloud providers do not *overcommit* resources, provide seamless elasticity, and are robust to capacity loss due to failures. This paper describes the design and implementation of *Kerveros*: a scalable and efficient admission-control system for Microsoft Azure.

Admission control in the cloud is hard because it needs to account for a variety of fluctuations in both supply and demand across a large number of workload and machine types. On the supply side, datacenter machines and racks regularly fail at scale and maintenance tasks like software updates may also require rebooting machines. The cloud provider needs to maintain high availability amidst both deterministic and stochastic events by keeping enough resources free to provide seamless failover if necessary. On the demand side, to facilitate predictability, cloud providers have recently introduced the notion of "reserved resources" (or capacity *reservations*) [4, 6, 7, 20, 35] to guarantee the availability of capacity in the future. With such reservations, customers pay to have the provider set aside resources that can be claimed later. Accordingly, admission-control systems need to accommodate both on-demand VM requests and reservations.

The goal of our cloud admission-control system is to ensure that it simultaneously achieves *high computational scalability* and *a low rate of SLA violations* while avoiding inefficient capacity usage (which can lead to negative margins). One possible approach to accomplish this is to re-use existing VM allocators [21, 43] to directly allocate space for reservations, maintenance, and potential tenant growth (scale-outs). However, such a "placeholder" approach is slow and inherently inflexible, since the capacity allocated to such placeholders cannot be immediately used for other incoming VM requests that can pack well in the reserved space.

Instead, our solution is based on an approximate *buffer* approach that avoids early binding of placeholders. At a high level, we maintain buffers of resources for handling failures, maintenance, growth, reservations, etc., and track the total number of remaining resources after accounting for currently running VMs *and* buffers. Implementing this idea involves several algorithmic challenges, such as reasoning about the capacity required by VMs with multi-dimensional resource demands (e.g., CPU, memory, disk), quantifying the impact of buffers on a large variety of possible VM requests (e.g., Azure has more than 1000 VM types; see Figure 3 for details),
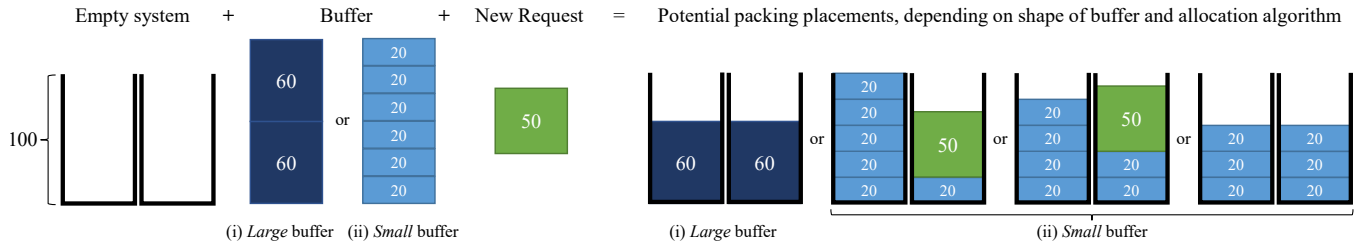
Figure 1: Consider an empty cluster of two machines with capacity 100 units each. Suppose one reservation with total size of 120 has already been accepted. We consider two cases: (i) the reservation consists of two VMs of size 60; (ii) the reservation consists of six VMs of size 20. A new single VM request of size 50 enters the system. Note that by solely taking the available capacity units into account (i.e., $200 - 120 = 80 \geq 50$), the system would accept the new request in both cases. However, when accounting for packing considerations, the request should be accepted only in case (ii). Further, if the underlying allocation algorithm was designed to load balance the VMs corresponding to a reservation, then the request should be rejected in both cases.

simultaneously accounting for buffers defined at different levels of the datacenter hierarchy (e.g., cluster vs. zone), and mimicking how the allocator would place VMs belonging to admitted reservations. Indeed, simple back-of-the-envelope buffer calculations might result in two undesired scenarios: (i) rejecting a request that can actually fit in the cloud, or worse, (ii) accepting a request at the cost of compromising reservation and availability guarantees; Figure 1 illustrates this scenario.

To address these challenges, we introduce a novel admission control mechanism which relies on *Allocable VM (AV)* counts, a bookkeeping technique for quickly determining the capacity available for an incoming VM request or reservation. The AV count, defined per VM type, quantifies the number of VMs that can fit in the inventory at a given time, and reduces the multi-dimensional problem into a formulation with a single dimension (the AV count). *Kerveros* exposes AV counts to the allocator, which can then accept or reject incoming VM or reservation requests based on a simple comparison (i.e., is the requested capacity less than the current AV count?). Importantly, this allows the allocator to respond to requests with high throughput and low latency, a critical requirement for extreme-scale VM allocation platforms [21, 43].

To enable these highly efficient capacity checks, we design the Conversion Ratio Algorithm (CRA), which allows *Kerveros* to quickly translate all buffer sizes to a common unit: the AV count of the incoming request's VM type. We supplement CRA with a data-driven Linear Adjustment Algorithm (LAA), which periodically emulates the allocator to reduce potential biases of CRA. We implement these algorithms in Azure's resource-allocation platform, while selectively reusing and enhancing existing allocator infrastructure (e.g., data stores, request handling agents, etc.). We further accelerate the algorithm with a caching layer that allows for incremental AV count updates.

Our results from both simulation and production measurements demonstrate that *Kerveros* sustains at least four nines of availability without any significant degradation

in request-processing throughput. Our admission-control strategy estimates the available capacity with less than 1% error at the 95th percentile. Importantly, this level of accuracy enables Azure to avoid capacity wastage, leading to high return on investment (ROI). We emphasize that for today's global-scale cloud providers, even a 1% improvement in such a capacity-efficiency metric can be worth 100s of millions of dollars in saved hardware expenditure, translating to sizable impact on the cloud provider's bottom-line margin.

To the best of our knowledge, this paper is the first to describe the design, implementation and evaluation of an admission-control system deployed in a large public cloud. Prior research on datacenter resource management (e.g., Protean [21] and Borg [43, 45]) focuses mostly on on-demand VM placement. The closest work available is Meta's RAS system [34], which partitions resources at the granularity of machines to different sub-organizations and periodically re-assigns machines across partitions by solving a mixed integer linear program. While this approach can suit a first-party workload with a modest number of partitions, it is less efficient for dynamic public-cloud workloads (§5).

In summary, our main contributions are:

- The design of scalable and efficient admission-control algorithms for a large and heterogeneous public cloud inventory (§3).

- A robust system design that separates the admission-control logic from the components that enforce it, allowing for latencies of a few milliseconds for admission and placement decisions (§4). Our system has been successfully deployed in Microsoft Azure.

- Our extensive evaluation using measurements from both simulations and production demonstrate that our design achieves scalable and accurate admission control (§5).

- Supplementary to this paper, we release a new trace that can be used by the research community to design and test different packing and admission-control algorithms: https://github.com/Azure/AzurePublicDataset.
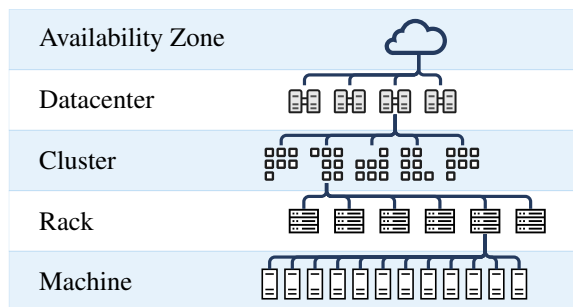
Figure 2: Cloud topology. A zone contains one or more datacenters that house a heterogeneous mix of clusters. Each cluster is comprised of homogeneous machines organized in racks. Racks provide isolation for fault tolerance.

2 Background and Motivation

Microsoft Azure is a large public geo-distributed cloud provider with a massive global footprint. Resources in Azure are organized into regions, each of which consists of one or more availability zones (Figure 2). Every zone contains one or more datacenters. Each datacenter is further divided into clusters and racks. Each cluster has a homogeneous set of machines (or servers); however, a zone can have a heterogeneous mix of clusters. As Figure 3 shows, zones can have tens of different hardware types. A zone can have hundreds of thousands of machines, while a cluster is much smaller (at most a few thousand machines).

Each zone has its own allocation service (or simply, *allocator*) that assigns VMs to physical machines. The assignment (or placement) considers a set of hard and soft constraints, which are evaluated sequentially for each VM. Examples of hard constraints include not violating the physical capacity of a machine and not running a VM type on hardware that does not support it. An example of a soft constraint is to prefer an already-occupied machine to increase packing efficiency [21].

In this section, we discuss the general resource management problem in Azure by focusing first on challenges arising from both dynamic and diverse demand patterns (§2.1), as well as fluctuations in the available compute supply (§2.2). These challenges, coupled with the fact that demand might exceed supply, motivate the need for a robust admission-control system; we outline its requirements in §2.3.

2.1 Demand Versatility

Azure has multiple different offerings for compute, which each impose specific requirements on the underlying allocation system.

VM requests. Azure has multiple hardware generations in its datacenter and offers over a thousand VM types. The majority of VM types are supported on most hardware generations, but some types require specialized hardware (e.g., GPUs for ML). We note that other major cloud providers like AWS and GCP also offer a large number of VM types [5, 19].
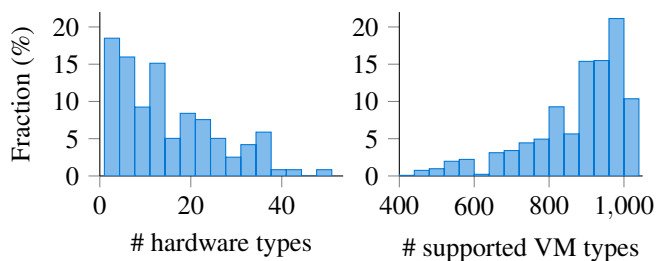

Figure 3: Histograms for number of different hardware types (left) and supported VM types (right) across Azure zones.

New VM types are regularly introduced as new hardware types and scenarios are onboarded to the cloud.

On-demand *VM requests* are the most common capacity consumption mode. A VM request specifies the type of the VM (which in turn determines the number of CPU cores, memory, disk, network requirements for the VM, and optional accelerators) and the VM's priority. Multiple VM requests may be grouped into a *tenant request*. A tenant request is accompanied by a tenant service model, where additional constraints can be imposed on the collection of VM requests (e.g., fault-domain requirements).

By default, VMs are spread across an entire zone. However, a tenant may request all its VMs be co-located within specific inventory boundaries such as a cluster or datacenter. A tenant using legacy Azure services can also be pinned to a single cluster; this means that such a tenant cannot create new VMs outside its cluster.

A tenant request succeeds only if all its associated VM requests are successfully allocated. There is no explicit SLA on allocation time, but it is desirable to keep these times as low as possible to ensure a fast and reliable deployment experience. The volume of VM requests is large: a zone can handle more than two million requests in a day. The demand pattern can be quite bursty: Figure 4 shows that a zone can easily receive a few thousand requests per minute. Hence, low latency and high throughput are critical requirements for the VM allocation service (see §2.3).

Higher-order consumption modes, such as Function-as-a-Service (FaaS or serverless computation [39]) are internally provisioned through VMs.

Customer scale-outs. Customers may decide to increase the number of VMs in their tenants; these are termed tenant *scale-outs*. Any allocation request, including a scale-out request, must be handled within milliseconds; the system can either accept or reject the scale-out request based on available capacity. Though there is no external acceptance SLA for scale-outs, Azure internally tracks the acceptance rate of these requests and attempts to sustain very high acceptance rates (at least 4 nines). Towards this end, the admission-control logic must explicitly reserve capacity within individual clusters for scale-out of pinned tenants (see §3.1). We note that
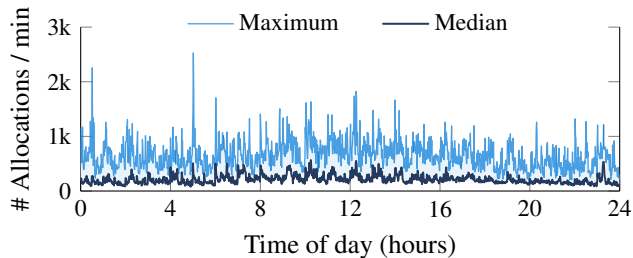
Figure 4: Number of high-priority allocations per minute for a zone over a day, aggregated over a 2-week period. Demand in the tail is bursty with large spikes.
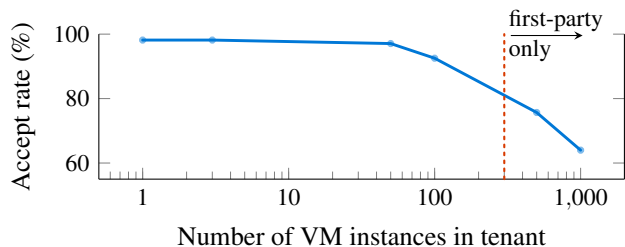

Figure 5: Tenant size versus allocator's acceptance rate. Data is obtained from one zone during a busy month. Tenant requests above 300 VMs are currently offered only internally.

Azure does not distinguish between scale-outs and new tenant requests for tenants that can span across an entire zone.

Reservations. Consider a scenario where a user wishes to terminate their VMs for the day, but expects to re-instantiate the VMs the following morning. Doing so through a regular on-demand tenant request might result in delays if capacity is not immediately available in the morning. As Figure 5 shows, this issue is exacerbated for large tenant requests. To address such scenarios, Azure offers the *reservation* option. A reservation request includes the VM type and the quantity. A reservation need not result in *actual* allocation of VMs; instead, it serves as a commitment from Azure to provide the desired number of VMs in the future, whenever the customer decides to materialize the reservation.

Reservations are active as soon as the request is accepted (i.e., we do not support reservation requests with future start dates). The user can terminate a reservation at any time. To support economy of scale, VMs that are created against a reservation cannot be pinned to a single cluster. This basic reservation model is offered by large public clouds as an *On-Demand Capacity Reservation* [4, 7]. We briefly discuss direct extensions of this basic reservation model, such as reservations with a future start date, in §6. Other reservation models are surveyed in §7.

2.2 Supply Fluctuations

Supply in Azure can change dynamically. In this subsection, we outline the situations that can trigger these changes.
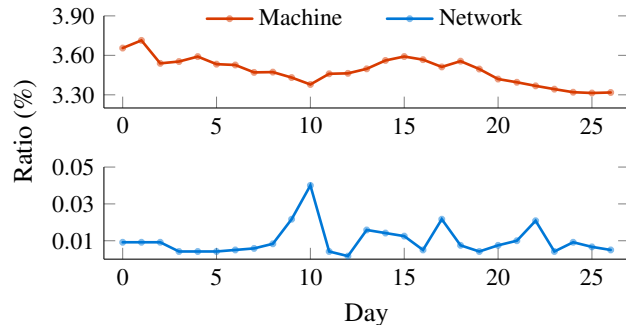

Figure 6: Network and machine failures across 30 days. The average network failure is around 0.01%, but each network failure can affect several machines. Outside of network failures, ~3.5% of machines are in a failed state every day.

Machine and other hardware failures. Figure 6 shows the failure frequencies for machines and network hardware over a period of 30 days. As the figure demonstrates, a network failure is much less probable. However, network failures impact more machines (e.g., a top-of-rack switch can affect around 50 machines). The cloud provider has to account for such failures and set aside capacity to migrate the VMs in affected machines to respect availability SLAs. It is thus crucial to both predict and have readily available mitigation for such failures.

Additional events affecting supply. Azure, like any other large cloud supplier, experiences other events that affect the available supply. One can roughly classify these events based on their predictability. *Deterministic* events include planned maintenance of machines (e.g., software updates) and decommissioning of hardware. *Stochastic* events include urgent security updates to patch a zero-day vulnerability, hotfixes to handle software bugs, and malicious attacks that overload the system.

2.3 Admission Control

We now describe the problem of admission control, and highlight requirements and design challenges for *Kerveros*. Admission control can be defined as the set of decisions that determine the acceptance of any form of resource request (new tenant, scale-out, reservation). Admission-control decisions need to take into account not only the present state of the zone, but also potential realizations of future state. Future state is influenced by a variety of factors including reservations that have been admitted but not yet materialized, potential hardware failures, tenant scale-outs, etc.

Challenges. As mentioned earlier, admission control in the cloud presents several challenges:

1. **Multiple elements affecting demand and supply.** *Kerveros* has to consider multiple elements with different characteristics: on the demand side, one has to consider

different modes of consumption (e.g., tenant vs. reservation requests) as well as potential tenant growth. On the supply side, we have different types of events (e.g., machine failures, rack maintenance) that affect inventory availability.

2. **Zone heterogeneity.** Beyond the inherent stochasticity in demand and supply, it is hard to determine how much capacity to set aside since not all machines can serve all VM types. For example, a new VM series might be served only on new hardware generations.

3. **Numerous VM sizes and fragmentation.** Unlike traditional resource allocation problems, which are often "single dimensional", admission control for the cloud needs to make decisions for VMs that request different types of resources (e.g., CPU, memory, disk). These multiple dimensions can cause fragmentation.

4. **Accounting for unclaimed capacity.** Reservations and other types of capacity protection impose an additional challenge since we would ideally like to "late bind" resources to such requests. This makes machine utilization an imperfect metric to determine if enough capacity is available for a new request.

Requirements. An admission-control system also has to meet several requirements to operate effectively within the context of a large cloud provider:

1. **Scalability.** Even for very large zone inventories, the decision of whether to accept or reject a tenant or reservation request has to be made within milliseconds.

2. **Respect admitted reservations.** All VMs that are requested against a previously accepted reservation should be fulfilled (i.e., assigned to machines) whenever the user opts to materialize the reservation.

3. **Availability.** The cloud provider must ensure high availability to customers. Accordingly, capacity must be set aside to facilitate the migration of VMs to other machines in case of machine failures or other events affecting supply.

4. **Elasticity.** An admission-control system should reserve capacity to allow tenants pinned to clusters to scale out.

5. **Accuracy.** The system should not reject requests when capacity is in fact available.

6. **Efficiency.** The system should set aside as little capacity as possible while satisfying the above properties (i.e., increase the return on investment).

Time scale of decision making. We observe that the above challenges and requirements necessitate that decisions be made at different time scales. For example, estimating how much capacity to set aside for failures or growth requires comprehensive data analysis that is inherently time consuming. On the other hand, the VM allocation service itself has to remain highly performant and process requests at low latency. Consequently, a natural design choice is to decouple the overall admission-control responsibility between fast- and slow-twitch systems. Accordingly, the allocator only executes

low-overhead capacity limit checks to determine whether to accept or reject a resource request (i.e., admission control *enforcer*)[1]. The entire logic for estimating the available capacity is performed by *Kerveros*, the focus of this paper, and is performed off the critical path.

3 Design of *Kerveros*

The high-level goal of *Kerveros* is to answer the following question: *How much capacity is available for an incoming tenant or reservation request?* As described earlier, this information is used by the allocator to accept / reject requests.

To address the complexities of demand and supply fluctuations (§2), our approach relies on two main concepts: (i) *buffers*, to specify a need for capacity; and (ii) *allocable VM count*, an auxiliary bookkeeping technique to quickly determine the available capacity for an incoming tenant or reservation request.

3.1 Buffers

A buffer is an accounting of capacity that must be protected for a specific purpose. The allocator treats this capacity as unavailable when admitting tenant or reservation requests. *Kerveros* supports three buffer types: (i) *Reservation buffers* to accommodate already admitted reservations; (ii) *Growth buffers* to accommodate existing tenant growth; (iii) *Healing buffers* to accommodate currently-running VMs that might need to be migrated in case of hardware failures.

As we detail below, we use appropriate counts of VM types to quantify the size of each buffer. Formally, a buffer is defined as a tuple (t, x), where t is the VM type and x is the number of VMs of that type that ought to be protected. The size of each buffer may change over time. For instance, a reservation buffer can become smaller as the customer gradually starts using VMs corresponding to that reservation. On the other hand, a healing buffer may become larger over time, e.g., as hardware ages and failures become more likely.

We make an important design choice for buffers: although capacity is protected, it is *not* mapped to specific physical machines. Buffers are defined at higher levels of the cloud hierarchy (e.g., cluster or zone) to reflect the amount of capacity that must be protected in aggregate at that level. The physical allocation occurs only when the protected capacity is needed to fulfill its purpose. For example, after hardware failures, healing buffer capacity can be used for migrating VMs from the affected machines. To ensure maximum utilization of resources, we use unclaimed protected capacity to offer spot VMs [2,3], which can be immediately preempted to free up capacity whenever buffer capacity is claimed.

---

[1]For a tenant request, the capacity limit check is immediately followed by an actual allocation of the VMs to physical machines. For reservation requests, the allocator performs only the limit check since placement is late-bound for reservations in our design.

We next provide more details on how *Kerveros* sets sizes of buffers of different types.

Reservation buffers. Setting reservation buffers is straightforward: the buffer size is set exactly according to the user-provided reservation requirement. That is, suppose reservation k requires x VMs of type t within a certain zone; *Kerveros* sets the respective zone-level buffer as $R_k = (t, x)$. When the user claims u VMs of that reservation, the buffer is updated to $R_k = (t, x - u)$.

Growth buffers. Growth buffers are used at a cluster level to account for the expansion of existing tenants pinned to each cluster. *Kerveros* defines a single growth buffer for each VM type. Intuitively, the buffer size should be proportional to the current consumption of that VM type. More specifically, for a given cluster c and VM type t, let x_{tc} be the current number of active VMs belonging to tenants pinned to cluster c. Then we set the corresponding growth buffer to $G_{tc} = (t, \alpha_{tc} x_{tc})$, where $\alpha_{tc} > 1$ is the effective growth rate. We use a ML model to set this parameter. In a nutshell, we consider a small set of possible effective growth rates (e.g., five values in the range between 1.03 and 1.1). The input features to the ML model consist of tenants' information (e.g., account IDs, VM lifetimes), hardware details (e.g., generation, SKU), and cluster-specific fragmentation details (intuitively, a cluster that is "badly" packed would induce higher effective growth rates); the statistics on fragmentation are obtained from the allocator. The ML model (implemented using XGBoost) is trained daily using historical growth data.

Healing buffers. *Kerveros* uses healing buffers to account for hardware failures and other events affecting supply (§2). For example, the protected capacity may be used to migrate VMs away from non-functional hardware. Since tenants can be constrained to specific clusters, healing buffers are maintained at the cluster level. Since full-machine VMs are the hardest to allocate (as they require a completely empty machine), the healing buffer is defined in units of full-machine VMs; formally, the buffer is of the form $H_c = (L, x_c)$, where L denotes the full-machine VM type, and x_c is the quantity.

We calculate x_c using a data-driven approach. The high-level idea is to set x_c in proportion to the hardware failure probability (i.e., the buffers should be larger if the failure probability is higher). More specifically, we would like to ensure that the total number of non-functional machines does not exceed the buffer size x_c with high probability; let p_c denote that probability (e.g., $p_c = 99.9\%$). Towards this end, we extract from historical failure data the empirical distribution of the total number of non-functional machines over a given period of time (e.g., a day). The total number accounts for a variety of events including machine and network failures as well as maintenance events. Then, x_c is simply derived as the p_c-percentile value of this distribution. *Kerveros* may add some slack to the obtained value (e.g., 10%) for clusters that are highly fragmented (as perceived

by the allocator). The choice of p_c is specific to the cluster, and depends on various factors. For example, *Kerveros* uses higher p_c values for clusters that have a high ratio of tenants pinned to that cluster, since such tenants have fewer fallback options in case of failures.

In rare cases, the healing buffer capacity is increased to allow completion of urgent software updates.

3.2 Allocable VMs

Kerveros uses the notion of allocable VM counts (AV counts) to reason about available capacity. Specifically, the AV count, $\mathbb{A}[t]$, gives the number of additional VMs of type t that can currently fit in the zone. For an incoming VM request of type t and demand x, *Kerveros* first calculates[2] $\mathbb{A}[t]$, and then the allocator rejects the request if $\mathbb{A}[t] < x$.

In this section, we describe how *Kerveros* calculates the AV counts across entire clusters and zones. The calculations are non-trivial because they require a *conversion* mechanism that accounts for buffers of different VM types t defined at different hierarchies of the datacenter (cluster versus zone). We note that other approaches that do not explicitly account for multiple resource dimensions might result in either under- or over-estimating the available capacity.

3.2.1 Overview

The high-level pseudocode of our AV count calculation is provided in Algorithm 1. Informally, the algorithm implements the following calculation:

$$\mathbb{A}[t] = \begin{bmatrix} \text{available capacity for} \\ \text{type } t \text{ across clusters} \end{bmatrix} - \begin{bmatrix} \text{buffers converted from} \\ \text{type } t' \text{ to type } t \end{bmatrix}.$$

The algorithm starts by initializing the AV counts, excluding buffers, and accounting only for the active VMs in the system (Step 1). Since certain buffers are defined only at a zone level, the algorithm then proceeds to transform them to cluster-level buffers (Step 2), so that all buffers can be analyzed at the same level of the cloud hierarchy. The remaining calculations are done at a cluster level, except a final aggregation step. The heart of the algorithm is converting buffers of other VM types into buffers of type t (Step 7) and then deducting them from the current AV count (Step 8). This conversion step is the subtle part of the algorithm. We term the full algorithm the *Conversion Ratio Algorithm (CRA)*.

3.2.2 CRA Algorithmic Details

We next describe each of these steps in detail.

AV count initialization (Step 1). The initialization step calculates $\mathsf{A}[t, c]$: the number of VMs of type t that can fit in cluster c, excluding buffers. Each VM type requires different quantities of the various compute resources (e.g., CPU, memory) available in a machine; thus these non-buffered AV counts are obtained by calculating the maximum

---

[2]In practice, we update the AV counts only periodically (every minute), and use caching to track the updated AV counts. See §4 for details.

Algorithm 1 CRA: Calculating AV Counts for a VM type t.

1: Calculate the AV counts excluding buffers: $\mathtt{A}[t',c]$ for all t'.
2: Distribute zone-level buffers to clusters.
3: **for** each cluster c **do** ▷ Calculate per-cluster counts.
4:     $a_c = \mathtt{A}[t,c]$
5:     Aggregate buffers per VM type.
6:     **for** each aggregate buffer (t',x) **do**
7:         Convert buffer from t' into type t: $\mathbb{C}(t' \rightarrow t,x,c)$.
8:         $a_c \mathrel{-}= \mathbb{C}(t' \rightarrow t,x,c)$ ▷ Deduct from current total.
9: Aggregate cluster counts for zone: $\mathbb{A}[t] = \sum_c a_c$.

number of VMs that can fit in each machine (considering all resource dimensions) and taking the sum over all machines in the cluster:

$$\mathtt{A}[t,c] = \sum_{\substack{\text{machine in} \\ \text{cluster } c}} \min_d \left\lfloor \frac{\text{Available resource } d \text{ on machine}}{\text{Required resource } d \text{ for type } t \text{ on machine}} \right\rfloor.$$

Going from zone- to cluster-level (Step 2). *Kerveros* temporarily breaks down the zone-level reservation buffers into cluster-level buffers, so that all buffers can be analyzed at the cluster level. The apportioning to cluster-level buffers is done by mimicking how the allocator would spread the VMs across multiple clusters when reservations are materialized. Considerations that are taken into account include load balancing, prioritizing newer-generation hardware for new VM types, and more. We omit details for brevity.

Aggregating cluster buffers per VM type (Step 5). Each cluster may have multiple buffers of the same VM type. We aggregate all buffers of the same type into a single buffer by summing their sizes. Intuitively, having a single buffer makes the conversion across types (discussed next) less lossy and more efficient.

Converting buffers into other VM types (Step 7). We convert a buffer from one type (t') to another (t) to estimate the impact that the original buffer has on allocations of type t.

One way to accomplish this is by using conversion ratios that encode the relative sizes of VM types t and t'. However, naïve conversion ratios have issues, since VMs have multi-dimensional sizes. To illustrate this, consider a simplified example with the following assumptions:
- Two requested resource types: CPUs and memory.
- Two VM types: large (2 CPUs, 4 GB) and small (1 CPU, 1 GB).
- Two machine sizes: M_1 (25 CPUs, 40 GB) and M_2 (25 CPUs, 25 GB).

Table 1 shows the counts of each VM type that can fit in one empty M_1 and M_2 machine, and also the counts after adding 10 and 20 small VMs using various counting methods. Simple conversion ratios based on resource ratios in isolation (CPU and RAM in Table 1) can lead to sub-optimal (yellow bars) or incorrect outcomes (red bars). This has repercussions on packing efficiency and SLA adherence: underestimates can

Machine State	Method	Large on M_1	Large on M_2	Small on M_1 & M_2
Empty	Actual	10	6	25
10 small	Actual	7	3	15
	CPU	5	1	15
	RAM	7	3	15
	CRA	6	3	15
20 small	Actual	2	1	5
	CPU	0	0	5
	RAM	5	1	5
	CRA	2	1	5

Table 1: Counts of large and small VMs that can fit in two machines M_1 and M_2 using various other conversion-ratio-based methods compared to the true count (Actual). CPU treats 1 small VM as $1/2$ a large VM, and RAM treats 1 small VM as $1/4$ a large VM.

strand resources (leading to fragmentation and worse packing efficiency) and overestimates can violate hardware constraints (leading to SLA violations). The optimal conversion ratio depends both on the machine size and the VMs already allocated (i.e., resources *remaining* on the machine).

We use the ratio of AV counts as a low-dimensional approximation instead. Formally, a buffer of type t' with size x is converted to a buffer of type t with size:

$$\mathbb{C}(t' \rightarrow t,x,c) = \left\lceil \frac{\mathtt{A}[t,c]}{\mathtt{A}[t',c]} \cdot x \right\rceil.$$

This works well in practice (empirical results in §5).

Example. To illustrate CRA, we return to the example in Figure 1. As a quick recap, we consider a single cluster with two empty machines, each with size of 100 units (for simplicity, we assume only a single resource dimension). The three VM types small (S), medium (M), and large (L) have sizes 20, 50, and 60 units respectively. A single medium-sized VM is requested – this request is rejected if $\mathbb{A}[M] < 1$. We first initialize the counts for our VM types (S, M and L):

$$\mathtt{A}[S,c] = 2 \cdot \left\lfloor \frac{100}{20} \right\rfloor = 10, \quad \mathtt{A}[M,c] = 2 \cdot \left\lfloor \frac{100}{50} \right\rfloor = 4,$$
$$\mathtt{A}[L,c] = 2 \cdot \left\lfloor \frac{100}{60} \right\rfloor = 2.$$

In both scenarios in Figure 1, there is a single incoming request of size 50 and buffers of 120 units total that need to be taken into account; the only difference between the scenarios is the VM type of these buffers:
- *Large buffers.* Assume that we have two large buffers. We convert these buffers to type medium, which yields $\mathbb{C}(L \rightarrow M,2,c) = 4$ medium VMs. This results in $\mathbb{A}[M] = 0 < 1$, so we reject the request.

$$\mathbb{A}[M] = \mathtt{A}[M,c] - \mathbb{C}(L \rightarrow M,2,c)$$
$$= \mathtt{A}[M,c] - \left\lceil \frac{\mathtt{A}[M,c]}{\mathtt{A}[L,c]} \cdot 2 \right\rceil = 4 - \left\lceil \frac{4}{2} \cdot 2 \right\rceil = 0.$$
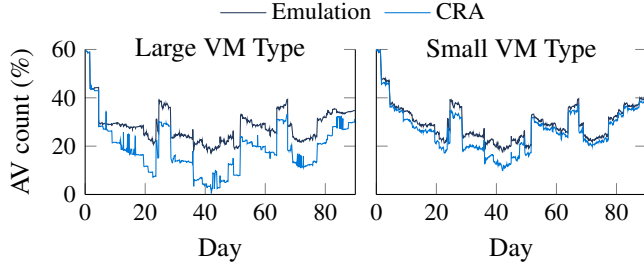
Figure 7: CRA versus emulation for two VM types, where the AV count is expressed as a percentage of the total capacity. Simulated results from a single trace.

- *Small buffers.* Assume that we have six small buffers. We convert these buffers to type medium, which yields $\mathbb{C}(S \to M, 6, c) = 3$ medium VMs. This results in $\mathbb{A}[M] = 1 \not< 1$, so we accept the request.

$$\mathbb{A}[M] = \mathtt{A}[M,c] - \mathbb{C}(S \to M, 6, c)$$
$$= \mathtt{A}[M,c] - \left\lceil \frac{\mathtt{A}[M,c]}{\mathtt{A}[S,c]} \cdot 6 \right\rceil = 4 - \left\lceil \frac{4}{10} \cdot 6 \right\rceil = 1.$$

In Appendix A, we provide a theoretical analysis of CRA under simplified assumptions; specifically, we prove that the conversion results in a bounded waste of resources.

3.2.3 Linear Adjustment Algorithm (LAA)

The Conversion Ratio Algorithm offers an efficient and scalable approach for obtaining the AV counts. Nevertheless, the output of this algorithm might sometimes be fairly inaccurate. The main reasons for inaccuracy are potential fragmentation issues while dealing with conversion between multi-dimensional VM types and not explicitly modeling how the VMs would be placed using the allocator (e.g., erroneously assuming the rightmost outcome in Figure 1 and rejecting the new request).

The above limitations can be mitigated if *Kerveros* could emulate the placing of the different buffers and filling up of the inventory by allocating VMs using the allocator. This is the main idea behind the *Linear Adjustment Algorithm (LAA)*. Since such emulation is compute-intensive and time-consuming, we run it periodically (every 30 minutes) and in isolation, i.e., without interfering with the handling of customer requests (more details in §4.1.2). We then use the emulation result to calibrate CRA's output.

To see how the emulation output should be accounted for, we compare its output to CRA's output. Figure 7 shows a time series of emulation and CRA's output for two different VM types. We observe that the gap between the two methods is steady at times, but is spiky at other times. The LAA should account for both these phenomena.

Formally, for any VM type t, let $\mathbb{A}'[t]$ and $\mathbb{E}'[t]$ be the AV counts obtained by CRA and the allocator emulation at the time when the emulation was run last, respectively. We then
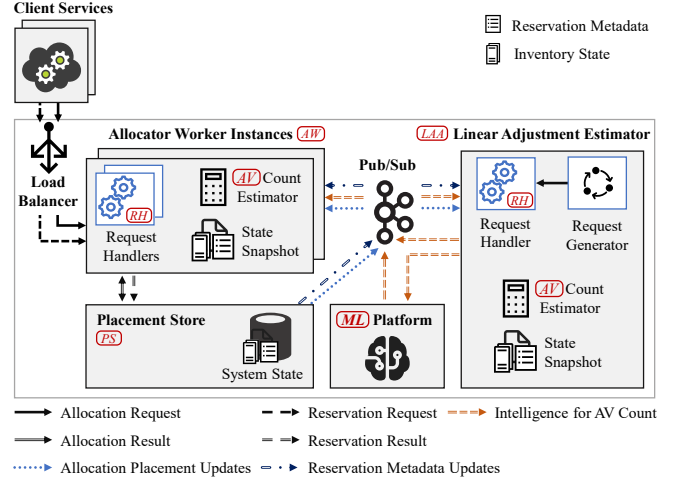

Figure 8: *Kerveros* system architecture.

adjust the current VM count estimate $\mathbb{A}[t]$ (result of CRA) to obtain a new estimate:

$$\mathbb{A}_{\text{adjusted}}[t] := \beta_1^t \cdot \mathbb{A}[t] + \beta_2^t \cdot \mathbb{A}'[t] + \beta_3^t \cdot \mathbb{E}'[t] + \beta_4^t, \quad (1)$$

where the coefficients β_i^t are learnt using standard linear regression, with the emulation outputs serving as ground truth for training. β_2^t and β_3^t incorporate information on the gap between ground-truth and CRA in the previous time step, and β_4^t models the constant gap between the two methods. In §5, we show that LAA substantially decreases AV count error, while maintaining scalability.

4 System Implementation

In this section, we first describe *Kerveros*'s various microservices that allow it to respond to allocation and reservation requests in an availability zone and enforce its admission-control logic in an efficient and scalable manner (§4.1). We then discuss tradeoffs that arise from various design decisions made in our implementation (§4.2).

4.1 Architecture

Kerveros's microservices work in concert to handle allocation and reservation requests, estimate AV counts, execute emulation runs for AV count adjustment, persist state when allocations and reservations are accepted by the system, and train ML models as required (Figure 8). The microservices primarily use a distributed publish-subscribe (pub-sub) platform [42] to transfer state among themselves.

4.1.1 Allocation Worker Instances

Multiple *stateless* allocation worker instances $\widehat{AW}$ are used to handle both reservation and tenant allocation requests. An allocation worker instance is a process typically running on a dedicated machine. Each instance has two types of agents:

request-handling (RH) agents to serve requests, and an AV count estimation agent to periodically compute the number of allocable VMs. The number of worker instances and RH agents deployed in a zone is configured based on the request demand in the zone and size of the allocation inventory.

Request handlers $\boxed{RH}$**.** A RH handles both reservation and tenant requests.

Reservation requests. For a reservation request, the RH performs a zonal admission-control check. The zonal check evaluates whether there is enough capacity in the zone by comparing the number of requested VMs with the zonal AV count. All buffer types (reservation, healing, and growth) are considered for this check. On success, the reservation metadata is persisted to the placement store (see below).

Tenant requests. For new tenant requests, the same zonal admission-control check is executed. The agent then proceeds to filter and sort the inventory machines for each requested VM, following a series of hard and soft constraints respectively, to assign a specific machine for each VM. One of the filtering steps involves performing cluster-scope admission-control checks to ensure clusters have sufficient capacity for the requested VMs.

The buffers used in the admission-control steps are adjusted based on the nature of the request and scope of the check. For example, at cluster scope, only healing and growth buffers are considered for new tenants when computing AV counts. Similarly, only healing buffers are considered for tenant growth requests. As another example, all admission-control checks are skipped for reservation-backed VM requests (i.e., requests for VMs against an already-accepted reservation) and healing requests, since these requests have already been accepted (either as part of the reservation or as part of the original VM / tenant request before hardware failure).

On success, the "VM → machine" mapping is persisted in the placement store.

AV count estimator $\boxed{AV}$**.** The AV count estimator executes both CRA (§3.2.2) and the linear adjustments (§3.2.3). To avoid adding latency to the RH response time, the estimator is implemented as a separate agent off the critical path of the RH. It runs in a tight loop (every minute), and updates the RHs with new AV counts through in-memory state transfer.

To obtain the AV counts, the estimator requires information about machine occupancy and health, reservation metadata, adjustment coefficients, and reservation-VM maps for VMs allocated against reservations. Each estimator learns about this information through the pub-sub platform. This information is organized through multiple pub-sub topics. Given the distributed nature of the platform, each estimator works with a somewhat stale view of the inventory and reservations (we discuss the impacts of this in §4.2).

4.1.2 LAA Instance

The LAA instance performs periodic emulation to obtain more accurate AV counts (§3.2.3). To do so, it listens to the same pub-sub topics as the worker instance, but does not handle customer requests and does not persist any results to the placement store. It starts each emulation run from a snapshot of the entire inventory (with buffer information). It then creates allocation requests corresponding to the buffers and allocates them using the snapshot of the inventory state. These results are stored as local in-memory modifications on the initial inventory snapshot. Once the buffers are allocated, the LAA instance computes more accurate AV counts for each VM type from the remaining available capacity by repeatedly allocating (till failure) and deallocating VMs for each type. We note that these operations do not interfere with the critical path of real request handling or admission-control enforcement. Finally, the LAA instance sends relevant estimation data to the ML platform, which runs the linear regression required to tune the β_i parameters in Equation 1; the training is performed at a coarser time granularity (every day, using a week's worth of emulations results).

4.1.3 Offline ML Platform

The offline ML platform $\boxed{ML}$ performs relevant ML training tasks; its output is consumed by the AV count estimators. In addition to updating the LAA coefficients, the platform provides the predictions required for buffer management (e.g., determining the effective growth rate for a cluster).

4.1.4 Placement Store

A *stateful* microservice, called the placement store $\boxed{PS}$, persists the results of the computations performed by the RH agents in the worker instances.

Correctness of the VM→machine assignment is critical since incorrect assignments can lead to VM start failures and violation of explicit guarantees provided to customers. Hence, the PS validates each VM→machine assignment to check for conflicting assignments made by other concurrent RH agents, and returns a `retry` if validation is not successful. This ensures that the RH's stale view of the inventory does not lead to correctness issues. PS uses fine-grained machine-level locking to allow results from multiple machines to be checked and persisted concurrently.

On the other hand, validating admission-control checks at a zonal level with concurrent allocation workers would require the PS to take a global lock on the entire inventory. This would severely compromise scalability. Hence, the PS does not perform any *global* capacity checks, which means that buffer enforcement is not guaranteed to be correct for every request. We note however that temporary reductions in buffer capacity are rarely seen in practice and are more acceptable (see discussion below).

4.2 Practical Considerations

We now briefly discuss the implications of some of these design decisions.

Dependence on underlying allocator. While we build and design *Kerveros* within the context of Azure, we note that other major public clouds face similar challenges (e.g., a multitude of VM types and consumption modes, hardware failures, efficiency requirements). Moreover, *Kerveros*'s algorithms are agnostic to the details of the underlying allocator (e.g., exact hard and soft constraints enforced).

Caching. As described in §3.2, AV counts are computed for every machine separately and then aggregated to derive the allocable VMs that can fit in the cluster. Instead of computing AVs for every machine from scratch, the AV counts are maintained in an in-memory cache per machine and per cluster. This cache is initialized when the AV count estimator starts, and then is updated incrementally only for the machines that are modified because of changes in machine occupancy or health. Caching the AV counts in this manner massively reduces the computation overhead because the number of machines altered between consecutive AV count updates is orders of magnitude smaller than the total number of machines in the inventory. The periodic nature of AV count computation (as opposed to calculating AV counts for every request) might lead to using protected capacity.

Optimistic concurrency. As noted earlier, *Kerveros* uses optimistic concurrency control for better scalability, which allows allocation worker instances to see stale versions of state in the system. This is a limitation, since it allows the possibility of accepting multiple requests eating into the "same" protected capacity. This can occur, for example, when multiple worker instances accept requests or reservations simultaneously while not being aware of each other.

Due to dynamic churn (in particular VM shutdowns) in the workload (Figure 9), usages of protected capacity are temporary and rarely result in SLA violations. Running out of capacity due to stale AV counts or optimistic concurrency is rare, but if it happens, we have multiple knobs for mitigation, such as temporarily eating into healing buffers, evicting internal non-critical workloads, restricting the creation of new VMs, and migrating VMs to reduce fragmentation.

5 Evaluation

In this section, we seek to answer the following questions:

- Does *Kerveros* perform better than relevant baselines on the following dimensions: packing efficiency, SLA violations, and runtime?
- How well do *Kerveros*'s various optimizations work to count AVs accurately (e.g., LAA)?
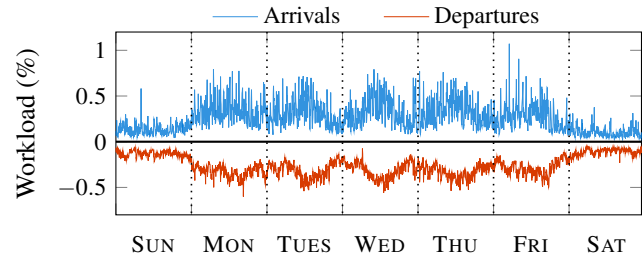- How does *Kerveros* trade off accuracy vs. compute effort?

Figure 9: *Churn* for high-priority workloads in a single zone. Average is computed over 5-minute intervals, and computed over a 2-week period. Workload is weighted by the size of VM requested and normalized by the total volume of the zone. That is, roughly an average of 0.3% of total workload arrives and exits the zone every 5 minutes.

5.1 Experimental Setup

In this section, we outline the metrics of comparison, the baselines, as well as details on how we run experiments in production and simulation.

Metrics. We use the following metrics to measure *Kerveros*'s effectiveness:

- **Packing efficiency.** This metric estimates the capacity wasted by resource fragmentation due to inefficient packing. To measure packing efficiency, we temporarily fill the system with full-node VM requests, and set packing efficiency to be the final system load (including unused buffers) as a percentage of total cores (proxy). We choose full-node VM requests since they are generally the hardest to place.
- **Scalability.** The runtime (latency) of *Kerveros*'s AV count estimator.
- **SLA violations.** The proportion of requests where SLAs (machine failures, growth, reservation) are violated.
- **AV error.** The deviation in AV counts between offline emulation (which serves as an oracle) and other approaches. We use this metric to study the effectiveness of *Kerveros*'s approximate AV counting approaches.

Baselines. We compare *Kerveros* against the following:

- **Offline emulation (Oracle).** In the event of a tenant or reservation request, the allocator first makes temporary allocations for all unused buffers in the system. The allocator then checks the feasibility of accepting the current request; if there are not enough resources to satisfy the request, it is rejected. All temporary allocations for unused buffers are then discarded. This is slow but gives an accurate AV count, which can then be used as ground truth to estimate the AV error described above.
- **Placeholder (PH).** All buffers are allocated and assigned to physical machines at the time of request (or update). For example, an accepted reservation is allocated all of its required resources at admission time.
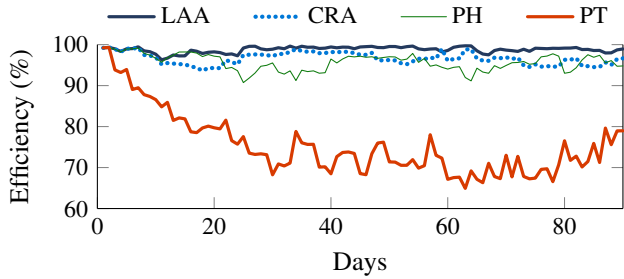
Figure 10: Comparison of packing efficiency across various baselines against the inventory partitioning algorithm (PT). The results are computed over a single trace. As can be seen, PT does not work well with our workloads.
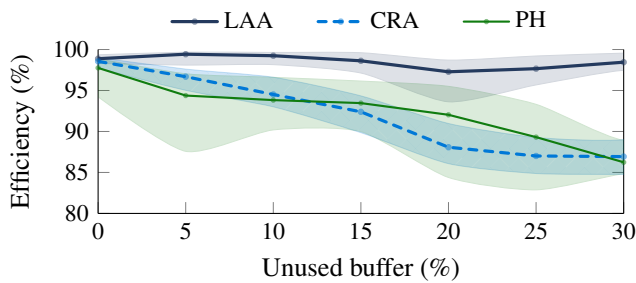

Figure 11: Packing efficiency vs. unused buffer size. We show the median and 25/75th percentiles, aggregated over all traces.

- **Inventory partitioning (PT).** Each buffer is allocated its own set of dedicated machines, enough to satisfy demand. The partitioning approach is similar to the method used by RAS [34], where *machines* are dynamically partitioned as new reservations arrive.

Production. Azure collects telemetry on different aspects of VM resource management in an internal data analytics platform. We use this data to measure healing and growth failures as well as latencies (§5.2). All production metrics are gathered for a period of one month in 2022.

Simulator. Production data is not sufficient to provide a full evaluation of *Kerveros*. In particular, comparison against other approaches requires a simulator since the baselines we consider are not deployed in our production setting.

To this end, we use our event-driven simulator to test various aspects of the entire admission control system (including both the allocator and *Kerveros*). Due to the allocator's relative complexity, the simulator includes only a lightweight version, which supports the key constraints of the allocator logic. The simulator handles both tenant and reservation events (arrival, departure, and updates). Despite supporting a subset of the allocator's logic, the simulator provides an excellent approximation of the system in production and is able to scale adequately to large inventories. We have extended the simulator to support the above baselines.
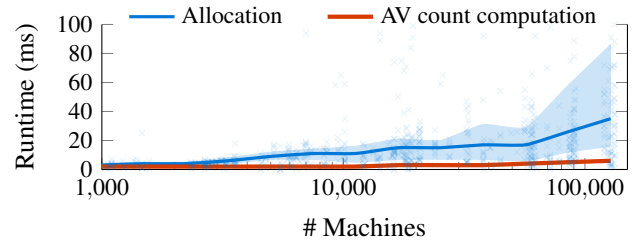

Figure 12: Latency (25/50/75th percentiles) versus inventory size, extracted from Azure over a month. Randomly-sampled allocation times are included for further context. End-to-end allocation latency is more significantly affected by inventory size compared to the AV count computations.

Simulation traces. We run simulation experiments on twenty traces that include both tenant and reservation requests. The traces use historical production data for tenant requests. Reservations, on the other hand, are a relatively new offering without significant traffic yet. Consequently, we synthetically add reservations to augment the historical data. The reservation characteristics, such as VM type and size, are extrapolated from real reservations.

5.2 End-to-End Experiments

Our goal here is to show that *Kerveros*'s CRA and LAA approaches outperform other baselines along three axes: packing efficiency, scalability, and SLA adherence.

5.2.1 Packing Efficiency

Figure 10 shows a timeline view of packing efficiency for one of the traces. The partitioning (PT) approach of splitting by machine is inefficient for our workloads: PT might work well in other limited settings (e.g., small number of total tenants). However, our general setting includes a large number of requests which require many fractions of machines. We therefore exclude PT from the rest of the analysis.

We now further explore the remaining baselines and aggregate results over all traces. Figure 11 illustrates how unused buffer sizes influence packing efficiency. As the unused buffer size increases, CRA suffers from accumulated rounding errors, and the PH algorithm can lock into sub-optimal packing decisions, resulting in inferior packing efficiency. LAA can compensate for rounding errors, and sustains high packing efficiency even with large unused buffers.

5.2.2 Scalability

Figure 12 shows the latency of the AV count computation for various inventory sizes. For reference, we also show the allocator's latency for a single VM. We observe that *Kerveros*'s approximate AV counting algorithm scales well with inventory size, taking less than ten milliseconds even for inventories of over a hundred thousand machines. Approximate AV counting is cheap because the underlying computation is proportional to the number of VM types; in
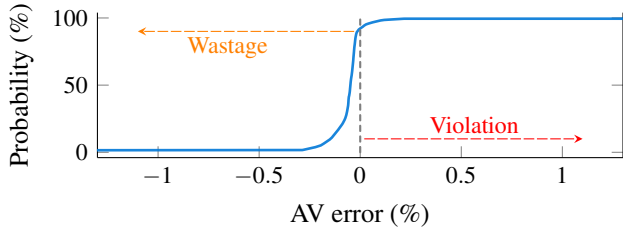
Figure 13: Cumulative density function (CDF) of AV error from LAA. SLA violations can occur when the AV error is positive (i.e., the algorithm overestimates AV counts, or asserts that there is more available capacity than reality).

(a) Healing. (b) Growth.

Figure 14: The impact on availability due to healing and growth failures. Note that these values are not weighted by time, but represent a normalized count of all failures within each day. The actual loss of availability to each customer is less than what is indicated above. This historical data is aggregated over the entire cloud for a month in 2022.

contrast, the allocator runtime depends on the inventory size (since the allocator has to choose the most suitable machine for allocation). Overall, the results indicate that *Kerveros*'s AV-based approach is not the computational bottleneck.

5.2.3 SLA Violations

We also want to verify that *Kerveros*'s approach results in a low number of SLA violations. We verify this in both production and simulation.

Reservation. Figure 13 shows the cumulative density function (CDF) of AV errors obtained from simulation. When the AV error is positive, *Kerveros* overestimates the available capacity. This behavior might lead to reservation SLA violations in some extreme cases (for example, if all users' reservations, healing buffers, and growth buffers are claimed almost simultaneously). As the LAA curve shows, *Kerveros* reserves sufficient capacity to cover all promised resources around 86% of the time; *Kerveros* requires less than 1% additional capacity to satisfy requests about 99.7% of the time. With typical workload churn, we expect to obtain this additional capacity promptly.

Healing. Figure 14a shows production data over a month for instantaneous SLA violations due to healing failures.
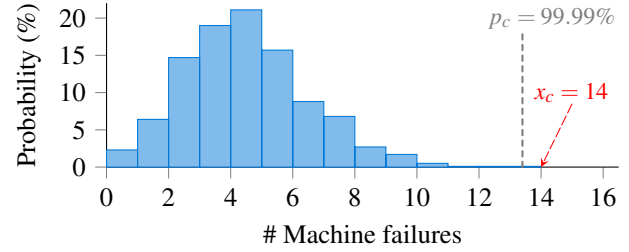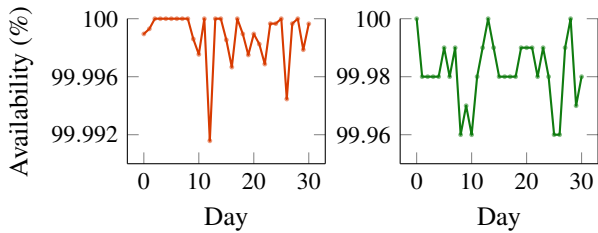

Figure 15: Illustration of healing buffer size, where $x_c = \lceil 13.4 \rceil = 14$ machine-equivalent buffers are prepared to satisfy the desired 99.99% SLA.
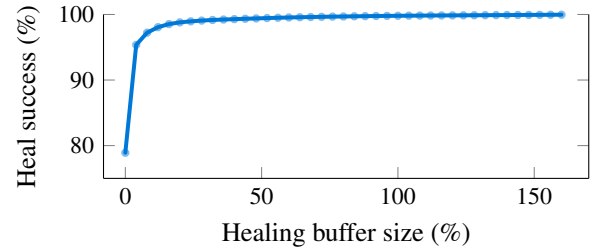

Figure 16: Tradeoff between healing buffer size and healing success rate.

We observe that *Kerveros* is able to sustain four nines of availability through the entire month by appropriately sizing healing buffers.

As Figure 15 shows, we calculate the healing buffer size based on a data-driven method (see §3.1). To better understand the tradeoff between packing efficiency and SLA adherence quantitatively for healing, we show how the healing buffer size influences the healing success rate based on data over a month in 2022, as shown in Figure 16. The x-axis shows the ratio of the buffer size compared to the size used in production, where 100% healing buffer size corresponds to the current production's decisions. When the ratio is zero, *Kerveros* does not reserve any healing buffers, but can still heal VMs affected by hardware failures if enough resources are coincidentally available in the cluster.

Growth. Figure 14b shows production data on instantaneous growth failures for the same month. We observe that growth is supported more than 99.9% of the time.

5.3 Deep Dive on AV Counting Algorithms

We now further evaluate the various components used to calculate AV counts.

Improvements from linear adjustment. Figure 17 shows the AV error of CRA (top) and LAA (bottom) versus the size of unused buffers. The estimation error with CRA increases with the unused buffer size, as fragmentation is amplified. LAA is more robust by fixing biases periodically (§3.2.3).
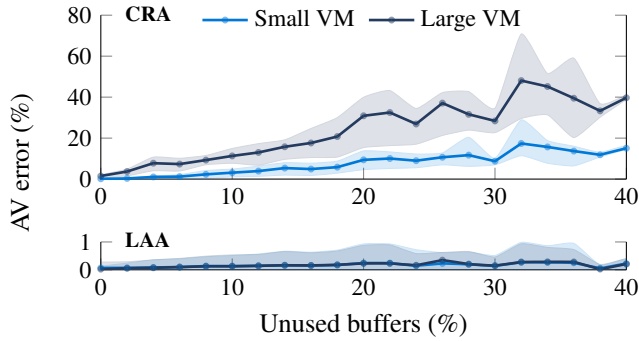
Figure 17: AV errors when calculating AV counts for two VM types (small and large) versus unused buffer size. The top graph, with average and 25/75th percentiles for CRA, shows CRA performing increasingly worse as unused buffer grows. LAA with an adjustment frequency of 30 minutes obtains small error even as the unused buffer percentage becomes large. We show 95th percentiles for LAA for emphasis.
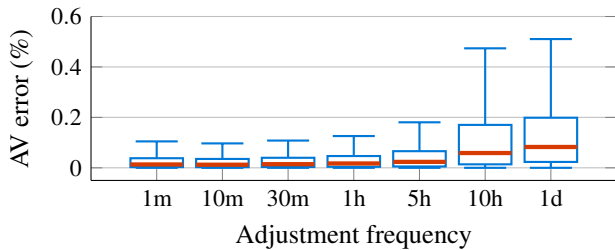

Figure 18: AV errors versus adjustment frequency in LAA. As we increase the frequency of adjustment, error is reduced at the expense of additional computation. In the *x*-axis, 'm' stands for minute, 'h' for hour, and 'd' for day.

Adjustment frequency. Figure 18 shows how the AV error changes with the frequency of LAA emulation. The figure shows that emulating every few minutes does not drastically reduce AV error. Consequently, we choose an emulation frequency of 30 minutes. Interestingly, we find that even with just one adjustment a day, the LAA algorithm still performs better than the basic CRA approach.

6 Discussion

We briefly discuss additional considerations relevant to *Kerveros*.

The public cloud eco-system. Modern public clouds have a large number of components that interact with each other, making multi-faceted decisions related to economically incentivizing usage, managing quota, provisioning capacity, etc. For example, special permissions or quotas are needed for very large customer demands; similarly, discounts are often offered to incentivize prominent customers. Such decisions are made on a slower time scale, often involving humans in the loop, and influence the inputs to *Kerveros*.

Constraints beyond capacity checks. The decision of admitting a tenant request is complex, and involves a variety of constraints beyond capacity checks (e.g., fault domains, locality). However, when *Kerveros* reserves buffers (healing, growth, reservations), it can afford to take a simplified view, relying on the economy of scale (large inventory, workload churn) and mitigation actions as a last resort. Accordingly, the basic CRA algorithm does not account for the entire set of constraints. Nonetheless, CRA is supplemented by LAA. LAA in turn strongly relies on emulating the allocator's logic, which does account for multiple preferences and constraints and uses realistic tenant requests against unused reservations. We note that the actual tenant requests against reservations can only specify a limited set of constraints by design (e.g., limit the maximum number of fault domains). The net effect is that *Kerveros* has proven to be reliable in production, despite the simplifying assumptions.

Priorities. While some business priorities are enforced at higher layers, allocation requests to *Kerveros* can have different priorities based on preemption level: higher-priority requests can preempt lower-priority ones to grab capacity [21, 45]. However, all buffers (reservation, growth, healing) are maintained only for the highest (non-preemptible) priority. Regardless of their priority, *Kerveros* handles all tenant and reservation requests on a first-come-first-serve basis.

Predictive modeling. Having a large percentage of unused buffers impacts resource efficiency; running preemptible workloads (e.g., spot VMs) on unused buffers is a partial mitigation. Currently, we avoid setting aside capacity in anticipation for future new customers. Incorporating ML models into *Kerveros* to predict resource usage and utilize the unused capacity even more aggressively is an interesting future direction.

Comparison to early binding. As an alternative to *Kerveros*'s late-binding approach, an early-binding or placeholder approach that exploits the existing allocation system to allocate buffers when needed (e.g., when a reservation request is accepted, or when a machine failure occurs) may seem like a more natural solution. Our evaluation of *Kerveros* against the baseline placeholder solution (PH) in §5 showed that this strategy suffers from worse packing efficiency. In addition, it introduces other various complications:

- Early binding requires lock-in of VM configuration parameters (e.g., VM type) when not necessary yet, reducing flexibility on the clients' side.
- Buffers like the healing buffer need to have their sizes updated regularly, which is more inefficient with the early-binding approach.
- Early binding makes it harder to support over-subscription using spot VMs or harvest VMs.

- Early binding might require complex migration logic to move "placeholder" VMs between machines to pack VMs efficiently on the available physical machines.
- The early-binding approach incurs an additional caching cost, even if live migrations are free, i.e., the corresponding caches that reference the source and destination machines for each placeholder migration need to be invalidated.

Reservations with future start dates. *Kerveros* only supports reservations starting immediately. To guarantee reservations starting at future time t, we either need accurate estimates of the capacity at t (high risk), or we can reserve capacity now and hold it until t (high cost). Both approaches have significant issues that become worse as t is pushed out further into the future; these issues are also exacerbated by larger reservations (both in terms of the number and size of the reservation). It might be possible to estimate future capacity using a probability distribution, but we expect the accuracy to be poor without a reservation end time. Alternatively, rather than requesting a specific start and end time for a reservation, a user could specify a reservation with a *demand profile* that indicates the desired usage and expected growth over time.

7   Related Work

Kerveros builds on a rich line of previous work on admission control and reservations in the cloud.

Admission control. Admission control is a broad topic that has been studied in a variety of contexts such as cloud systems [9, 16, 26, 27, 29, 43, 45, 51, 52, 55], computer networks [8, 11, 18, 28, 30, 36, 44, 46, 47, 53], cellular networks [37, 50], mobile edge computing [1, 23, 24], real-time database systems [12, 22, 32, 38, 48], distributed systems [13–15, 41, 49, 54], and caching systems [10, 33, 40], among others. Each domain provides unique challenges and requirements that advocate for custom solutions to be developed. This work focuses on datacenter-scale VM admission control with support for VM reservations. To the best of our knowledge, there is no published work in this space that explicitly covers availability, scalability, and efficiency considerations. The bulk of the related papers in this space is centered more around VM allocation and placement, and admission control is addressed only at a high level. Similarly, reservations are a relatively new offering and are not directly addressed. For example, Protean [21] describes Microsoft's rule-based zonal allocator, while focusing on systems enhancements to reduce latencies. However, Protean does not address either the problems of admission control or support for reservations. Google's Borg [43, 45] scheduler introduces efficient packing and machine-sharing techniques to achieve high resource utilization, but admission control is only briefly described in terms of a quota system, and managing the fragmentation and quota allocation of the admission control is outside the scope of the paper. In terms of reservations, Borg uses the concept of Borg Allocs, which are akin to the placeholder approach that we compare against.

Reserving cloud resources. The (VM) reservations we consider in this paper are a relatively new consumption mode that has been introduced to provide predictability to customers. Note that this mode is different from reserved instances (RIs) [6, 7, 17, 20, 35], under which customers make a 1-3 year *commitment* in return for a significant discount over on-demand offerings. For reservations in datacenters, we are only aware of Meta's RAS [34] system, which manages user reservations at the granularity of a server. RAS works by partitioning machines and dynamically migrating machines between partitions, guided by a mixed-integer linear program. While the partitioning approach works well for Meta's internal services, it is not well-suited for our public cloud setting (§5).

Resource reservations have also been proposed for internal big data analytics systems (e.g., [16, 25]). However, the provider there has more information about the jobs (e.g., job duration and deadline) and can better create an explicit resource allocation plan over time.

8   Conclusion

This paper describes the design, implementation, and evaluation of *Kerveros*, an admission-control system deployed in a large public cloud. Our design accurately retains capacity for hardware failures, tenant scale-out, and reservations while being computationally scalable to large inventories and peak loads. *Kerveros* can be extended to support additional scenarios such as improving margin efficiency through reservation overbooking and supporting enhanced reservation semantics (e.g., reservations with a start date).

Acknowledgements

References

[1] Nadine Abbas, Wissam Fawaz, Sanaa Sharafeddine, Azzam Mourad, and Chadi Abou-Rjeily. SVM-based Task Admission Control and Computation Offloading using Lyapunov Optimization in Heterogeneous MEC Network. *IEEE Transactions on Network and Service Management*, 19(3):3121–3135, 2022.

[2] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM Transactions on Economics and Computation (TEAC)*, 1(3):1–20, 2013.

[3] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, et al. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[4] AWS. EC2 Capacity Reservations. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-capacity-reservations.html, 2023.

[5] AWS. EC2 Instance Offerings, 2023. https://instances.vantage.sh.

[6] AWS. EC2 Reserved Instances. https://aws.amazon.com/ec2/pricing/reserved-instances/, 2023.

[7] Microsoft Azure. On-demand Capacity Reservations. https://azure.microsoft.com/en-us/reservations/, 2023.

[8] Sihem Bakri, Bouziane Brik, and Adlen Ksentini. On Using Reinforcement Learning for Network Slice Admission Control in 5G: Offline vs. Online. *International Journal of Communication Systems*, 34(7):e4757, 2021.

[9] Gaurav Baranwal and Deo Prakash Vidyarthi. Admission Control in Cloud Computing using Game Theory. *The Journal of Supercomputing*, 72(1):317–346, 2016.

[10] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, 2017.

[11] Pablo Caballero, Albert Banchs, Gustavo De Veciana, Xavier Costa-Pérez, and Arturo Azcorra. Network Slicing for Guaranteed Rate Services: Admission Control and Resource Allocation Games. *IEEE Transactions on Wireless Communications*, 17(10):6419–6432, 2018.

[12] M Carey, Sanjay Krishnamurthi, and Miron Livny. Load Control for Locking: The Half-and-Half Approach. In *Proc. of 9th Symp. on Principles of Database Systems*, 1990.

[13] Huamin Chen and Prasant Mohapatra. Session-based Overload Control in QoS-Aware Web Servers. In *Proceedings of Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 516–524. IEEE, 2002.

[14] Xiangping Chen, Prasant Mohapatra, and Huamin Chen. An Admission Control Scheme for Predictable Server Response Time for Web Accesses. In *Proceedings of the 10th international conference on World Wide Web*, pages 545–554, 2001.

[15] Ludmila Cherkasova and Peter Phaal. Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Transactions on Computers*, 51(6):669–685, 2002.

[16] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-Based Scheduling: If You're Late Don't Blame Us! In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.

[17] Alibaba Cloud ECS. How to Use Alibaba Cloud Reserved Instances to Reduce Costs. https://www.alibabacloud.com/blog/how-to-use-alibaba-cloud-reserved/instances-to-reduce-costs_595237/, 2023.

[18] Domenico Ferrari and Dinesh C Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, 1990.

[19] GCP. Compute Engine Instance Offerings, 2023. https://gcpinstances.doit-intl.com.

[20] GCP. Reservations of Compute Engine Zonal Resources. https://cloud.google.com/compute/docs/instances/reserving-zonal-resources, 2023.

[21] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020.

[22] Hans-Ulrich Heiss and Roger Wagner. *Adaptive Load Control in Transaction Processing Systems*. Universität Karlsruhe. Fakultät für Informatik, 1991.

[23] Dinh Thai Hoang, Dusit Niyato, and Ping Wang. Optimal Admission Control Policy for Mobile Cloud Computing Hotspot with Cloudlet. In *2012 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 3145–3149. IEEE, 2012.

[24] Jiwei Huang, Bofeng Lv, Yuan Wu, Ying Chen, and Xuemin Shen. Dynamic Admission Control and Resource Allocation for Mobile Edge Computing Enabled Small Cell Network. *IEEE Transactions on Vehicular Technology*, 71(2):1964–1973, 2021.

[25] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, 2016.

[26] Kleopatra Konstanteli, Tommaso Cucinotta, Konstantinos Psychas, and Theodora Varvarigou. Admission Control for Elastic Cloud Services. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 41–48. IEEE, 2012.

[27] Kleopatra Konstanteli, Tommaso Cucinotta, Konstantinos Psychas, and Theodora A Varvarigou. Elastic

Admission Control for Federated Cloud Services. *IEEE Transactions on Cloud Computing*, 2(3):348–361, 2014.

[28] J-Y Le Boudec. Application of Network Calculus to Guaranteed Service Networks. *IEEE Transactions on Information Theory*, 44(3):1087–1096, 1998.

[29] Nikolaos Leontiou, Dimitrios Dechouniotis, and Spyros Denazis. Adaptive Admission Control of Distributed Cloud Services. In *2010 International Conference on Network and Service Management*, pages 318–321. IEEE, 2010.

[30] Jörg Liebeherr, Dallas E Wrege, and Domenico Ferrari. Exact Admission Control for Networks with a Bounded Delay Service. *IEEE/ACM Transactions on Networking*, 4(6):885–901, 1996.

[31] Ming Mao and Marty Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 423–430. IEEE, 2012.

[32] Axel Mönkeberg and Gerhard Weikum. Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data Contention Thrashing. In *VLDB*, volume 92, pages 432–443, 1992.

[33] Giovanni Neglia, Damiano Carra, Mingdong Feng, Vaishnav Janardhan, Pietro Michiardi, and Dimitra Tsigkari. Access-Time-Aware Cache Algorithms. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 2(4):1–29, 2017.

[34] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 505–520. Association for Computing Machinery, New York, NY, USA, October 2021.

[35] Oracle. Capacity Reservations. https://docs.oracle.com/en-us/iaas/Content/Compute/Tasks/reserve-capacity.htm, 2023.

[36] Josep Xavier Salvat, Lanfranco Zanzi, Andres Garcia-Saavedra, Vincenzo Sciancalepore, and Xavier Costa-Perez. Overbooking Network Slices through Yield-Driven End-to-End Orchestration. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 353–365, 2018.

[37] Amilcare Francesco Santamaria and Andrea Lupia. A New Call Admission Control Scheme based on Pattern Prediction for Mobile Wireless Cellular Networks. In *2015 Wireless Telecommunications Symposium (WTS)*, pages 1–6. IEEE, 2015.

[38] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich Nahum, and Adam Wierman. How to Determine a Good Multi-Programming Level for External Scheduling. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 60–60. IEEE, 2006.

[39] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[40] David Starobinski and David Tse. Probabilistic Methods for Web Caching. *Performance Evaluation*, 46(2-3):125–137, 2001.

[41] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. Distributed Resource Management across Process Boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 611–623, 2017.

[42] Khin Me Me Thein. Apache Kafka: Next Generation Distributed Messaging System. *International Journal of Scientific Engineering and Technology Research*, 3(47):9478–9483, 2014.

[43] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The Next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[44] Guillaume Urvoy-Keller, Gérard Hébuterne, and Yves Dallery. Traffic Engineering in a Multipoint-to-Point Network. *IEEE Journal on Selected Areas in Communications*, 20(4):834–849, 2002.

[45] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[46] Matteo Vincenzi, Elena Lopez-Aguilera, and Eduard Garcia-Villegas. Timely Admission Control for Network Slicing in 5G with Machine Learning. *IEEE Access*, 9:127595–127610, 2021.

[47] Boqian Wang, Zhonghai Lu, and Shenggang Chen. ANN-Based Admission Control for On-Chip Networks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[48] Donghui Wang, Peng Cai, Weining Qian, and Aoying Zhou. Discriminative Admission Control for Shared-Everything Database under Mixed OLTP Workloads. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 780–791. IEEE, 2021.

[49] Matt Welsh and David Culler. Adaptive Overload Control for Busy Internet Servers. In *4th USENIX Sym-*

posium on Internet Technologies and Systems (USITS 03), 2003.

[50] Chen-Feng Wu, Liang-Teh Lee, Hung-Yuan Chang, and Der-Fu Tao. A Novel Call Admission Control Policy using Mobility Prediction and Throttle Mechanism for Supporting QoS in Wireless Cellular Networks. *Journal of Control Science and Engineering*, 2011, 2011.

[51] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. SLA-based Admission Control for a Software-as-a-Service Provider in Cloud Computing Environments. *Journal of Computer and System Sciences*, 78(5):1280–1299, 2012.

[52] Haitao Yuan, Jing Bi, Wei Tan, and Bo Hu Li. CAWSAC: Cost-Aware Workload Scheduling and Admission Control for Distributed Cloud Data Centers. *IEEE Transactions on Automation Science and Engineering*, 13(2):976–985, 2015.

[53] Xiaolu Zhang, Demin Li, Wei W Li, and Wei Zhao. An Optimal Dynamic Admission Control Policy and Upper Bound Analysis in Wireless Sensor Networks. *IEEE Access*, 7:53314–53329, 2019.

[54] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161, 2018.

[55] Timothy Zhu, Daniel S Berger, and Mor Harchol-Balter. SNC-Meister: Admitting More Tenants with Tail Latency SLOs. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 374–387, 2016.

Appendix

A   Theoretical Guarantees

In this section, we show that the conversion ratio approach does not lead to an excessive wastage of resources.

Notation.   We start by defining some useful notation.
- Let y_m^t denote the type-t VMs that can fit in a machine m, ignoring buffers.
- Let x_c^t denote the number of VMs of type t required for buffers in cluster c, and let $x_c^{t' \to t} = \mathbb{C}(t' \to t, x_c^{t'}, c)$ be the converted number of VMs from type t' to type t.
- We define $C_{t' \to t} = \frac{\mathsf{A}[t,c]}{\mathsf{A}[t',c]}$, which allows us to write the conversion ratio from type t' to type t as

$$x_c^{t' \to t} = \mathbb{C}(t' \to t, x_c^{t'}, c) = \left\lceil C_{t' \to t} \cdot x_c^{t'} \right\rceil.$$

Setting.   Our analysis focuses on the following setting:
- We assume that one of the resources (e.g., CPU) is the main bottleneck in a cluster; let q_t denote the amount of that resource that VM type t requires.
- When $C_{t' \to t} < 1$ (i.e., we are converting from "smaller" to "larger" VMs), we omit the ceiling function from the converted AV count calculation. Instead, $x_c^{t' \to t} = C_{t' \to t} \cdot x_c^{t'}$.
- Let $M_c^{t,t'}$ denote the set of machines in cluster c that can currently fit both VM types t' and t. The conversion ratio from t' to t is then calculated using only the machines in $M_c^{t,t'}$. Specifically, we let $\mathsf{A}[t,c] = \sum_{m \in M_c^{t,t'}} y_m^t$ and $\mathsf{A}[t',c] = \sum_{m \in M_c^{t,t'}} y_m^{t'}$ for calculating $C_{t' \to t}$.

Results.   To quantify the waste in resources, we compare the total capacity of the VMs that are being converted with the total capacity of the AV counts that are deducted by this conversion. In particular, let $U_{original} = \sum_{t'} x_c^{t'} \cdot q_{t'}$ denote the total capacity of the VM types before conversion (i.e., "real" capacity of buffers), and $U_{converted} = q_t \cdot \sum_{t'} x_c^{t' \to t}$ denote the converted capacity of these VMs.

Theorem 1 *The conversion guarantees at least $\frac{1}{4}$−utilization of the converted AV counts' capacity; explicitly: $U_{original} \geq \frac{1}{4} \cdot U_{converted}$.*

To prove Theorem 1, we will need the following auxiliary lemma that relates the conversion ratio with the sizes of the VM types.

Lemma 1 *Under the above assumptions, $C_{t' \to t} \leq 2 \cdot \frac{q_{t'}}{q_t}$.*

Proof of Lemma 1.   Consider a machine m that can currently fit at least one VM of type t or one VM of type t'. Let Q_m denote the available capacity of the bottleneck resource (e.g., CPU) of machine m. By definition, $y_m^t = \lfloor \frac{Q_m}{q_t} \rfloor$ and $y_m^{t'} = \lfloor \frac{Q_m}{q_{t'}} \rfloor$. Then, the following are true:

$$q_t \cdot y_m^t \leq Q_m \quad \text{and} \quad q_{t'} \cdot (y_m^{t'} + 1) > Q_m.$$

Combining the above two inequalities, we obtain:

$$q_t \cdot y_m^t < q_{t'} \cdot (y_m^{t'} + 1) \Rightarrow \frac{y_m^t}{y_m^{t'}} < \frac{q_{t'}}{q_t} + \frac{q_{t'}}{q_t \cdot y_m^{t'}} \leq 2 \cdot \frac{q_{t'}}{q_t}$$

where the last inequality follows from the fact $y_m^{t'} \geq 1$.

Since this is true for all machines m that fit both VM types, we obtain:

$$\mathsf{A}[t,c] = \sum_{m \in M_c^{t,t'}} y_m^t \leq 2 \cdot \frac{q_{t'}}{q_t} \sum_{m \in M_c^{t,t'}} y_m^{t'} = 2 \cdot \frac{q_{t'}}{q_t} \cdot \mathsf{A}[t',c].$$

We can then easily see that:

$$C_{t' \to t} = \frac{\mathsf{A}[t,c]}{\mathsf{A}[t',c]} \leq 2 \cdot \frac{q_{t'}}{q_t}.$$

Proof of Theorem 1.   We need to consider two cases.
- **Case 1:** $C_{t' \to t} < 1$. In this case, we obtain the following:

$$x_c^{t' \to t} = C_{t' \to t} \cdot x_c^{t'} \leq 2 \cdot \frac{q_{t'}}{q_t} \cdot x_c^{t'}.$$

We then get for the original and converted capacity of VM type t':

$$q_{t'} \cdot x_c^{t'} \geq \frac{1}{2} \cdot q_t \cdot x_c^{t' \to t}.$$

- **Case 2:** $C_{t' \to t} \geq 1$. Similarly, in this case:

$$x_c^{t' \to t} = \left\lceil C_{t' \to t} \cdot x_c^{t'} \right\rceil \leq C_{t' \to t} \cdot x_c^{t'} + 1 \leq C_{t' \to t} \cdot (x_c^{t'} + 1)$$
$$\leq 2 \cdot C_{t' \to t} \cdot x_c^{t'} \leq 4 \cdot \frac{q_{t'}}{q_t} \cdot x_c^{t'}.$$

In the above, the second inequality follows from $C_{t' \to t} \geq 1$. The third inequality is due to $x_c^{t'} \geq 1$ (otherwise, no buffer of type t' would exist, clearly leading to zero waste in the conversion by default). Finally, the last inequality uses Lemma 1. As a result,

$$q_{t'} \cdot x_c^{t'} \geq \frac{1}{4} \cdot q_t \cdot x_c^{t' \to t}.$$

Putting both cases together, we can see that for all converted VM types t', we have:

$$q_{t'} \cdot x_c^{t'} \geq \frac{1}{4} \cdot q_t \cdot x_c^{t' \to t}.$$

Summing over all such VM types:

$$\sum_{t'} q_{t'} \cdot x_c^{t'} \geq \frac{1}{4} \cdot \sum_{t'} q_t \cdot x_c^{t' \to t} \Rightarrow U_{original} \geq \frac{1}{4} \cdot U_{converted}.$$

which concludes the proof of Theorem 1.