Protean: VM Allocation Service at Scale

Ori Hadary Luke Marshall Ishai Menache Abhisek Pan
Esaias E Greeff David Dion Star Dorminey Shailesh Joshi
Yang Chen Mark Russinovich Thomas Moscibroda *

Microsoft Azure and Microsoft Research

Abstract

We describe the design and implementation of Protean – the Microsoft Azure service responsible for allocating Virtual Machines (VMs) to millions of servers around the globe. A single instance of Protean serves an entire availability zone (10-100k machines), facilitating seamless failover and scale-out to customers. The design has proven robust, enabling a substantial expansion of VM offerings and features with minimal changes to the core infrastructure. In particular, Protean preserves a clear separation between *policy* and *mechanisms*. From a policy perspective, a flexible rule-based Allocation Agent (AA) allows Protean to efficiently address multiple constraints and performance criteria, and adapt to different conditions. On the system side, a multi-layer caching mechanism expedites the allocation process, achieving turnaround times of few milliseconds. A slight compromise on allocation quality enables multiple AAs to run concurrently on the same inventory, resulting in increased throughput with negligible conflict rate. Our results from both simulations and production demonstrate that Protean achieves high throughput and utilization (85-90% on a key utilization metric), while satisfying user-specific requirements. We also demonstrate how Protean is adapted to handle capacity crunch conditions, by zooming in on spikes caused by COVID-19.

1 Introduction

The Cloud has revolutionized the way computing resources are consumed. Providers allow end-users easy access to secure, elastic and state-of-the-art resources, while applying efficient management techniques in order to optimize their return on investment. In particular, resource *virtualization* is used to maximize the utilization of the underlying hardware. Consequently, one of the most crucial components in the cloud stack is the Virtual Machine (VM) *allocator*, which assigns VM requests to the physical hardware. Indeed, sub-optimal placement decisions can result in fragmentation (and in turn, unnecessary over-provisioning of physical resources), performance impact and service delays, and even rejection of incoming requests and customer impacting allocation failures.

In this paper, we describe in detail the VM allocator for Azure – one of the leading cloud service providers in the world. Azure provides and manages infrastructure for SAAS, PAAS and IAAS workloads. Its fleet consists of millions of physical machines spanning more than a hundred countries. Azure offers more than 500 different VM types tailored to a vast array of application requirements reflected in the virtual resource specification of each VM. VMs serve as the primary units of (multi-dimensional) resource allocation, and the means through which customers are able to leverage the rich array of computing services offered by Azure; see §2 for an analysis of Azure workloads.

The rapid growth of Azure both in terms of its feature set and massive geo-scale mandated that its core VM allocator be designed in a *robust* manner. First, the allocator logic must be *extensible* – to efficiently facilitate new features, constraints and offerings over time. Second, close attention was given to *flexibility* – in our context, the ability to configure the allocator to different working conditions and scenarios. Third, the core algorithms had to be highly *optimized*: Given Azure's scale, even 1% in fragmentation reduction can lead to cost savings in the order of $100M per year.

From an operational perspective, the total demand in Azure is in the order of millions of VMs per day. Such large scale leads to a complicated system challenge – satisfying this high request rate while maintaining fast response times and high resource utilization. In principle, an allocation service needs to control a sufficiently large inventory of underlying capacity (or domain), so that new requests assigned to the domain can be accommodated, and customers within the domain can scale-out (namely, get additional VMs upon request). However, controlling a large inventory inherently impacts the *latency* of an allocation. To avoid unacceptable delays, a design must include efficient mechanisms for determining the physical placement of the VM. In addition, to achieve adequate *throughput*, the system architecture may incorporate multiple allocation processes [43]. Parallelizing the allocation logic introduces new challenges, such as sustaining high resource utilization while keeping conflicts to a minimum.

While some of these challenges have been discussed in similar contexts [17, 38, 43, 48], most previous works either do not provide full details on the design and implementation, or resort to simulation studies (or small size implementations) without providing comprehensive evaluation from a global-scale production deployment. In this paper, we describe the

*O.H, L.M, I.M and A.P contributed equally to this paper.

design, implementation and evaluation of Protean, the core allocation service governing all VM placement and resource allocation in Azure. An instance of Protean operates at the granularity of an availability zone (typically 10-100 thousands of machines), which allows for high acceptance rates and seamless scale-out capabilities.

To achieve the desired robustness while controlling such large inventories, Protean's design provides a clear separation between *policy* and system *mechanisms*. Policy is expressed through a flexible rule-based Allocation Agent (AA), which addresses multiple constraints and performance criteria for allocating VMs. AA's logic is inherently *extensible*; rules can be refined and added with minimal disruption to the system. Crucially, the rule-based semantics foster *explainability*, and force clear and conscious trade-offs between the numerous metrics and optimization criteria. On the system side, a multi-layer caching infrastructure keeps track of previous allocation outcomes through efficient update mechanisms, resulting in an order of magnitude reduction in turnaround time compared to a system without this caching layer. Notably, the memory footprint of the cache is manageable (e.g., around 1GB for 10k machines), and scales sublinearly with the number of machines. By slightly compromising on the allocation quality, we enable multiple AAs to run concurrently, resulting in increased throughput with negligible conflict rate. The number of AAs, as well as key rule parameters, are tuned at a slow time-scale using production data.

Our results from real production measurements and a variety of simulations demonstrate that Protean achieves low latency (typically 20ms per VM), while satisfying user-specific requirements and values of 85-90% for a key utilization metric. Importantly, Protean can easily satisfy the peak demands observed in production (up to 2000 requests per second), and may sustain much higher throughput if needed, as demonstrated in simulations §6.2. In addition, we show how Protean adapts to different conditions, by focusing on recent capacity challenges during the COVID-19 crisis. In particular, we discuss how Protean seamlessly allowed critical control-plane policy changes that were required to support the sudden increase in demand. In summary, our main contributions are:

- We provide a detailed analysis of the workload and inventory of Azure. Our analysis (§2) reveals key characteristics, which motivate Protean's design.
- We design a flexible rule-based allocation agent (§3), which allows operators to incorporate new logic and *explain* allocation outcomes to customers.
- To our knowledge, we provide the first detailed account of the allocation logic and key implementation details of a core VM allocator in a leading public cloud provider. Our implementation includes a novel caching infrastructure tailored to expedite the VM allocation process (§5).
- We evaluate Protean using extensive measurements from geo-scale production, and augment these evaluations with low and high fidelity simulators where necessary (§6). Our results show that Protean achieves low latency and high throughput while sustaining high utilization under diverse operating conditions.

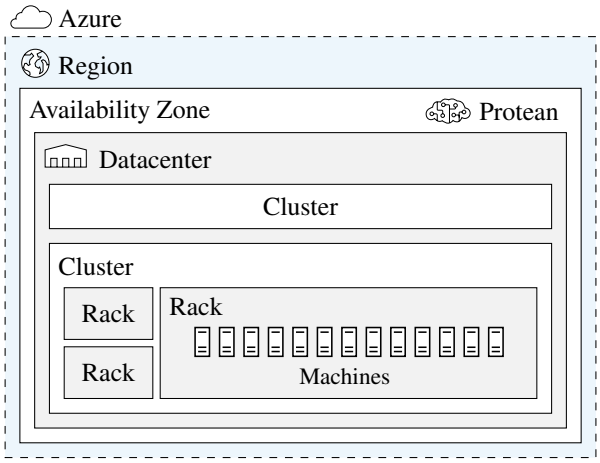

Figure 1: Azure Cloud Topology: Regions consist of availability zones, each of which comprise several datacenters that house racks of servers.

2 Background and Motivation

2.1 Azure – A Global-Scale Public Cloud

The inventory. The global Azure inventory is arranged in a hierarchy of *regions* and availability *zones*, exposed directly to the customer. A region can have up to three zones, each in turn consisting of one or more *datacenters* (see Fig. 1). Each datacenter is divided into *clusters* and *racks*. There is no strict upper bound on the size of a zone or a datacenter. Our larger zones have over a hundred thousand machines, spread over more than a hundred clusters, with each cluster having around a thousand machines. The smaller zones have only around a thousand machines.

The inventory within a zone is typically heterogeneous, with machines ranging across multiple hardware generations and Stock Keeping Unit (SKU) configurations, including special servers for HPC, GPUs, etc. Table 1 summarizes the distribution of the different hardware generations for one of the zones. In this case, the bulk of the inventory belongs to two generations, while the others represent a generation that is being decommissioned, and a new generation that is in early-stage deployment. In contrast to zone heterogeneity, a cluster is a *homogeneous* set of machines (e.g., identical SKUs and configurations) spanning multiple racks; each cluster supports most VM types. In §3.2 we discuss how we exploit cluster homogeneity to improve request latency.

The workload. As mentioned, Azure exposes numerous options for renting VMs. Users specify their requirements in

Gen	Cores Per Machine	# Machines
3	10	7295
4	24	34208
5	40	18016
6	48	2064

Table 1: Distribution of hardware generations for a zone. Individual similar SKUs are aggregated within each generation.

the form of an *service request*. Each zone may accommodate millions of requests per day. A service request consists of one or more *VM requests*, grouped as a *tenant*. The tenant service model expresses the relationships and constraints imposed on these VMs. An allocation succeeds only if all the requested VMs within a tenant are successfully allocated (gang-scheduling). A service model specifies the type of each VM (which in turn determines the core, memory, disk, or network requirements for the VM), fault domain requirements, and priority of the service. By default, the tenant VMs can be spread across the entire zone. However, a tenant can request all its VMs to be co-located within specific inventory boundaries such as a datacenter or a row, or conversely, can forbid too many of its VMs from being placed on the same machine or rack. A tenant can even specify that no VMs from any other tenant be placed on the machine that hosts its VMs.

A customer can resize (scale in/out) or delete an existing tenant, or create a new tenant. Platform initiated requests due to unexpected machine failures, planned maintenance, or decommissioning of machines can lead to reallocation of some or all tenant VMs. Note that higher-level services can stitch together multiple tenants to expose alternative grouping semantics to customers, such as jobs with tasks that can be incrementally scheduled, or auto-scaled group of identical VMs (e.g., virtual machine scale sets [4]). These services are responsible for breaking the groups of VMs into tenants before sending service requests to Protean.

Protean – a zone allocation service. Azure operates an allocation service for each zone, termed Protean.[1] Requests are assigned to each zone either directly by the customer, or by a higher-level service. The main role of Protean is to find a physical placement (machine) for each VM in an allocation request, subject to explicit requirements and constraints specified in the underlying service model, as well as other internal operational considerations. To cope with large request loads, Protean employs multiple *Allocation Agents (AAs)*, which run in parallel. Similar to [43], each agent is aware of the entire inventory and can choose any eligible machine from the inventory to host a VM. The authoritative state of the inventory is maintained in a persistent store. Each AA maintains its own view of the inventory, which is updated periodically and in response to allocation or inventory related events.

[1] Protean means able to change frequently, versatile.

2.2 Workload Analysis

We next analyze some properties of our workload, with a focus on characteristics that have influenced Protean's design.

Demand is heterogeneous. Our zones exhibit workloads which are fairly diverse in nature. We observe a large number of different VM types, see Table 2. The distribution is generally nonuniform – some VM types may account for up to 50% of the workload, while others are rare. To give more insight into the challenge pertaining to packing the VMs, Table 3 shows the distribution of CPU requirements, measured in number of cores. We observe that most VMs require a small number of cores, but some require half or even an entire server.

VM Type	Zone1 (%)	Zone2 (%)
A	4.6	0.1
B	3.5	3.6
C	6.5	12.9
D	0.7	8.4
E	1.9	3.7
F	3.2	4.4
G	0.6	3.1
H	0.8	2.2
I	2.4	7.4
J	23.7	31.6
K	21.3	2.1
L	3.5	0.4
M	0.0	2.2

Table 2: Distribution of VM types for selected zones. VM types having $< 2\%$ in both zones are excluded. We avoid using real VM type names to preserve confidentiality.

Subsequent requests are similar. While our system supports many VM types, we observe that subsequent requests are fairly "similar". For example, Fig. 2 shows the *reuse distance*, which for each request of VM type v, measures the number of unique VM types requested since the last time that v was requested. We observe that more than 80% of requests have zero reuse distance, while the majority has distance less than five. This behavior can be attributed to a combination of factors, such as large service requests that ask for the same type of VM, and having a relatively small set of popular VM

VM Cores	Zone1 (%)	Zone2 (%)
1	17.1	27.0
2	37.4	52.4
4	32.0	10.5
8	8.9	4.5
$\geqslant 10$	4.3	2.6
$\geqslant 20$	0.3	3.1

Table 3: Distribution of VM resources for selected zones.

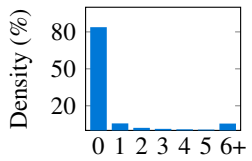

Figure 2: Reuse distance for VM requests in a zone for an entire day.

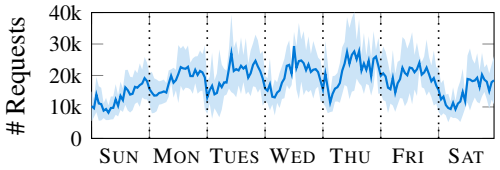

Figure 3: Number of VM requests for a zone. Counted over each hour, and averaged over day-of-week for five weeks (shaded area is standard deviation).

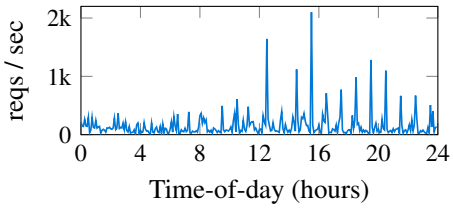

Figure 4: Requests per second for a zone on a single day, with max over 5 min intervals.

types (see Table 2). This "locality" property plays a key role in our design for the allocation agent.

VM lifetime varies substantially. We observe that most VM lifetimes are short, in the order of several minutes. However, some VMs can stay in the system "forever" – for weeks and months. See Figure 5 for empirical lifetime distributions across representative zones.

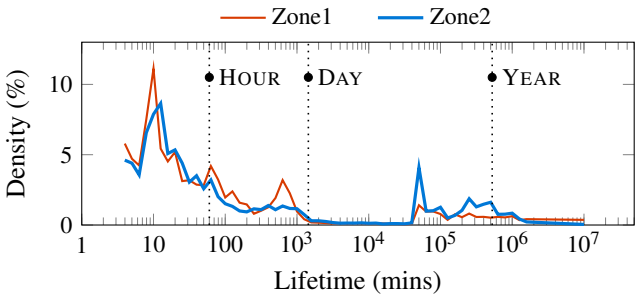

Figure 5: Zone-based VM lifetime distributions. The data represents VMs that were alive during a ten-day period in March 2020. The lifetime of VMs that remained alive when the data was collected (June 2020) may be longer than reported.

Demand has spikes and diurnal pattern. Fig. 3 shows the request count over a week in one of our busier zones, averaged over day-of-week for 5 weeks. We observe some diurnal patterns, e.g., typically less usage overnight. Demand can also have large spikes throughout the day, as seen in Fig. 4. Note that the demand reaches above 2k requests per second. This behavior forces us to provision for the peak by employing multiple AAs for each Protean instance (see §4). At the same time, we exploit the off-peak periods to better prepare the AAs for future allocations (see §5.2).

Tenants sizes are typically small, but can be huge. Our analysis on two large zones indicates that 94% of requests are for a single VM. 99% of requests are for five or less VMs. We also observe a few requests for hundreds of VMs; naturally, such requests would pose additional challenges (e.g., spreading the VMs across different fault domains).

2.3 Takeaways

Scale and uncertainty. Our analysis demonstrates that the incoming demand is highly variable. Because Protean latency and throughput requirements cannot be compromised, our design has to account for extreme demand conditions. Furthermore, Protean has to accommodate small and large regions, which requires flexible configurations (for example, the number of AAs).

Opportunity for caching. We have provided evidence that subsequent requests are similar over time. This motivates the "caching" of placement evaluation logic, and reuse across multiple requests – this idea is central in our design and facilitates scaling to large zones and regions.

The packing challenge. Our workload is highly diverse – numerous VM types of different sizes, high variability in lifetimes (which are unknown in advance). This poses a substantial challenge in adequately "packing" the VM in physical servers. Algorithmically, a simplified version of our packing problem already maps to dynamic bin packing [11], which is an NP-hard problem in the offline setting (i.e., assuming all VM arrivals are known), with quite bad competitive ratio in the online setting [6]. In our practical setting, we have other elements that make the problem even more challenging (multiple priorities, fault domain requirements, etc.). Supplementary to this paper, we release a new trace that can be used by the research community to design and test different packing algorithms [3].

3 Rule-Based Allocation Agent

In this section, we describe the main design principles of Protean's allocation agent.

3.1 Metrics and Constraints

Metrics. Protean targets several metrics related to both performance and quality of the allocation. The key metrics are:

- *Latency.* A single VM allocation should be satisfied promptly, typically within 20 ms.
- *Throughput.* Protean should be able to handle peak demands without delaying or throttling requests.

- *Acceptance.* Protean should minimize the rejection rate. A rejection occurs when a VM request cannot be satisfied.

We note that latency is not important in isolation (although might become excessive for large tenants). Nonetheless, lower latency facilitates higher throughput. Intuitively, when latency is lower, requests can be processed by fewer AAs, resulting in fewer conflicts and likely higher throughput. Furthermore, lower latency decreases the probability that an individual AA drifts from the true state of the inventory, which improves allocation quality. Naturally, the three metrics above depend on the size of the inventory. Latency and throughput become more challenging with larger inventories, but accepting requests becomes easier. We are also interested in efficient usage of compute resources; we will formally define utilization-related metrics in §6.

Requirements and constraints.
Addressing multiple considerations. First and foremost, Protean must make *correct* assignments; for example, an allocation cannot violate the capacity of a machine. Additionally, the request may include certain constraints that the allocation must satisfy. For example, certain VMs may require a specific type of hardware (e.g., GPUs). Furthermore, a service request for multiple VMs may require that the VMs are spread across multiple fault domains (typically across different racks).
Tenant experience. Protean should avoid allocations on machines that are not in a "ready" state; have not been updated with the latest host environment; or are likely to fail in the near future. If a machine fails, then its hosted VMs *must* be allocated to other machines. Low priority VMs are used by Azure offerings, such as Batch [1] and Spot Virtual Machines [2]. While these VMs are allowed to be preempted, Protean still aims to minimize their eviction rates.
Adaptability. Protean must allow for an easy configuration of allocation logic, and adjust for different conditions.
Extensibility and interpretability. Protean should be easily and safely extendable, in order to enable engineers to incorporate new allocation logic. Accordingly, the allocation logic should allow for incremental changes, and performing A/B testing in production. Moreover, Protean should enable operators to interpret the allocation decisions (e.g., "why did the request fail?", "why was machine x chosen for VM request v?"); explaining allocation outcomes is regarded as one of the main challenges in large-scale cloud scheduling [46].

3.2 Allocation Rules

As discussed above, Protean has to account for multiple considerations simultaneously. First, strict placement constraints need to be enforced (e.g., a VM has to be allocated to a specific hardware type); other placement considerations can be viewed as "preferred", for example, it is better to place the VM on a server that is perceived as healthy, has certain disk configuration, etc. On top of that, Protean targets "high-quality" allocations; for example, packing the servers efficiently by minimizing fragmentation, balancing allocations across racks, avoiding lower-priority VMs evictions, etc. Due to the numerous dimensions involved, Protean's allocation logic is organized as a set of *rules*. The rules determine which machine will be assigned for each individual VM. A service request for k VMs will invoke the rule logic k times.

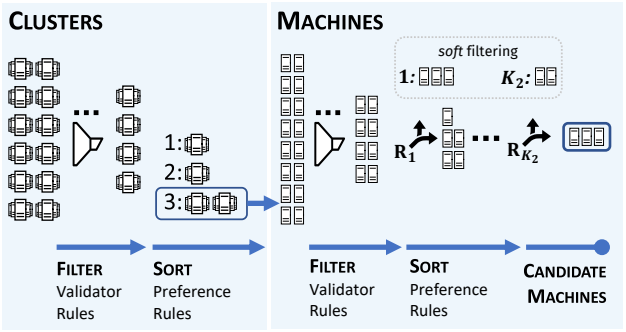

Figure 6: Rule based selection.

Rules are classified into either *validator* or *preference* rules. A validator rule accounts for hard constraints, whereas a preference rule can be viewed as a soft constraint. The rules are arranged in a two-level hierarchy – cluster selection rules followed by machine selection rules (Fig. 6). In total, there are currently around one hundred rules.

Cluster and machine selection rules. Cluster selection rules effectively reduce the time complexity of the selection process by limiting the scope of the machine selection rules to a small number of clusters in a zone. Because clusters are homogeneous (see §2), we implement several cluster validator rules which filter out clusters that are not relevant for the VM request (e.g., a VM that requires a GPU machine can by hosted only on a cluster with GPU machines). In addition, a few cluster preference rules are used to sort the valid clusters (e.g., we prefer emptier clusters to balance the available capacity across clusters). Based on that, Protean chooses the k highest-quality clusters, where $k \in [8, 16]$ is a configurable parameter; the inventory for the machine selection rules will in turn consist of the machines in these clusters. The parameter k is set based on a tradeoff between exposing a large set of machines for making high-quality decisions and sustaining adequate latency.

In turn, machine selection rules again consist of validator rules that exclude specific machines from being considered, followed by preference rules which eventually select a small number of machines that are the best match for the particular VM. A randomized tie-breaking rule picks one of these machines for the physical assignment of the VM. In what follows we provide a more formal description of rule semantics, as well as some examples.

Validator rules. Each validator rule implements the Boolean method IsValid(x, v) to indicate whether an object x is a valid candidate for placing VM v; an object can be either a cluster or

a machine, depending on the rule. Validator rules are used to prune the set of objects in the inventory to a subset of objects that are valid candidates for placing the VM. **Examples**: (1) AreNodeResourcesValid(x, v) checks whether machine x has enough available capacity to accommodate VM v. The method returns *true* if all resource dimensions can be fulfilled (CPU, memory, disk, etc.). (2) IsTypeSupported(x, v) checks whether cluster x is compatible with the VM type corresponding to v.

Preference rules. A preference rule quantifies the extent to which each candidate object x (either a cluster or a machine, depending on the rule) is a good fit for the VM. Each preference rule r accounts for a specific consideration (e.g., packing quality, balancing, cluster/machine quality, etc.) through a numeric *score* $S_r(x,v)$. We use the convention that a lower score is better. **Examples:** (1) BestFit(x,v) assigns a score $\sum_i w_i(a_i(x) - d_i(v))$, where $a_i(x)$ is the availability of "resource" i (e.g., CPU cores, memory, SSD)[2] in machine x, $d_i(v)$ is the requirement of the VM for that resource, and w_i is the weight of the resource which quantifies its scarcity (intuitively, the higher w_i the scarcer the resource). A lower score here implies that the machine is a better fit for that VM, since it is closer to being fully packed; we note that similar packing heuristics have been proposed in [39]. (2) PreferNonEmptyMachines(x,v) This rule prefers to use machines that are non empty, primarily in order to improve packing quality. (3) PreferEmptierClusters(x,v) This rule quantifies how many empty cores cluster x has. The idea here is to balance the available capacity among clusters. This is done to minimize the probability that the cluster capacity is exhausted, which is important from several perspectives. For example, some customers require affinity within the cluster, and would not be able to scale out if the cluster is completely full.

3.3 Accounting for Multiple Rules

As illustrated in Fig. 6, the sequence of validator rules filters out objects (clusters, machines) that are not eligible for the particular VM. One of the main challenges in the design of Protean was: how to account for multiple preference rules? The inherent issue here is that different rules represent different and hard-to-compare preferences. We describe below the principles of our approach.

Compare method. Each preference rule r implements the Compare$(x, y \mid v)$ method to compare two objects x and y based on their scores; the method returns 0 if scores are equal, 1 if $S_r(x,v) < S_r(y,v)$, or -1 otherwise.

Comparisons and sorting. Each preference rule expresses its relative importance using a *weight* (or gain value). Two objects are compared according to an aggregate score computed as the sum of products of the compare value returned by each rule compare method and its weight. Using the pairwise comparisons, Protean computes a sorted list of the entire set of objects based on their aggregate preference scores.

Weight assignment. While our system allows to set any positive value for the rule weights, we have chosen to set the weights in a way that imposes strict ordering between the preference rules. The rules are assigned weights according to an order-preserving encoding (i.e., weights are exponentially apart from each other), such that, effectively, any rule can only express a preference among objects that have been preferred by the previous rule. Accordingly, the entire set of rules (including validator rules) can be regarded as a *filtering* process in which the set of preferred objects is narrowed as more rules are considered; see §3.4 for discussion.

Quantization. Having a strict prioritization among the preference rules, requires "smoothing" the preference rules, so that all rules can contribute. We do so by quantizing the score of some rules into a small number of buckets (e.g., rules with a continuous score, such as BestFit). For example, we may apply the transformation $\lceil S \cdot N \rceil$, where $S \in [0,1]$ is the original (continuous) score and N is the number of buckets. The rule ordering and the specific quantization values entail domain knowledge and understanding of business needs and preferences. Their setting is based on trial and error, building on simulation results as well as production telemetry.

3.4 Discussion

We conclude this section by discussing how the rule-based allocator helps us achieve our design goals.

Addressing multiple considerations. Having multiple rules allows us to address multiple hard constraints, and explicitly influence the quality of the allocation through designated rules (e.g., best-fit for packing). The fundamental requirement of making "correct" allocations will manifest itself in certain system mechanisms; for example, ensuring that decisions are made based on the true state of the zone (§5).

User experience. By design, Protean will *not* fail a VM request if there exists a feasible assignment for that VM. This holds because (i) Protean chooses clusters that contain feasible nodes; (ii) Protean considers all nodes in the selected clusters. In addition, Protean has rules that target better user experience (e.g., prefer "healthier" machines).

Adaptability. The rules themselves can be customized and refined as needed. For example, if a specific rule is too "aggressive", a simple configuration change can make it softer, e.g., by making the quantization coarser. As a concrete example, we describe in §6 how Protean has been adapted to tackle a capacity crunch during the Covid-19 crisis.

Extensibility and robustness. Our rule-based allocator is inherently extensible and robust. It is not too difficult to insert a new rule, or to modify or delete an existing one; two main design choices enable that: (i) rather than using a general

[2] Protean currently does not account for power. Power budgets are defined for different aggregations of servers: chassis, racks, rows, etc. A separate power-capping system [31] ensures that power usage does not exceed the budget; since power consumption falls within the budget at high percentiles, capping engages rarely.

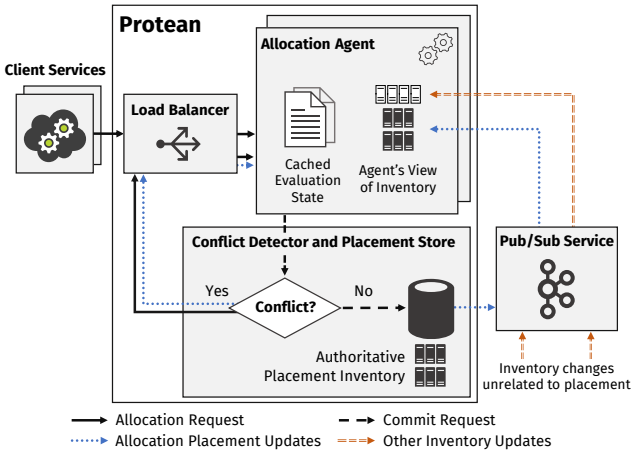

Figure 7: Protean System Architecture

weighted score function, Protean maintains a ranking of the rules, and auto-generates weights that maintain the exponential distance property; (ii) the internal rule scores are not factored in the sorting of the machines, which makes the design or modification of rules more robust. This is enabled by using the rule compare method as a building block.

Interpretability. Strict ordering allows for better interpretability. For example, we can infer why a certain machine was not chosen for a particular VM request. More broadly, we can aggregate statistics to determine how aggressive each rule is (e.g., what fraction of objects it "filters on average"). This evaluation helps us adjust the quantization of rules and their ordering if needed; see §6 for an example.

4 Architecture

In this section, we provide a high-level overview of Protean, and describe how the AA handles a service request.

Protean operation. Fig. 7 describes the high-level system architecture of Protean. Protean employs multiple Allocation Agents (AAs) that operate concurrently, following an optimistic concurrency model. The AAs are organized to run in multiple machines. Each machine hosts a single process, which in turn creates multiple worker threads, one thread per AA. Allocation requests from clients are routed to these processes through a load-balancer. Within a process, the requests are stored in a shared work-queue until they are picked up and processed by a free AA. The number of AAs is determined according to the peak instantaneous demand in the zone, while the number of AAs per machine depends on the memory footprint of each AA. Each AA makes allocation decisions based on its own (possibly stale) view of the inventory, and after processing a request successfully, tries to commit the result to a replicated store. The replicated store performs conflict detection, and serializes the commits to the same node (commits to different nodes are handled in parallel). Further, it stores all inventory information that is modified by the VM placement decisions from AAs. The replicated store functions as the authoritative source for the latest placement-related inventory state, and publishes all changes through a publish-subscribe (pub/sub) service.

Algorithm 1: Service allocation algorithm

```
1  def ALLOCATE_SERVICE(v_1,...,v_n, retries):
2      v_1,...,v_n ← ORDER(v_1,...,v_n)
3      for i = 1 to n do
4          m_i = ASSIGN_MACHINE_TO_VM(v_i)
5          if IS_INVALID_MACHINE(m_i) then
6              return FAILED
7      if COMMIT(m_1,...,m_n) then
8          return SUCCEEDED
9      else if retries < MAX_RETRIES then
10         return QUEUE FOR RETRY
11     else
12         return FAILED
```

Changes in the inventory that are not influenced by placement decisions, such as changes in machine health or capabilities, are also published via the pub/sub service. The AAs learn about inventory changes primarily through the updates produced by the pub/sub service. Additionally, on commit failures due to conflicts, they learn about the latest placement-related information for the conflicting machines as part of the failure notification.

Service allocation workflow. A service request may consist of multiple VM requests that are processed sequentially by a single AA. Algorithm 1 summarizes how Protean handles a service request. ORDER determines the order in which the VM requests will be processed. The goal of the ordering is to minimize the risk that a request is rejected due to fault domain considerations. ASSIGN_MACHINE_TO_VM attempts to assign a machine to a single VM by applying the rule logic; it is applied sequentially for each of the requested VMs (see §5 for implementation details). If the AA succeeds in assigning machines for all of the requested VMs, COMMIT tries to commit the service allocation result to the authoritative store. The commit fails if any of the VM-Machine assignments is invalidated because of a conflicting assignment made by another AA. On commit failures, the allocator state is rolled back and the entire request is re-queued for retry. The number of retries is configurable. We allow for a relatively high number of retries (more than 10) to avoid unnecessary allocation failures; however this has a negligible effect in production (e.g., the 99.9-percentile allocations succeed after three retries). The commit stage is pipelined with the previous stages, so that the AA is free to process the next request while a commit is in flight.

5 Protean Implementation

In this section, we describe our caching framework, which substantially expedites the ASSIGN_MACHINE_TO_VM pro-

cedure (§5.1–5.2). We also discuss our flexible conflict detection and reduction mechanisms (§5.3).

5.1 Preliminaries

To make high quality assignments, the AA initially considers the entire set of machines in the inventory as candidates.

Cluster selection. As discussed in §3.2, the AA starts the selection process by filtering and sorting clusters instead of machines. Since a zone has at most a few hundred clusters today, filtering and sorting the clusters is very fast (a couple of milliseconds at most). Accordingly, the cluster selection phase does not require any additional enhancements (such as caching past selection decisions). The output of this phase is the best 8 to 16 clusters; their machines (typically 10-15k) are the candidates for the machine selection process.

Machine selection – basic complexity. Recall that the machine-selection logic first trims the set of candidate machines to the set of valid machines by evaluating all validator rules for each machine. Then, it builds a comparison-based total ordering of the machines in the valid set, based on the suitability of each machine in hosting the VM (§3.3). Finally, a machine is randomly selected from the set of best machines. Building an *evaluation result* – the ordered list of valid machines – incurs a runtime complexity of $N\sum_{i=1}^{K_1} T_v(i) + N\log N\left(\sum_{i=1}^{K_2} T_p(i)\right)$, where N is the number of candidate machines, K_1 the number of validator rules, K_2 the number of preference rules, $T_v(i)$ the time to compute IsValid for the i^{th} validator rule, and $T_p(i)$ the time to compute Compare for the i^{th} preference rule. If the AA attempted to build this evaluation result from scratch for every request, it would exceed the required latency bounds for anything more than just a couple of thousand machines.

Motivation for caching. First, we observe *"Locality in requests"*. Each VM request is characterized by a vector of *trait* values. Example traits include: VM-Type, priority, and RequireIsolation (i.e., the VM should be on a machine of its own). There are tens of traits, each of which can take several values (including a "don't care" or empty option). In Sec. §2.2 we show that requests exhibit "locality" when zooming in on a single dimension (VM type). We note that this phenomenon carries over to the entire vector: there are a few value vectors commonly used across multiple requests, especially if they are chronologically close. The second observation is that the *inventory state changes slowly*. The state of a machine can change because of allocation-related events (addition, suspension, or deletion of VMs), or because of changes in health or other conditions of a VM or a machine. However, allocation-related events are the dominant reason for such changes. Hence the machines that change between consecutive executions of the AA are primarily the machines whose states were altered as a result of allocation decisions made by other AAs running in parallel. Since the number of parallel AAs is relatively small, there are typically not many such changes. These characteristics would allow us to drastically reduce the amount of computation performed in an execution of the machine selection logic by caching and reusing an evaluation "state" from previous executions. Intuitively, the only computation that is required is to update the state to incorporate the impact of inventory changes since the previous run. We next discuss the details of our caching approach.

5.2 Caching for Efficient Machine Selection

Each AA maintains a collection of cached objects, which together hold the information required for machine selection.

5.2.1 Caching Rule State

Caching internal rule state for efficient execution. Rules are the basic building blocks of the selection process. So, first and foremost, we use caching to improve the execution time for IsValid and Compare methods of the rules ($T_v(i)$ and $T_p(i)$ respectively). Specifically, every *rule type* implements these methods. The instantiations of each rule type, termed *rule objects*, are cached for reuse. A newly created rule object computes and stores all the information that it requires to execute the IsValid or Compare method in constant time. Usually this information is stored on a per machine basis. For example, the PreferNonEmptyMachines rule (see §3.2) stores a <*MachineID, Boolean*> dictionary that tracks whether each machine is empty or not.

Just-in-time updates of rule state. Every time a cached rule object is used, its internal state has to be brought up-to-date before its IsValid or Compare method can be called. To that end, along with the IsValid or Compare method, each rule implements the $Update(x_1,\ldots,x_m)$ method in order to update its stored state. The *Update* method is called immediately before the rule object is used. Its argument, $(x_1,\ldots,x_m)$, represents the latest state for machines that have *changed* from the last time the object was updated. Every rule can execute its IsValid or Compare function in constant time once it has updated its state with the latest changes.

Splitting rule state into multiple objects. The stored state of a rule object may depend on one or more request traits. For example, the AreNodeResourcesValid rule depends on the requested VM-Type, and hence must cache the Boolean whether the machine has enough capacity for each <*Machine, VM-Type*> pair. We observe, however, that to process a particular VM request, the rule object only needs the information for the VM-Type value of that request. Updating the state for every other VM-Type value would increase the just-in-time update time, and in turn the request processing time. Hence, instead of creating a single rule object for all requests, a rule object is created on demand for every VM-Type value. A rule object for a particular VM-Type value is used for all requests asking for that value. In effect, requests are divided into *equivalence classes* based on the relevant trait value, and

Time	Request	Cached Objects	Hits	Misses (Create New)
T1	$req(x_1,y_1)$	None	None	$Eval(x_1,y_1)$ $R_1(x_1,y_1)$ $R_2(x_1)$
T2	$req(x_1,y_2)$	$Eval(x_1,y_1)$ $R_1(x_1,y_1)$ $R_2(x_1)$	$R_2(x_1)$	$Eval(x_1,y_2)$ $R_1(x_1,y_2)$
T3	$req(x_1,y_1)$	$Eval(x_1,y_1)$ $Eval(x_1,y_2)$ $R_1(x_1,y_1)$ $R_2(x_1)$ $R_1(x_1,y_2)$	$Eval(x_1,y_1)^*$	None

Table 4: Example cache timeline. *If $Eval(x_1,y_1)$ needs to be updated then $R_1(x_1,y_1)$ and $R_2(x_1)$ would also be requested (and have cache hits).

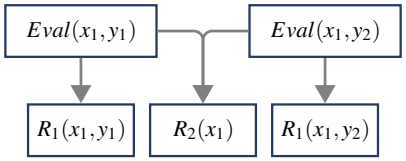

Figure 8: Cache hierarchy for the example in Table 4.

a single rule object handles all requests belonging to a class. For rules depending on multiple request traits, each unique combination of the trait values define a new equivalence class of requests for that rule. For the special case where a rule does not depend on any request trait (e.g., PreferNonEmptyMachine), a single rule object is used for all requests. This rule object specialization technique substantially reduces processing latency, and further decreases the effective memory size needed for caching.

Caching rule objects. Rule object references are stored in a constant size pool. The size is determined through trial-and-error based on memory footprint and hit-rate considerations. A rule object is identified by its type and the request trait value combination that it is associated with. Rule objects are evicted from the pool either if it is full (following a standard LRU eviction policy), or if they reach a certain age. Age-based eviction allows us to reduce the memory footprint during periods of low load.

5.2.2 Caching Rule Evaluation State

Caching the rule objects helps in substantially reducing $T_v(i)$ and $T_p(i)$. However, without any additional enhancements, we would still pay the sorting complexity of $N \log N$. Hence, we introduce additional objects termed *RuleEvaluation* objects. A rule evaluation object essentially holds the complete state of the evaluation, for a specific vector of trait values (see §5.1). The state includes the evaluation result (sorted list of machines) and references to relevant rule objects whose trait values match the respective values in the entire vector of trait values. A RuleEvaluation object is created after computing the evaluation result for a new vector of trait values, which serves as the identifier for the object). The object will then be reused for all requests that map to this identifier.

Updating the RuleEvaluation object. Similar to the rule objects, a cached RuleEvaluation object is brought up-to-date before it is used. However, unlike rule objects, RuleEvaluation objects use a common *Update* method, whose goal is to update the evaluation result with the changed machines. The method proceeds as follows: (1) the cached rule objects are brought up-to-date by calling their *Update* methods; (2) the modified machines are removed from the evaluation result; (3) the validator rules are run for each of these machines to determine which machines are valid; (4) valid machines are inserted back into the ordered list with an updated position. Because each insertion takes $\log N$ time, the *Update* method has runtime complexity of $M \log N$, where M is the number of machines with modified state. This is a substantial reduction in complexity, because $M \ll N$ (M is in the order of tens). RuleEvaluation objects are cached in another constant size memory pool, with an LRU eviction policy.

Example. Consider an example scenario where each request can have two traits $X \in \{x_1,x_2\}$ and $Y \in \{y_1,y_2\}$; and the allocation logic is expressed through two rules: R_1, which depends on traits X and Y, and R_2, which depends on X. Figure 8 shows the various rule and RuleEvaluation objects that are created and reused as the allocation engine serves incoming requests with different trait values. The accompanying Table 4 shows the hierarchy of objects that are created and cached as a consequence of processing the requests. The first request at time T1 has trait values $X = x_1$ and $Y = y_1$, and accordingly two new rule objects $R_1(x_1,y_1)$ and $R_2(x_1)$, and an evaluation object $Eval(x_1,y_1)$ are created. The second request at time T2 uses a different value y_2 for trait Y, and hence cannot reuse $Eval(x_1,y_1)$ or $R_1(x_1,y_1)$ objects. It reuses $R_2(x_1)$ since the trait value for X does not change, and creates new objects $R_1(x_1,y_2)$ and $Eval(x_1,y_2)$. The third request at time T3 uses the same trait values as the first, and hence is able to reuse all three objects that were created during the processing of the first request. Overall, two RuleEvaluation objects are created, corresponding to the two trait value vectors $\{x_1,y_1\}$ and $\{x_1,y_2\}$. They share a single object for rule R_2, but use two separate objects for rule R_1.

5.2.3 Additional Cache Hierarchies

Multiple rules often depend on the same part of the state. For example, multiple rules need to track whether machines are empty (e.g., BestFit and a rule that attempts to balance capacity across racks). For such cases, we encapsulate the shared part of the state in a *Shared-Cache* type, which multiple rules can refer to. Just like a Rule, a Shared-Cache implements the *Update* method, and declares any request traits it depends on. Shared-Caches play a huge role in reducing memory usage. These objects are cached in their own constant size memory pool. As mentioned, a cached object may depend on other

cached objects. The rule selection engine thus ensures that all dependencies are updated before the object is updated. Our cache hierarchy embeds desirable properties. For example, when a new RuleEvaluation object is created, it may often rely on existing rule and shared-cache objects.

5.2.4 Efficiently Updating the Cache

Tracking and updating mechanisms. Recall that each AA maintains its own private caches. Since each cached object can in principle be updated just before it is used, objects can exist at different levels of staleness. To facilitate seamless updates, each AA has a *journal* that keeps track of changes to any machine in the inventory. The journal maintains a global revision number which is incremented upon every update. In addition, the journal stores only the latest machine state for every machine. Every cached object stores the highest revision number it has seen, which corresponds the latest inventory update it has consumed. An object brings itself up to date by reading only the journal records with higher revision numbers. Consequently, the update operation has runtime complexity at the order of the number of machines that were *modified*, rather than the entire inventory. The AA updates the journal during an ongoing evaluation to record each VM placement decision that it is making. In addition, the AA updates the journal between evaluations by processing enqueued incoming changes from the pub/sub service or the placement store.

Background updates. An up-to-date cache can handle a request in few ms by simply extracting the best machine(s). However, when this is not the case, just-in-time cache update times can be a significant part of the total VM request latency. Nonetheless, because the system has multiple AAs that are provisioned to handle the rare periods of peak load, they remain inactive for most of the time. Hence, when an AA has no requests to process, it is used to opportunistically update the caches (starting with RuleEvaluation cache objects and proceeding recursively).

5.2.5 Discussion

Design advantages. One clear advantage of our caching approach over other alternatives (e.g., node sampling or strict partitioning the inventory) is that Protean can sustain low latency and high throughput without giving up on allocation opportunities. Another appealing property of our implementation is that the complexity of creating, reusing, and updating a rule object is almost completely hidden from the creator of a rule. A rule only has to implement the IsValid (or Compare) and Update methods and declare the request traits it depends on. The rest is handled by the machine selection engine within the AA. This clear separation between the rules and the evaluation engine has been instrumental in the extensibility and adaptability of Protean.

Global rules. There are a few machine selection rules that do not express preference for individual machines, but rather among groups of machines (e.g., prefer the least used rack). We refer to such rules as *global* rules. Most global rules reason about clusters and hence are part of cluster selection. However, a few rules also reason about racks, and hence are part of the machine selection stage. Global rules require us to adjust our caching methodology. To understand the issue, observe that for such rules, a change in a single machine within the group might impact the value of all other machines in that group (e.g., an allocation to a single machine in a rack might make the rack less attractive than another rack). Hence, a single machine change makes *all* machines in the group 'dirty', and the cached objects would require updating all the machines in the group. To still benefit from our caching infrastructure, we use a divide and conquer approach: in a nutshell, we divide the machine inventory into *cells*. Each cell consists of a subset of machines who are considered identical from the perspective of all the global rules. We apply our caching mechanisms separately for each cell; that means that we maintain a sorted list of valid machines (the filtering and sorting is done based on all non-global rules). To obtain the actual evaluation result, we pick the best machine from each cell, and do the required comparisons and sorting based on all rules. While these comparisons slow the evaluation time, we note that the original complexity term of $N \log N$ reduces to $N_c \log N_c$, where N_c is the number of cells. In our current setting, cells correspond to racks. The number of racks after cluster selection is in the order of a hundred, hence $N_c \ll N$.

5.3 Conflict Detection and Reduction

Occasional spikes of thousands of requests push all AAs to work at full tilt. Naturally, chances of commit failures due to conflicts increase considerably during such periods. Conflicts reduce the effective throughput and increase outright failures; as a request fails after a fixed number of retries. We employ the following strategies to reduce such failures.

Fine-grained conflict detection. We built a conflict-detection mechanism which allows commits to succeed even when the AA makes a placement decision based on an out-dated view of a machine. The logic verifies that the new placement decision does not over-commit the machine resources or violate other anti-colocation constraints (such as placing a new VM on a machine that already hosts a VM requiring isolation). If so, it merges the new placement decision with the current state of the machine as part of the commit. This mechanism has led to 25% drop in conflicts in one of our busiest zones, compared to the simpler strategy of rejecting all out-dated placement decisions.

Trading allocation quality for conflict reduction. Conflicts increase during high-load periods, not only because of rapid inventory changes, but also because AAs apply the same logic; AAs are likely to identify highly overlapping sets of best ma-

chines if their respective requests are similar. The number of machines in this set might be very small, so that even a random selection from it may lead to a conflict with high probability. To address this challenge, the AA employs an hybrid strategy for selecting a machine in its final step. In periods with no conflicts, it selects from the set of best machines. Alternatively, it may use a more permissive *conflict-avoidance* scheme; this scheme randomly selects a machine from the top N_0 machines, where N_0 is a configurable parameter (note that the top N_0 machines may differ in their desirability). The conflict-avoidance scheme is enabled with a probability proportional to the ratio of conflict failures to total commit attempts, measured using a rolling window of the most recent commit attempts. The conflict-avoidance scheme is instrumental in satisfying demand at high-load periods; since such periods are infrequent, it has little effect on allocation quality.

6 Evaluation

6.1 Methodology

Production measurements. Since Protean is fully deployed across all regions of Azure, it is natural to evaluate it using measurements from production. Our infrastructure collects numerous diagnostic metrics and structured logs, which are used for monitoring and evaluation. These metrics and logs are aggregated into a central and easily accessible source, which allows custom queries for specific data extraction. Production measurements is the default method in our evaluation; we will mention explicitly when we use simulations.

Simulations. Simulations are incredibly useful for evaluating what-if scenarios, such as the effect of different inventory sizes, different rule configurations, etc. Our simulations use real traces and configurations as input, and can be considered a reasonably accurate representation of reality. In particular, the simulated workload includes both traces from production, as well as realistic probabilistic models of VM requests, derived from historical traces. We built two types of simulators. The *high-fidelity* simulator uses the actual production code of Protean to perform the allocations, and outputs large amounts of data for debugging purposes. Our *low-fidelity* simulator includes a lightweight emulation of the allocator (e.g., supports a subset of the rules). This simulator still provides an excellent approximation of the system, is orders-of-magnitude faster, and especially useful for large scale evaluations.

6.2 Performance and Scale

Here we evaluate key mechanisms and design choices that help Protean scale. We focus on the caching mechanism and the effect of multiple AAs.

Cache evaluation: Hit-rate, latency and update overhead. As discussed in §2, the nature of our workload motivates the use of caching. Our first goal is to understand the effectiveness of the hierarchical cache architecture. Towards that end, Fig. 9 shows typical hit-ratio patterns in one of our zones, focusing on four different cacheable classes over a day period (taking into account all cached objects for each class). The FdEmptyNodes and BestFit are cached rules; *NodeResources* is a Shared-Cache; and *RuleEvaluation* corresponds to the RuleEvaluation class. The stacked bar to the right of the figure shows the frequency of cache requests as a percentage of allocation requests (NodeResources cache is requested only in 0.16% of allocations, hence barely noticed). Lower-level caches are only requested when higher-level caches miss, so to interpret the results, both the request frequency and hit-ratio should be considered. For example, the NodeResources cache has a hit-ratio around 20%, but is less frequently accessed – compared to RuleEvalution, which has a much higher hit-ratio and is accessed on every request.

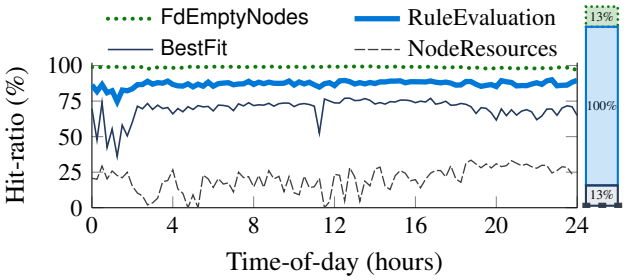

Figure 9: Cache-hit ratio over time for some caches. The frequency of requests for each cache is shown on the right.

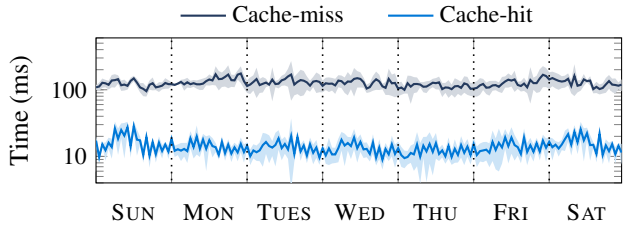

Figure 10: Latency in a large zone; average (+standard deviation) per hour per day, over two months.

The resulting benefit of our cache and high hit-rates for evaluation caches is improved latency. This can be clearly seen from Fig. 10, which depicts the effect of a cache hit/miss for the RuleEvaluation cache. Given our high hit-rate, the overall average latency is close to 20 ms per allocation. A cache-miss still uses many lower-level caches so that the latency is typically 70-80ms. We note that Protean's latency is affected by additional functionality beyond the allocation process itself, such as tracking and outputting debug information about every allocation.

To gain further insights into the cache operation and resulting latencies, we track the average number of machines updated per allocation in one of our zones (∼30k machines), over an entire day. Cache hits/misses have a significant impact on the number of updated machines: approximately 50 for a

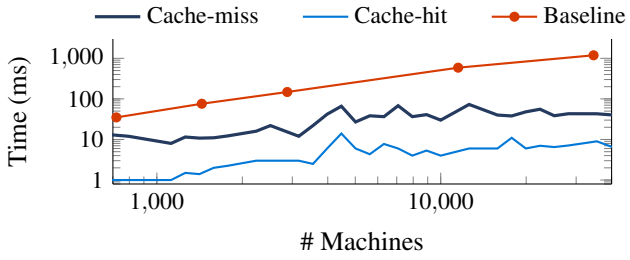

Figure 11: Median latency vs inventory size over one-month. Baseline results are based on high-fidelity simulation.

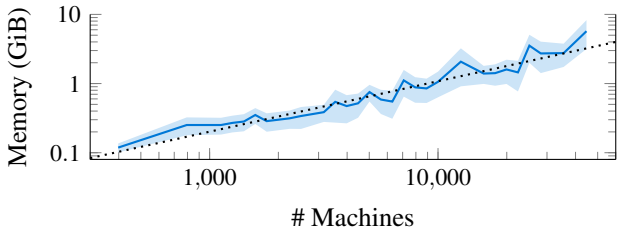

Figure 12: Memory used per AA vs inventory size. Average (+standard deviation) over a month over all zones.

hit verses 2000 for a miss. A hit thus translates to less overhead in the allocation process and in turn, to lower latency. Notably, the number of updates is relatively small even in case of a miss (6.6% of the machines). This can be attributed to two main factors. First, the cluster selection process filters out a large part of the inventory. In particular, we observe through simulations that cluster selection rules filter out up to 20% of the inventory for zones of 10k machines or more. Second, a substantial part of inventory updates occur asynchronously via background resolve. Indeed, our production measurements indicate that 80-96% of machine updates are done by that mechanism.

Scaling with inventory. Fig. 11 shows the inventory size effect on latency under three different scenarios. The first two scenarios correspond to cache hit or miss for RuleEvaluation cache, where the results represent production measurements of zones with different sizes. To further examine the effectiveness of mechanisms, we include a third scenario ("baseline") where the caching and cluster selection are disabled; because we would rather not disable these mechanisms in production, we use our high-fidelity simulator to obtain the results; observe that the median latency reaches around 1000 ms for larger zones. In addition to latency performance, it is also important to examine the cache's memory footprint. Fig. 12 shows the memory required per AA, as a function of inventory size. The fitted line, obtained via regression, shows that the growth of our the memory footprint is sublinear ($\sim x^{0.73}$), which helps keep memory sizes manageable at scale.

Multiple allocation agents. Multiple allocators influence important metrics, such as the number of conflicts, delay and

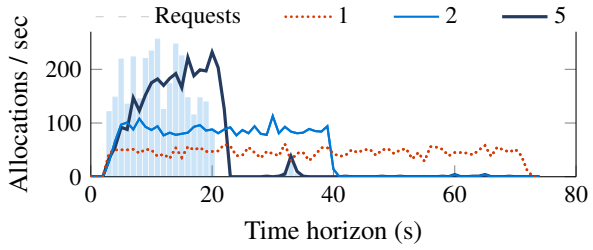

Figure 13: [Simulation] Throughput (allocations per second) over a selected time horizon, for various number of AAs. The bars represent the requests, and the curves represent their processing. A shorter x-axis "span" means better performance.

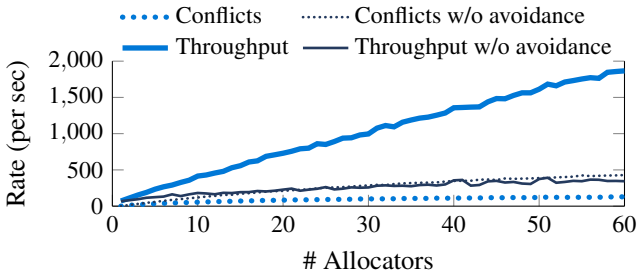

Figure 14: [Simulation] 99th percentile of conflicts per second and corresponding throughput. Uses production trace as input.

throughput. Since the number of AAs in production is fixed, we use our low-fidelity simulator for the experiments: we emulate an heterogeneous inventory of one of our zones with nearly 25k machines. We replay an actual request trace of an entire day. We use the same conflict-avoidance (see §5.3) parameters (100 machines allowed for final random selection, rolling window size of 50 commits) and number of retries before rejection (20) as in production. To expedite the low-fidelity simulations, the actual cache infrastructure is not integrated in the simulator. To mimic the cache, we use a realistic statistical model, derived from production measurements of the same zone over a month period. A cache hit/miss is determined using a Bernoulli random variable, with an average hit-ratio ($p = 0.9$) obtained from production; a hit results in 14ms latency, whereas a miss incurs a higher latency of 88ms.

Our first experiment examines how different number of AAs handle a spike in demand (see Fig. 13). The key takeaway here is that a single allocator struggles to satisfy the load in a reasonable time, causing excess delays to requests. With five AAs, the requests are handled in more than 3x less time (adding more AAs yields similar results). Our second experiment (Fig. 14) replays another trace from the same zone; the figure depicts the 99th percentile for the conflicts per second, as well as the throughput observed during the same time. Our collision-avoidance strategy provides clear gains: note that throughput increases substantially with the number of AAs with little effect on conflicts.

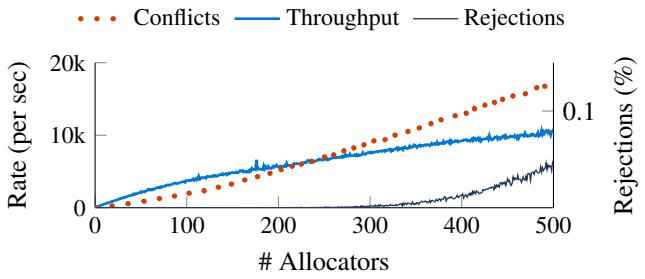

Figure 15: [Simulation] Scale test. 99th percentile of conflicts per second with corresponding throughput and rejection rate.

Our system easily handles the request volume currently seen in production. It is of natural interest to study (via simulations) the possible performance trends at higher scale. Towards that end, we take an existing request trace and speed up time by factors of up to 1000x. Fig. 15 shows throughput, conflicts and rejection rates as we scale up the demand, verses the number of AAs handling the requests; the simulations use our standard avoidance scheme. We observe that although throughput reaches over 10k requests per second, this comes at the expense of significant conflicts which in turn affect the rejection rate. The throughput eventually plateaus, likely due to a combination of fixed inventory size and increasing conflicts. In a production setting, we would have several options to deal with increased demand (e.g., tune the avoidance scheme, or longer-term increase of inventory size).

6.3 Allocation Quality

Quality vs. performance tradeoff. There are different criteria for quantifying quality; for example, balancing allocations across multiple fault domains is important for satisfying large service requests. In this section, we zoom in on a key efficiency metric – *packing density* – which measures the average number of allocated cores on non-empty machines (certain machines must be kept empty, e.g., for failover of large VMs). Formally, the packing density at time t is the ratio between the number of allocated cores, and the number of non-empty machines times the number of cores in each machine. We note that packing density can be defined similarly for other resources, such as memory; we focus on CPU because it is typically the bottleneck resource. Table 5 summarizes a set of experiments in one of our zones, using the low-fidelity simulator on a 5 month trace. The different rows correspond to different parameter configurations of the BestFit rule; in particular, the configurations differ by the number of buckets (see §3.2), where ∞ means no quantization. Recall that the more buckets we use the finer is the quantization of the score, which allows for better discrimination of machines by the packing quality. On the flip side, a finer quantization means that downstream rules are left with fewer candidate machines. The results demonstrate some interesting trends. As expected, the packing density (denoted PD %) increases with the number of buckets; note that the most significant increase is from two to three buckets. More buckets increases the packing density by a little, however at the cost of filtering out a substantial percent of candidate machines (Post-BestFit). The effect is magnified at the extreme of no quantization, where very few candidate machines are left. As a consequence, not only downstream rules become meaningless, but also the conflict rate increases. This is because different allocators are more likely to pick the same machine for allocation. In view of the above analysis, we currently use three buckets in production.

Buckets	PD (%)	Post-BestFit (%)	P_{99} Conflicts / min
1	83.5	27.6	13.4
2	84.3	25.0	13.5
3	86.3	21.0	11.6
4	87.3	16.5	12.0
5	87.8	13.7	10.7
∞	89.1	2.3	18.0

Table 5: [Simulation] The trade-off between packing and robust allocations. PD (%) is the packing density averaged over five months.

Adapting to COVID-19 capacity crunch. As a consequence of the COVID-19 pandemic, Azure observed a sharp increase in demand. As an immediate response, we increased the utilization limits in each cluster by 1%. These limits are used to leave enough buffers for in-cluster scale-outs as well as to account for failures. The increase was done easily by modifying configurable threshold values in a cluster validator rule IsClusterBelowLimit(x,v). This limit change slightly increased the risk for scale-outs and fail-overs. To mitigate the risk, we used Protean to identify fragmented machines, and recommend migration targets that would improve packing (what-if analysis). A supplementary VM migration mechanism used these recommendations to live-migrate some VMs (targeting first-party VMs only), resulting in improved packing density. Fig. 16 shows both the average utilization (i.e., ratio between number of allocated cores and total number of cores in Azure) and the packing density over our entire fleet. The dashed lines indicates the point of time at which the above changes were made. The net effect of Protean adaptation was a sizeable increase in utilization, facilitated by a significant improvement in packing density. We also depict in the same figure the relative trends for the overall capacity fulfillment rate (CFR) – the fraction of allocations that are successfully deployed. CFR dipped slightly in mid-March, but went up again exceeding its target of four nines by mid-April.

7 Related Work

Resource management for large compute clusters. Numerous systems have been implemented for various domains, including batch scheduling for HPC applications

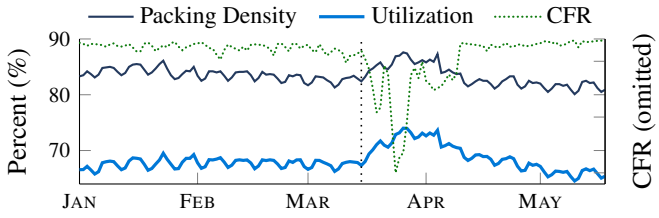

Figure 16: CPU usage across Azure. Absolute capcity fulfillment rate (CFR) values are omitted for confidentiality (hence, the CFR y-axis is not labeled). The curve represents the relative trend and scaled to fit the graph.

[25, 44, 45], big-data analytics [7, 20, 23, 28, 38, 40, 47, 52], stream-processing [34], AI [36], etc. More related to our context is the work on hyper-scale cloud computing clusters, see [8, 12, 15–17, 43, 48] and references therein. Our cloud workload analysis adds to a body of work on this topic, e.g., [5, 10, 12, 18, 26, 30, 35, 42, 46, 50].

Scheduler types. One useful way to classify large-scale schedulers is based on how they process work items (jobs, tasks, VMs, etc.). *Centralized monolithic* schedulers [21, 24, 52] use a single agent to process requests. They avoid concurrency issues, yet are harder to scale. A subset of these schedulers, optimizes placement decisions by *batch-processing* multiple jobs together [19, 21, 24]. Our demanding latency and throughout requirements preclude using these approaches. To cope with scale and management complexities, *two-level* schedulers [23, 47] perform course-grained resource management, while leaving the fine-grained scheduling to application frameworks. Similarly, *distributed schedulers* [36, 38, 41] decentralize the scheduling logic by employing sophisticated queue management strategies at the target machines (see also works on *hybrid schedulers* [13, 14, 29]). Two-level or various distributed approaches are less applicable for VM scheduling, which is inherently IaaS-centric. Our AA is centralized, while target machines create their assigned VMs according to a simple FCFS policy.

Concurrent schedulers. Similar to Sparrow [38], Apollo [7] and Omega [43], Protean is a concurrent scheduler which employs multiple agents over a shared inventory. Omega handles conflicts immediately as part of scheduling, whereas Apollo and Mercury allow conflicting scheduling decisions to queue on target nodes while deferring conflict resolution. As in [43], we use multiple concurrent allocation agents and a conflict resolution model. Indeed, our customers prefer VM requests to fail early rather than waiting longer in hope for success; this allows higher level services to quickly try other alternatives, such as using another zone or modifying some request properties.

Allocation scope. Cluster selection and caching allow Protean to make resource assignment decisions based on the entire inventory, similar to [7, 15]. Alternatively, schedulers can statically partition the inventory [49], or use random sampling to make a decision using a subset of the inventory [17, 38, 48]. Protean shares similarities with Google's Borg [48]. Borg employs other optimizations for scalability, such as caching node preference scores until the node changes, and avoiding duplicate work by evaluating decisions for only a single task within a group of identical tasks. Protean caches not only node-centric data, but also rule and evaluation outcomes that can be used across different requests. In addition, Borg introduces the notion of equivalent classes, where feasibility and scoring is determined only for a single task out of identical tasks in a job. Protean extends this idea by considering requests across tenants to be equivalent if they share the same trait values. Finally, unlike Borg, we do not employ sampling (termed "relaxed randomization"), but rather use other techniques to help with scale (multi-layer caching and cluster selection).

Resource efficiency. Cloud schedulers attempt to increase actual resource usage through a variety of techniques, e.g., reclaiming unused resources, harvesting, profiling, heterogeneity and interference awareness [9, 12, 15, 16, 22, 27, 32, 33, 37, 48, 51, 53]. Protean's flexible rule-based logic facilitates dynamic resource adjustment and interference mitigation strategies; their description is outside the scope of this paper.

8 Conclusion

We describe Protean, the VM allocation service of Azure. Our design separates policy from mechanisms, which has allowed us to successfully expand our VM offerings over the years. A flexible rule-based allocator facilitates refining the allocation logic and explaining it to customers. VM requests are processed in milliseconds, due to a hierarchical caching framework. Results from production demonstrate that Protean sustains adequate trade-offs between performance and quality.

Acknowledgements

We are grateful to our shepherd, John Wilkes, and the anonymous reviewers for their detailed and thoughtful feedback. We would like to acknowledge the contributions of the engineers who have been involved in the design, implementation, and maintenance of Protean over the years - Jason Chu, Chris Cowdery, Dustin Dobransky, Eric Hao, Ryan Hidalgo, Valentina Li, John Miller, Mukund Nigam, Jason Seo, Kanishk Thareja, Karel Trueba, Yiran Wei and Brian Yan. We also thank Saurabh Agarwal, Girish Bablani, Ricardo Bianchini, Íñigo Goiri, Marcus Fontoura and Saad Syed for helpful discussions.

References

[1] Azure batch, 2020. https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms.

[2] Azure spot virtual machines, 2020. https://azure.microsoft.com/en-us/pricing/spot/.

[3] Azure VM packing trace (public dataset), 2020. https://github.com/Azure/AzurePublicDataset.

[4] Azure's virtual machine scale sets, 2020. https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/virtual-machine-scale-sets-autoscale-overview.

[5] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *USENIX Annual Technical Conference (ATC)*, pages 533–546. USENIX Association, July 2018.

[6] Yossi Azar and Danny Vainstein. Tight bounds for clairvoyant dynamic bin packing. *ACM Trans. Parallel Comput.*, 6(3):15:1–15:21, 2019.

[7] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–300, 2014.

[8] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Queue*, 14(1):70–93, January 2016.

[9] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. Long-term SLOs for reclaimed cloud computing resources. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 1–13, 2014.

[10] Yue Cheng, Zheng Chai, and Ali Anwar. Characterizing co-located datacenter workloads: An Alibaba case study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys)*, 2018.

[11] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. Dynamic bin packing. *SIAM Journal on Computing*, 12(2):227–258, 1983.

[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 153–167, 2017.

[13] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC)*, page 497–509, 2016.

[14] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC)*, page 499–510, 2015.

[15] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 77–88, 2013.

[16] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 127–144, 2014.

[17] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 97–110, 2015.

[18] Sheng Di, Derrick Kondo, and Walfredo Cirne. Characterization and comparison of cloud versus grid workloads. In *Proceedings of the 2012 IEEE International Conference on Cluster Computing (CLUSTER)*, page 230–238, 2012.

[19] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[20] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *2015 USENIX Annual Technical Conference (ATC)*, pages 459–471. USENIX Association, July 2015.

[21] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 99–115, 2016.

[22] Jaeung Han, Seungheun Jeon, Young-ri Choi, and Jaehyuk Huh. Interference management for distributed parallel applications in consolidated clusters. In *Proceedings of the Twenty-First International Conference*

on Architectural Support for Programming Languages and Operating Systems (ASPLOS), page 443–456, 2016.

[23] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Symposium on Networked Systems Design and Implementation (NSDI)*, volume 11, pages 22–22, 2011.

[24] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, page 261–276, 2009.

[25] David B. Jackson, Quinn Snell, and Mark J. Clement. Core algorithms of the Maui scheduler. In *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, page 87–102. Springer-Verlag, 2001.

[26] C. Jiang, G. Han, J. Lin, G. Jia, W. Shi, and J. Wan. Characteristics of co-allocated online services and batch jobs in internet data centers: A case study from Alibaba cloud. *IEEE Access*, 7:22495–22508, 2019.

[27] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. Improving resource utilization by timely fine-grained scheduling. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[28] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo 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*, pages 117–134, 2016.

[29] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC)*, page 485–497, 2015.

[30] Cinar Kilcioglu, Justin M. Rao, Aadharsh Kannan, and R. Preston McAfee. Usage patterns and the economics of the public cloud. In *Proceedings of the 26th International Conference on World Wide Web (WWW)*, page 83–91, 2017.

[31] Alok Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Frujeri, Nithish Mahalingam, Pulkit Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, and Ricardo Bianchini. Prediction-based power oversubscription in cloud platforms, 2020.

[32] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, page 619–630, 2013.

[33] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, page 248–259, 2011.

[34] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, and Guoqiang Jerry Chen. Turbine: Facebook's service management platform for stream processing. In *Proceedings of the 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020.

[35] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. Towards characterizing cloud backend workloads: Insights from Google compute clusters. *SIGMETRICS Perform. Eval. Rev.*, 37(4):34–41, March 2010.

[36] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 561–577, 2018.

[37] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, page 184–200, 2017.

[38] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, page 69–84, 2013.

[39] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. Technical report, 2011. Available from https://www.microsoft.com/en-us/research/wp-content/uploads/2011/01/VBPackingESA11.pdf.

[40] Hang Qu, Omid Mashayekhi, David Terei, and Philip Levis. Canary: A scheduling architecture for high performance cloud computing. *arXiv preprint arXiv:1602.01412*, 2016.

[41] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[42] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC)*, 2012.

[43] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 351–364, 2013.

[44] Garrick Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC)*, page 8–es, 2006.

[45] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[46] 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 (EuroSys)*, pages 1–14, 2020.

[47] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*, pages 1–16, 2013.

[48] 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 (EuroSys)*, pages 1–17, 2015.

[49] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. Pigeon: An effective distributed, hierarchical datacenter job scheduler. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, page 246–258, 2019.

[50] John Wilkes. More Google cluster data. Google research blog, Nov 2011. Available from https://ai.googleblog.com/2011/11/more-google-cluster-data.html.

[51] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, page 607–618, 2013.

[52] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, page 265–278, 2010.

[53] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 755–770, 2016.