Autopilot: workload autoscaling at Google

Krzysztof Rzadca
Google + Univ. of Warsaw
kmrz@google.com

Pawel Findeisen
Google
pafinde@google.com

Jacek Swiderski
Google
jswiderski@google.com

Przemyslaw Zych
Google
pzych@google.com

Przemyslaw Broniek
Google
broniek@google.com

Jarek Kusmierek
Google
jdk@google.com

Pawel Nowak
Google
pawelnow@google.com

Beata Strack
Google
bstrack@google.com

Piotr Witusowski
Google
witus@google.com

Steven Hand
Google
sthand@google.com

John Wilkes
Google
johnwilkes@google.com

Abstract

In many public and private Cloud systems, users need to specify a *limit* for the amount of resources (CPU cores and RAM) to provision for their workloads. A job that exceeds its limits might be throttled or killed, resulting in delaying or dropping end-user requests, so human operators naturally err on the side of caution and request a larger limit than the job needs. At scale, this results in massive aggregate resource wastage.

To address this, Google uses *Autopilot* to configure resources automatically, adjusting both the number of concurrent tasks in a job (horizontal scaling) and the CPU/memory limits for individual tasks (vertical scaling). Autopilot walks the same fine line as human operators: its primary goal is to reduce *slack* – the difference between the limit and the actual resource usage – while minimizing the risk that a task is killed with an out-of-memory (OOM) error or its performance degraded because of CPU throttling. Autopilot uses machine learning algorithms applied to historical data about prior executions of a job, plus a set of finely-tuned heuristics, to walk this line. In practice, Autopiloted jobs have a slack of just 23%, compared with 46% for manually-managed jobs. Additionally, Autopilot reduces the number of jobs severely impacted by OOMs by a factor of 10.

Despite its advantages, ensuring that Autopilot was widely adopted took significant effort, including making potential recommendations easily visible to customers who had yet to opt in, automatically migrating certain categories of jobs, and adding support for custom recommenders. At the time of writing, Autopiloted jobs account for over 48% of Google's fleet-wide resource usage.

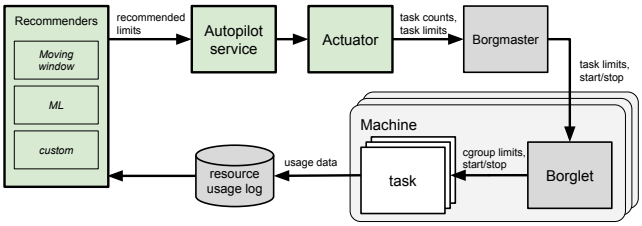


Figure 1. Autopilot dataflow diagram.

ACM Reference Format:
Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387524

1 Introduction

Many types of public and private Cloud systems require their users to declare how many instances their workload will need during execution, and the resources needed for each: in public cloud platforms, users need to choose the type and the number of virtual machines (VMs) they will rent; in a Kubernetes cluster, users set the number of pod replicas and resource limits for individual pods; in Google, we ask users to specify the number of containers they need and the resource limits for each. Such limits make cloud computing possible, by enabling the Cloud infrastructure to provide adequate performance isolation.

But limits are (mostly) a nuisance to the user. It is hard to estimate how many resources a job needs to run optimally: the right combination of CPU power, memory, and the number of concurrently running replicas. Load tests can help

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04.
https://doi.org/10.1145/3342195.3387524

find an initial estimate, but these recommendations will become stale as resource needs change over time because many end-user serving jobs have diurnal or weekly load patterns, and traffic changes across longer time scales as a service becomes more or less popular. Finally, the resources needed to handle a given load vary with new features, optimizations and updates of the underlying software or hardware stack. Exceeding the requested resources can result in poor performance if the CPU is capped, or cause a task to be killed because it runs out of memory (an OOM). Neither is good.

As a result, a rational user will deliberately overestimate the resources their jobs need, resulting in poor utilization of physical resources. One analysis [26] of a month-long trace of jobs executed at one of Google's clusters [27] shows 50% average memory utilization; another analysis [23] of Alibaba's YARN cluster showed tasks' peak memory utilization never exceeding 80%.

In response to the difficulties in configuring resources, a common pattern is to adopt a *horizontal autoscaler*, which scales a job by adding or removing replicas in response to changes in the end-user traffic, or the average CPU utilization. All major cloud providers (AWS, Azure and GCP) provide a horizontal autoscaling function; it is also available in some Cloud middleware, such as Kubernetes. A less common pattern is to use *vertical autoscaling* to tune the amount of resources available to each replica. The two techniques can also be combined.

Autopilot is the primary autoscaler that Google uses on its internal cloud. It provides both horizontal and vertical autoscaling. This paper focuses on Autopilot's vertical scaling of memory, since this is less commonly reported. The paper:

- Describes Autopilot, and the two main algorithms it uses for vertical autoscaling: the first relies on an exponentially-smoothed sliding window over historic usage; the other is a meta-algorithm based on ideas borrowed from reinforcement learning, which runs many variants of the sliding window algorithm and chooses the one that would have historically resulted in the best performance for each job;
- Evaluates the effectiveness of Autopilot's algorithms on representative samples of Google's workload; and
- Discusses the steps we took to have Autopilot widely adopted across our fleet.

2 Managing cloud resources with Borg

Autopilot's goals and constraints follow from Google's Borg infrastructure, and it is tuned for Google's workload. We provide a brief overview of both here: for more details about Borg see [34], and for more detailed information about the workload see [26, 27, 31, 35].

2.1 Machines, jobs and tasks

The Google compute infrastructure is made up of many clusters, spread in multiple geographical locations. A median cluster has roughly 10 000 physical machines, and runs many different kinds of workloads simultaneously. A single physical machine might simultaneously compute a memory- and CPU-heavy batch computation, store and serve queries for a slice of a memory-resident database and also serve latency-sensitive end-user requests.

We call a particular instance of a workload a *job*. A job is composed of one or more *task*s. A *task* is executed on a single physical machine; a single machine executes multiple tasks concurrently. A job is a logical entity that corresponds to a service with some functionality (e.g., a filesystem or an authentication service); tasks do the actual work, such as serving end-user or file-access requests. It is not unusual for a job to stay up for months, although during this time we might perform multiple roll-outs of the binary that the task runs. During such a roll-out, new tasks gradually replace tasks running the older binary.

The workloads we run can be split into two categories: *serving* and *batch*. Serving jobs generally aim at strict performance guarantees on query response time (e.g., a request-latency service level objective or SLO of $\leq$ 50 ms at the 95%ile). Such tight latency requirements preclude any in-band resource-allocation decisions beyond those of the OS kernel, so serving jobs have the resources they request explicitly set aside for them. In contrast, batch jobs aim to finish and exit "quickly", but typically have no strict completion deadlines. Serving jobs are the primary driver of our infrastructure's capacity, while batch jobs generally fill the remaining or temporarily-unused capacity, as in [4].

An out-of-memory (OOM) event terminates an individual task. Some jobs are reasonably tolerant of OOM events; some are not at all; and some fall in between. Overall, jobs composed of more tasks and having less state experience less service degradation when an individual task terminates, thus are more OOM-tolerant. Some jobs require low, repeatable latency to serve requests; some do not. Autopilot chooses defaults based on a job's declared size, priority and class, but permits our users to override them. Borg evicts tasks in order to perform security and OS updates, to make space for higher-priority tasks, and to let us pack work onto the machines more efficiently. We consciously share the burden of providing service resiliency between the compute cluster infrastructure and our applications, which are expected to run additional tasks to work around evictions and hardware failures. Borg publishes an expected maximum rate of these evictions, and the difference between the observed eviction rate and this gives us the freedom to perform experiments with a task's resource settings – an occasional OOM while we learn what a task needs is OK. (Tools such as VM live

migration are used to hide these internal optimizations from external Cloud VMs.)

A typical serving job has many tasks and a load balancer that drives the traffic to the available ones, so losing a task simply causes its load to be spread to others. This is normal, not a catastrophic failure, and enables us to be more aggressive about the utilization of our infrastructure, as well as to handle occasional hardware failures gracefully. Similar resilience is built into our batch jobs, using techniques such as those used in MapReduce [6].

2.2 Borg scheduler architecture

Each cluster is managed by a dedicated instance of Borg, our custom-built cluster scheduler. Below, we briefly describe the Borg architecture and features that directly influence Autopilot design; refer to [34] for a complete description.

Borg has a replicated *Borgmaster* that is responsible for making scheduling decisions, and an *agent* process called the Borglet running on each machine in the cluster. A single machine simultaneously executes dozens of tasks controlled by the Borglet; in turn, it reports their state and resource usage to the Borgmaster. When a new job is submitted to the Borgmaster, it picks one or more machines where there are sufficient free resources for tasks of the newly submitted job – or creates this situation by evicting lower-priority tasks to make space. After the Borgmaster decides where to place a job's tasks, it delegates the process of starting and running the tasks to the Borglets on the chosen machines.

2.3 Resource management through task limits

To achieve acceptable and predictable performance, the tasks running on a machine must be isolated from one another. As with Kubernetes, Borg runs each task in a separate Linux container and the local agent sets the container resource limits to achieve performance isolation using cgroups. Unlike traditional fair-sharing at the OS level, this ensures that a task's performance is consistent across different executions of the same binary, different machines (as long as the hardware is the same) and different neighbors (tasks co-scheduled on the same machine) [38].

In our infrastructure, CPU and RAM are the key resources to manage. We use the term *limit* to refer to the maximum permitted amount of each resource that may normally be consumed. Since Borg generally treats the jobs' tasks as interchangeable replicas, all tasks normally have the same limits.

A job expresses its CPU limit in normalized milli-cores [31], and this limit is enforced by the standard Linux kernel cgroups mechanism. If there is little contention (as measured by the overall CPU utilization), tasks are allowed to use CPU beyond their limits. However, once there is contention, limits are enforced and some tasks may be throttled to operate within their limits.

A job expresses its memory limit in bytes. As with the standard Linux cgroups, the enforcement of a job's RAM limit might be *hard* or *soft*, and the job's owner declares the enforcement type when submitting the job. A task using a hard RAM limit is killed with an out-of-memory (OOM) error as soon as the task exceeds its limit, and the failure is reported to the Borgmaster.[1] A task using a soft RAM limit is permitted to claim more memory than its limit, but if the overall RAM utilization on a machine is too high, the Borglet starts to kill tasks that are over their limit (with an OOM error) until the machine is deemed no longer at risk (*cf.* the standard cgroup enforcement that simply prevents over-the-limit containers reserving more memory).

Borg allows a job to modify its resource requirements while the job is running. In *horizontal scaling* a job can dynamically add or remove tasks. In *vertical scaling*, a job can change its tasks' RAM and CPU limits. Increasing a job's RAM and CPU limits is a potentially costly operation, because some tasks might then no longer fit onto their machines. In such cases, the Borglets on these machines will terminate some lower-priority tasks; these tasks, in turn, will get rescheduled to other machines and may trigger additional terminations of even lower priority tasks. (A few years ago, we drastically reduced the effective number of priority levels to reduce the amount of this kind of cascading.)

Although it is common to over-provision a job's limits, there is some back-pressure: we charge service users for the resources they reserve, rather than the ones they use, and the requested resources decrement the user's quota – a hard limit on the aggregate amount of resources they can acquire, across all their jobs, in a cluster. This is similar to the use of pricing and quotas for VMs in public clouds. Charging and quotas both help, but in practice have only limited effect: the downsides of under-provisioning typically far outweigh the benefits obtained by requesting fewer resources. We have found this to be a recurring theme: theoretically-obtainable efficiencies are often hard to achieve in practice because the effort or risk required to do so manually is too high. What we need is an automated way of making the trade-off. This is what Autopilot does.

3 Automating limits with Autopilot

Autopilot uses vertical scaling to fine-tune CPU and RAM limits in order to reduce the *slack*, i.e., the difference between the resource limit and the actual usage, while ensuring that the tasks will not run out of resources. It also uses horizontal scaling (changing the number of tasks in a job) to adjust to larger-scale workload changes.

[1] The standard Linux cgroup behavior is to kill one of the processes executing inside a container, but to simplify failure handling, the Borglet kills all the task's processes.

3.1 Architecture

The *functional* architecture of Autopilot is a triple of closed-loop control systems, one for horizontal scaling at the per-job level, the other two for vertical scaling of per-task resources (CPU and memory). The algorithms (described in detail in subsequent sections) act as controllers. Autopilot considers jobs separately – there is no cross-job learning.

Autopilot's implementation (Figure 1) takes the form of a collection of standard jobs on our infrastructure: each cluster has its own Autopilot. Each of Autopilot's resource *recommenders* (that size a job according to its historic usage) runs as a separate job, with three task replicas for reliability. A replicated *Autopilot service*, with an elected master, is responsible for selecting the recommender to use for a job and passing (filtered) recommendations to the Borgmaster via an *actuator* job. If the request is to change the number of tasks, the Borgmaster spawns or terminates tasks accordingly. If the request is to change resource limits, the Borgmaster firsts makes any scheduling decisions needed to accommodate them, and then contacts the appropriate Borglet agents to apply the changes. An independent monitoring system keeps track of how many resources each task uses; Autopilot just subscribes to updates from it.

Today, our users explicitly opt-in their jobs to use Autopilot, using a simple flag setting; we are in the process of making this the default, and instead allowing explicit opt-outs. Users may also configure several aspects of Autopilot's behavior, such as: (1) forcing all replicas to have the same limit, which is useful for a failure tolerant program with just one active master; and (2) increasing the limits so that load from a replica job in another cluster will be able to fail over to this one instantaneously.

When a Autopiloted job is submitted to the Borgmaster, it is temporarily queued until Autopilot has had a chance to make an initial resource recommendation for it. After that, it proceeds through the normal scheduling processes.

3.2 Vertical (per task) autoscaling

The Autopilot service chooses the recommender(s) to use for a job according to whether the resource being autoscaled is memory or CPU; how tolerant the job is to out-of-resource events (latency tolerant or not, OOM-sensitive vs. OOM-tolerant); and optional user inputs, which can include an explicit recommender choice, or additional parameters to control Autopilot's behavior, such as upper and lower bounds on the limits that can be set.

3.2.1 Preprocessing: aggregating the input signal. The recommenders use a preprocessed resource usage signal. Most of this preprocessing is done by our monitoring system to reduce the storage needs for historical data. The format of aggregated signal is similar to the data provided in [35].

The low-level task monitoring records a raw signal that is a time series of measurements for each task of a job (e.g., CPU or RAM usage, or the number of queries received). We denote the value recorded by the monitoring system at time τ from task i as $r_i[\tau]$. This time series typically contains a sample every 1 second.

Table 1. Notation used to describe the recommenders

$r_i[\tau]$	a raw, per-task CPU/MEM time series (1s resolution)
$s_i[t]$	an aggregated, per-*task* CPU/MEM time series (histograms, 5 min resolution)
$s[t]$	an aggregated, per-*job* CPU/MEM time series (histograms, 5 min resolution)
$h[t]$	a per-job load-adjusted histogram
$b[k]$	the value of k-th bin (boundary value) of the histogram
$w[\tau]$	the weight to decay the sample aged τ
$S[t]$	the moving window recommendation at time t
m	a model (a parametrized arg min algorithm)
d_m	the decay rate used by model m
M_m	the safety margin used by model m
L	the value of a limit tested by the recommender
$L_m[t]$	the limit recommended by model m
$L[t]$	the final recommendation of the ML recommender
$o(L)$	the overrun cost of a limit L
$u(L)$	the underrun cost of a limit L
w_o	weight of the overrun cost
w_u	weight of the underrun cost
$w_{\Delta L}$	weight of the penalty to change the limit
$w_{\Delta m}$	weight of the penalty to change the model
d	decay rate for computing the cost of a model
$c_m[t]$	the (decayed) historical cost of a model m

To reduce the amount of data stored and processed when setting a job's limits, our monitoring system preprocesses $r_i[\tau]$ into an aggregated signal $s[t]$, which aggregates values over, typically, 5 minute windows. A single sample of the aggregated signal, $s[t]$, is a histogram summarizing the resource usage of all the job's tasks over these 5 minutes.

More formally, for each window t, the aggregated *per-task* signal $s_i[t]$ is a vector holding a histogram over the raw signal $r_i[\tau]$ with $\tau \in t$. For a CPU signal, the elements of this vector $s_i[t][k]$ count the number of raw signal samples $r_i[\tau]$ that fall into each of about 400 usage buckets: $s_i[t][k] = |\{r_i[\tau] : \tau \in t \wedge b[k-1] \leq r_i[\tau] < b[k]\}|$, where $b[k]$ is the k-th bucket's boundary value (these values are fixed by the monitoring system). For a memory signal we record in the histogram just the task's peak (maximum) request during the 5 minute window (i.e., the per-task histogram $s_i[t][k]$ has just one non-zero value). The memory signal uses the peak value rather than the entire distribution because we typically want to to provision for (close to) the peak memory usage: a task is more sensitive to underprovisioning memory (as it would terminate with an OOM) than CPU (when it would be just CPU-throttled).

We then aggregate the per-task histograms $s_i[t]$ into a single per-job histogram $s[t]$ by simply adding $s[t][k] = \sum_i s_i[t][k]$. We do not explicitly consider individual tasks – e.g., by examining extreme values. Since individual tasks

in most Borg jobs are interchangeable replicas, we use the same limits for all of them, except in a few special cases.

3.2.2 Moving window recommenders. The *moving window* recommenders compute limits by using statistics over the aggregated signal s.

We want the limits to increase swiftly in response to rising usage, but reduce slowly after the load decreases to avoid a too-rapid response to temporary downward workload fluctuations. To smooth the response to a load spike, we weight the signal by exponentially-decaying weights $w[\tau]$:

$$w[\tau] = 2^{-\tau/t_{1/2}}, \quad (1)$$

where τ is the sample age and $t_{1/2}$ is the half life: the time after which the weight drops by half. Autopilot is tuned to long-running jobs: we use a 12 hour half life for the CPU signal and a 48 hour half life for memory.

One of the following statistics over s is used to compute the recommendation $S[t]$ at time t:

peak ($S_{\max}$) returns the maximum from recent samples, $S_{\max}[t] = \max_{\tau \in \{t-(N-1),\ldots,t\}}\{b[j] : s[\tau][j] > 0\}$, i.e., the highest value of a non-empty bucket across the last N samples, where N is a fixed system parameter.

weighted average (S_{avg}) computes a time-weighted average of the average signal value:

$$S_{avg}[t] = \frac{\sum_{\tau=0}^{\infty} w[\tau]\overline{s[t-\tau]}}{\sum_{\tau=0}^{N} w[\tau]}, \quad (2)$$

where $\overline{s[\tau]}$ is the average usage of histogram $s[\tau]$, i.e., $\overline{s[\tau]} = \Big(\sum_j (b[j]s[\tau][j])\Big) / \Big(\sum_j s[\tau][j]\Big)$.

j-%ile of adjusted usage (S_{pj}) first computes a load-adjusted, decayed histogram $h[t]$ whose kth element $h[t][k]$ multiplies the decayed number of samples in bucket k by the amount of load $b[k]$:

$$h[t][k] = b[k] \cdot \sum_{\tau=0}^{\infty} w[\tau] \cdot s[t-\tau][k]; \quad (3)$$

and then returns a certain percentile $P_j(h[t])$ of this histogram. The difference between h and the standard histogram s is that in s every sample has the same, unit weight, while in h the weight of the sample in bucket k is equal to the load $b[k]$.

Note that a given percentile of the load-adjusted usage S_{pj} can differ significantly from the same percentile of usage over time. In many cases, we want to ensure that a given percentile of the offered load can be served when the limit is set to accommodate the offered load, rather than simply a count of times that instantaneous observed load can be handled – i.e, we want to weight the calculation by the load, not the sample count. This difference is illustrated in Figure 2: if the limit was set at 1 (the 90%ile by time) then the 9/19 units of load in the last time instant would be above the limit (lower dashed line). In this case, the load-adjusted histogram

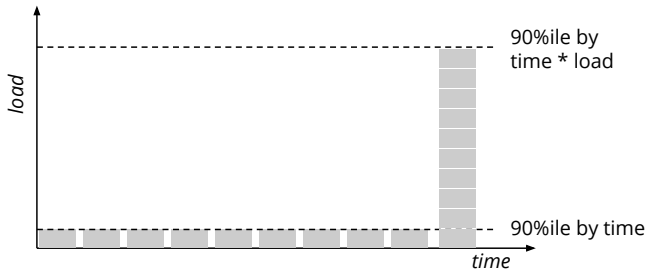


Figure 2. The 90%ile of a load signal can be significantly different than the 90%ile of the integral of the load-time curve. In this example, the 90%ile of the load is 1 unit, using a time-based samples, but 10 units if the magnitude of the load is also considered.

h is computed as follows. A single observation counted by the load-adjusted histogram h can be interpreted as a unit of the signal area processed at a certain load (the current level of the signal). h is equal to $h[1] = 1 \cdot 9$ (load of 1 during 9 time units) and $h[10] = 10 \cdot 10$ (load of 10 during 1 time unit). The 90%ile of h, or the limit required so that 90% of signal area is processed at or below the limit, is therefore 10 – which in this case means the whole signal can be processed within the limit.

Autopilot uses the following statistics based on the signal and the job class. For CPU limits, we use:

- *batch* jobs: S_{avg}, the mean, since if a job tolerates CPU throttling, the most efficient limit for the infrastructure is the job's average load, which allows the job to proceed without accumulating delays.
- *serving* jobs: S_{p95}, the 95%ile, or S_{p90}, the 90%ile of load-adjusted usage, depending on the job's latency sensitivity.

For memory, Autopilot uses different statistics as a function of the job's OOM tolerance. This is set by default to "low" for most large jobs, and "minimal" (the most stringent) for small ones, but can be overridden by the user:

- S_{p98} for jobs with low OOM tolerance.
- $S_{\max}$ for jobs with minimal OOM tolerance.
- For jobs with intermediate OOM tolerance, we select values to (partially) cover short load spikes; we do this by using the maximum of S_{p60}, the weighted 60%ile, and $0.5S_{\max}$, half of the peak usage.

Finally, these raw recommendations are post-processed before being applied. First, a recommendation is increased by a 10–15% safety margin (less for large limits). Then, we take the maximum recommendation value seen over the last hour to reduce fluctuations.

3.2.3 Recommenders based on machine learning. In principle, Autopilot solves a machine learning problem: for a job, based on its past usage, find a limit that optimizes a function expressing both the job's and the infrastructure's goals. The method described in the previous section (setting

a limit based on a simple statistics on a moving window) specifies an algorithm to solve this problem. In contrast, Autopilot's ML recommenders start with a cost function – a specification of the desired solution – and then for each job pick appropriate parameters of a model to optimize this cost function. Such automation allows Autopilot to optimize for each job the parameters that the previous method fixed for all jobs, such as the decay rate $t_{1/2}$, the safety margin or the downscaling stabilization period.

Internally, an ML recommender is composed of a large number of *models*. For each job the recommender periodically chooses the best-performing model (according to the cost function defined below computed over historical usage); then the chosen model is responsible for setting the limits. Each model is a simple arg min-type algorithm minimizing the cost – models differ by weights assigned to individual elements of arg min. One of the recurring problems of ML methods is interpretability of their results [8]: in Autopilot, the limits that the recommender sets must be explainable to the job owner. Having many simple models helps in interpreting ML recommender's actions: a single model roughly corresponds to inferred characteristics of a job (e.g., models with long stabilization times correspond to jobs that have rapidly changing utilizations). Then, given the weights imposed by the chosen model, its decisions are easy to interpret.

More formally, for a signal s, at time t the ML recommender chooses from an *ensemble of models* $\{m\}$ a single model $m[t]$ that is used to recommend the limits. A model is a parameterized arg min algorithm that computes a limit given historical signal values. A model m is parameterized by a decay rate d_m and a safety margin M_m.

At each time instant t, a model tests all possible limit values L (possible limit values correspond to subsequent bounds of the histogram buckets, $L \in \{b[0], \ldots, b[k]\}$). For each limit value L, the model computes the current costs of under- and overruns based on the most recent usage histogram $s[t]$ and then exponentially smooths it with the historic value. The overrun cost $o(L)[t]$ counts the number of samples in buckets over the limit L in the most recent histogram:

$$o(L)[t] = (1-d_m)(o(L)[t-1])+d_m\Big(\sum_{j:b[j]>L} s[t][j]\Big). \quad (4)$$

Similarly, the underrun cost $u(L)[t]$ counts the number of samples in buckets below the limit L,

$$u(L)[t] = (1-d_m)(u(L)[t-1])+d_m\Big(\sum_{j:b[j]<L} s[t][j]\Big). \quad (5)$$

Then, a model picks a limit $L'_m[t]$ that minimizes a weighted sum of overruns, underruns and a penalty $\Delta(L, L'_m[t-1])$ for a possible change of the limit:

$$L'_m[t] = \arg\min_L \Big(w_o o(L)[t]+w_u u(L)[t]+w_{\Delta L}\Delta(L, L'_m[t-1])\Big), \quad (6)$$

where $\Delta(x, y) = 1$ if $x \neq y$ and 0 otherwise. (Using the Kronecker delta, $\Delta(x, y) = 1 - \delta_{x,y}$.) This function captures the three key costs of making resource allocation decisions in a large-scale system. Overruns express the cost of the lost opportunity – in a serving job when an overrun occurs queries get delayed, meaning some end-users might be less willing to continue using the system. Underruns express the cost of the infrastructure: the more resources a job reserves, the more electricity, machines and people are needed. The penalty term Δ helps avoid changing the limits too frequently, because that can result in the task no longer fitting on its current machine causing it (or other tasks) to be evicted.

Finally, the limit is increased by the safety margin M_m, i.e.,

$$L_m[t] = L'_m[t] + M_m. \quad (7)$$

To pick a model at runtime (and therefore to optimize the decay rate d_m and the safety margin M_m for a particular job), the ML recommender maintains for each model its (exponentially smoothed) cost c_m which is a weighted sum of overruns, underruns and penalties for limit changes:

$$c_m[t] = d\Big(w_o o_m(L_m[t], t) + w_u u_m(L_m[t], t)+ w_{\Delta L}\Delta(L_m[t], L_m[t-1])\Big) + (1-d)c_m[t-1]. \quad (8)$$

As historic costs are included in $c_m[t-1]$, the underrun u_m and the overrun o_m costs for a given model consider only the most recent costs, i.e., the number of histogram samples outside the limit in the last sample, thus $o_m(L_m[t], t) = \sum_{j:b[j]>L} s[t][j]$, and $u_m(L_m[t], t) = \sum_{j:b[j]<L} s[t][j]$.

Finally, the recommender picks the model that minimizes this cost, but with additional penalties for switching the limit and the model:

$$L[t] = \arg\min_m \Big(c_m[t]+w_{\Delta m}\Delta(m[t-1], m)+w_{\Delta L}\Delta(L[t], L_m[t])\Big). \quad (9)$$

Overall, the method is similar to the multi-armed bandit problem with an 'arm' of the bandit corresponding to the value of the limit. However, the key property of the multi-armed bandit is that once an arm is chosen, we can't observe the outcomes of all other arms. In contrast, once the signal is known for the next time period, Autopilot can compute the cost function for all possible limit values — except in rare cases when the actuated limit turns out to be too small and a task terminates with an OOM (we show that OOMs are rare in Section 4.3). This full observability makes our problem considerably easier.

The ensemble has five hyperparameters: the weights in the cost functions defined above (d, w_o, w_u, $w_{\Delta L}$ and $w_{\Delta m}$). These weights roughly correspond with the dollar opportunity vs the dollar infrastructure costs. We tune these hyperparameters in off-line experiments during which we simulate Autopilot behavior on a sample of saved traces taken from representative jobs. The goal of such tuning is to produce a configuration that dominates alternative algorithms (such as the moving window recommenders) over a large portion

of the sample, with a similar (or slightly lower) number of overruns and limit adjustments, and significantly higher utilization. Such tuning is iterative and semi-automatic: we perform a parameter sweep (an exhaustive search) over possible values of the weights; and then manually analyze outliers (jobs for which the performance is unusually bad). If we consider that behavior unacceptable, we manually increase the weight of the corresponding jobs when aggregating results during the next iteration of the parameter sweep.

These off-line experiments use raw (unadjusted) usage traces, i.e., they do not try to adjust the signal according to newly set limits (e.g., after an OOM a task should be terminated and then restarted). However, depending on a particular job, the impact of an OOM or CPU throttling might be different – for some jobs, an OOM may increase the future load (as a load of the terminated task is taken over by other tasks), while for others, it might result in a decrease (as end-users fall off when the quality of service degrades). In practice this is not an issue, because usage-adjusting events are fairly rare, and we continually monitor Autopilot in production, where issues like overly-frequent OOMs are easy to spot.

3.3 Horizontal autoscaling

For many jobs, vertical autoscaling alone is insufficient: a single task cannot get larger than the machine it is running on. To address this, Autopilot horizontal scaling dynamically changes the number n of tasks (replicas) in a job as a function of the job's load. The horizontal and vertical scaling mechanisms complement each other: vertical autoscaling determines the optimal resource allocation for an individual task, while horizontal autoscaling adds or removes replicas as the popularity and load on a service changes.

Horizontal autoscaling uses one of the following sources to derive the *raw* recommendation $n_r[t]$ at time instant t:

CPU utilization: The job owner specifies (1) the averaging window for the CPU usage signal (the default is 5 minutes); (2) a horizon length T (the default horizon is 72 hours); (3) statistics S: max or P_{95}, the 95%ile; and (4) the target average utilization r^*. Autopilot computes the number of replicas at time t from the value of S for the most recent T utilization samples, $r_S[t] = S_{\tau \in [t-T,t]}\{\sum_i r_i[\tau]\}$. Then, the raw recommendation for the number of replicas is $n_r[t] = r_S[t]/r^*$.

Target size: the job owner specifies a function f for computing the number of tasks, i.e., $n_r[t] = f[t]$. The function uses data from the job monitoring system. For example, a job using a queuing system for managing requests can scale by the 95%ile of the request handling time; a filesystem server might scale by the amount of filespace it manages.

Horizontal autoscaling requires more customization than the vertical autoscaling, which requires no configuration for a vast majority of jobs. Even in the standard CPU utilization algorithm, the job owner has to at least set the target average utilization r^* (which is similar to how horizontal autoscaling in public clouds is configured). In our infrastructure, horizontal autoscaling is used mainly by large jobs. Their owners usually prefer to tune the autoscaler to optimize job-specific performance metrics, such as the 95%ile latency (either directly through specifying target size; or indirectly, by experimentally changing the target average utilization and observing the impact on the metrics).

The raw recommendation $n_r[t]$ is then post-processed to produce a *stabilized* recommendation $n_s[t]$ which aims to reduce abrupt changes in the number of tasks. Autopilot offers a job owner the following choice of smoothing policies (with reasonable defaults for the average job):

deferred downscaling returns the maximum recommendation from the T_d most recent recommendations: $n_s[t] = \max_{t-T_d,t}\{n_r[t]\}$. Thus, downscaling is deferred for the user-specified time T_d, while upscaling is immediate. Most of our jobs use a long T_d: roughly 40% use 2 days; and 35% use 3 days.

slow decay avoids terminating too many tasks simultaneously. If the current number of tasks $n[t]$ exceeds the stabilized recommendation $n_s[t]$, some tasks will be terminated every 5 minutes. The number of tasks to terminate at a time is chosen to reduce the number of tasks by half over a given period (98% of jobs use the default of one hour).

defer small changes is, to some degree, the opposite of the slow decay: it ignores changes when the difference between the recommendation and the current number of tasks is small.

limiting growth allows the job owner to specify a limit on the fraction of tasks that are initializing (i.e., haven't yet responded to health checks), and thus limit the rate at which tasks are added.

4 Recommender quality

This section explores how effective Autopilot is at Google, using samples from our production workloads. We focus on vertical scaling of RAM here because OOMs have such a directly measurable impact. We refer to [31] for an overview of the impact of CPU scaling.

4.1 Methodology

Our results are based on *observations* made by the monitoring system that monitors all jobs using our infrastructure. The large scale of our operation gives us good statistical estimates of the actual gain of Autopilot, but the downside of any purely observational study is that it doesn't control the treatment a job receives (Autopilot or manually-set limits), so we needed to compensate for this as far as possible.

One alternative to an observational study would be an A/B experiment, in which we would apply Autopilot to a

randomly chosen half of a sample set of jobs. Although we did such A/B studies on small groups of jobs, migrating high-priority, large, production jobs requires explicit consent from the jobs' owners, so was not practical at a statistically-significant scale.

Another alternative would be a simulation study using a recorded trace, but these have their own biases, and we do not have a reliable way to predict how a real job would respond to a simulated event such as CPU-throttling (e.g., end users observing increased latency might disconnect, lowering CPU usage, or reissue their queries, increasing the CPU usage) or an OOM event (e.g., the task might restart and succeed if the problem was a temporary overload, or simply error out again if it was caused by a memory leak).

To mitigate the problems of observability studies, we use results sampled from several different job populations.

The first population is a (biased) sample of 20 000 jobs chosen randomly across our fleet. We sample 5 000 jobs each from the following four categories: jobs with manually-set limits that use hard RAM limits; jobs with manually-set limits that use soft RAM limits; jobs that use the Autopilot moving window recommender; and jobs that use the Autopilot ML recommender. This population gives us fleet-wide measures of the effects of Autopilot, provided we control for a few potential issues:

- Most jobs with manually-set RAM limits use hard limits, whereas Autopilot switches all its jobs to soft RAM limits. This switch might itself reduce the number of OOMs[2]. We mitigate this problem by sampling equal numbers of jobs with hard and soft RAM limits.
- A job might be forced to use manual limits because Autopilot had trouble setting its limits correctly. We address this problem by using a second population which is comprised of a sample of 500 jobs that started to use Autopilot in a particular calendar month. We report their performance during the two calendar months just before and just after the change, to mitigate the risk that binaries or load characteristics might have changed. Even this population could be biased because we can only sample jobs that were successfully migrated, but our success rate is high, so we do not believe this is a significant concern.

4.1.1 Metrics. The performance metrics we report on are based on samples taken over 5-minute aggregation windows (the default for our monitoring system), across calendar days (to align with how we charge for resource allocations), and typically use 95th percentiles, to achieve an appropriate balance between utilization and OOM rates. The metrics are as follows:

footprint for a job during a calendar day is the sum of the average limits of the tasks (each task weighted by its runtime during that day). The footprint directly corresponds to the infrastructure costs of a job: once a task requests resources, other high-priority tasks cannot reclaim them. Footprint is expressed in bytes; however, we normalize it by dividing the raw value in bytes by the amount of memory a single (largish) machine has. So, if a job has a footprint of 10 machines, it means that it is allocated the amount of RAM equal to that of 10 machines (it does not mean it is allocated on 10 machines exclusively dedicated to this job).

relative slack for a job during a calendar day is (*limit* minus *usage*) divided by *limit* – i.e., the fraction of requested resources that are not used. Here, *usage* is the 95%ile of all 5-minute average usage values reported for all of a job's tasks during a calendar day, and *limit* is the average limit over that 24 hour period.

absolute slack for a job during a calendar day (measured in bytes) directly measures waste: it is the sum of *limit-seconds* minus *usage-seconds* over all tasks of a job, divided by 24×3600 (one day). This aggregation puts more emphasis on larger, more costly, jobs. Here, *limit-seconds* is the integral of the requested memory limit across the running time of the tasks, using the 5 minute averages. We normalize absolute slack as we normalize footprint, so if the total absolute slack for an algorithm is 50, we are wasting the amount of RAM equal to that of 50 machines. Achieving a small value of absolute slack is an ambitious target: it requires that all tasks have their limits almost exactly equal to their usage at all times.

relative OOM rate is the number of out-of-memory (OOM) events experienced by a job during a day, divided by the average number of running tasks the job has during that day. It is directly related to how many additional tasks our users need to add to a job to tolerate the additional *unreliability* imposed on it by Autopilot. Since OOMs are rare, we also track the number of job-days that experience no OOMs at all.

The metrics are reported for job-days (i.e., each job will report 30 or 31 such values in a calendar month), and calculate statistics (e.g., the median relative slack) over all the reporting days for all the jobs.

Autopilot may hit a scaling limit when it tries to increase the limit for a job (e.g., the task becomes larger than the available quota, or a user-specified boundary, or even a single machine's size). We did not filter out such OOMs as it is not clear what this job's future behavior would be, and the impact of such events should be independent of the algorithm used.

4.1.2 Job sampling and filtering. We show performance of a sample of jobs executing on our infrastructure in a single calendar month (or 4 months, in case of the analysis of

[2]Memory leaks can be found more quickly with hard limits; an internal user survey told us that most preferred them for non-Autopiloted jobs.

migrated jobs). While our infrastructure runs many types of jobs, its size is driven by high-priority, serving jobs, as such jobs are guaranteed to get the resources they declare as their limits. Thus, tighter limits translate directly to more capacity and a reduced rate of future infrastructure expansion. Our analysis therefore focuses on these jobs.

Unless otherwise noted, we consider only long-running jobs (services that are serving for at least the whole calendar month) as these jobs have the most significant impact on our infrastructure [26]. We also filter out a few job categories with special-purpose, unusual SLOs, and jobs using custom recommenders, to focus the discussion on the quality of the default algorithms.

4.2 Reduction of slack

Autopiloted jobs have significantly lower slack than non-Autopiloted jobs. Figure 3a shows the cumulative distribution function (CDF) of slack per job-days. The average relative slack of a non-Autopiloted job ranges from 60% (hard limits) to 46% (soft limits); while the average relative slack of a Autopiloted job ranges from 31% (moving window) to 23% (ML).

Non-Autopiloted jobs waste significant capacity. Figure 3b shows the cumulative distribution function of absolute slack by jobs within our sample. The total absolute slack summed over our sample of 10 000 non-Autopiloted jobs (and averaged over the month) is equal to more than 12 000 machines; while the absolute slack of the sample of Autopiloted jobs is less than 500 machines. The difference corresponds to tens of millions of USD of machine costs.

These comparisons might be biased, however, as when constructing those samples we controlled the *number* of jobs from each category, not the total amount of resources: if all Autopiloted jobs had small usage, and all non-Autopiloted jobs had large usage, we might end up with a similar savings of absolute slack regardless of the quality of limits in each group (however, the relative slack comparisons are still valid). To address this, we show the CDF of the footprint of jobs in Figure 3c. This plot confirms that Autopiloted jobs do have smaller footprints compared to non-Autopiloted jobs. However, as we will see when analyzing jobs that migrated to Autopilot, this smaller footprint is, at least partially, a consequence of Autopilot reducing jobs' limits. Moreover, small jobs use Autopilot by default (Section 5).

Finally, we analyze the reduction of slack in jobs that recently started to use Autopilot (Figure 4). Almost all jobs used hard memory limits before the migration; and almost all use the ML recommender after the migration. Our plots show results over 4 months of jobs' lifetime. All jobs start to use Autopilot in the same calendar month, denoted as month 0 (m0). We show the performance of these jobs over the two previous months, denoted as m-1 and m-2 when jobs used manual limits; and also performance over two months following the migration, denoted as m+1 and m+2 when jobs used limits set by Autopilot.

Figure 4a shows the CDF for relative slack per job-days. In the month before the migration, the average relative slack was 75%, with a median of 84%. In the month following the migration, the average relative slack decreased to 20% and the median decreased to 17%. The distribution of slack values remains consistent in the two months following the migration, suggesting that the gains are persistent.

The absolute slack (Figure 4b) show significant savings: before the migration, these jobs wasted an amount of RAM equal to the capacity of 1870 machines; after the migration, the jobs waste only 162 machines: by having migrated these jobs, we saved the capacity of 1708 machines.

The CDF of migrated jobs' footprint (Figure 4c) shows that the footprint of jobs increases in time suggesting organic growth in traffic. The total footprint of jobs two months before the migration was smaller than in the month before the migration; similarly, the total footprint in the month after the migration was smaller than the footprint two months after the migration. Migration in m0 reversed this trend: while footprint organically grows month-by-month, the footprint in m+1 was notably smaller than the footprint in m-1. After migration, the rate by which the footprint grows was also reduced, as the CDF of m+2 is closer to m+1 than the CDF of m-2 is to m-1.

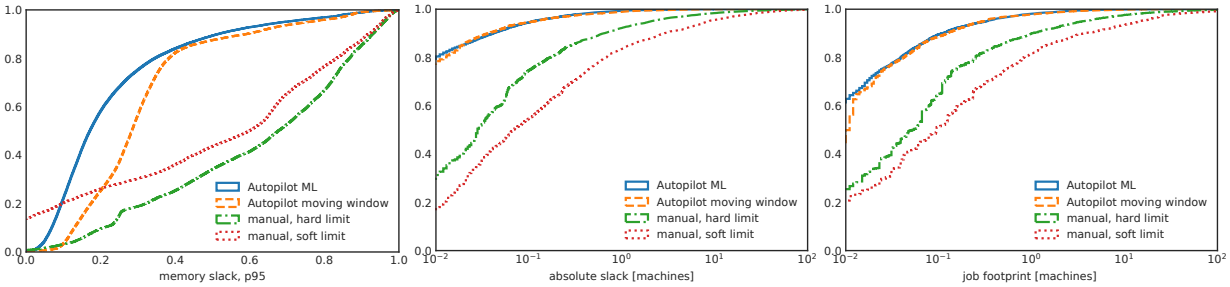
The distribution of footprint of 500 migrated jobs (Figure 4c) differs from the footprint of the 20 000 jobs sampled across our fleet (Figure 3c): the migrated jobs have a larger footprint than the fleet as a whole. This is because many small jobs were automatically migrated earlier than m0, the month we picked as the reference month for this sample (see Section 5 for details).

4.3 Reliability

The previous section demonstrated that Autopilot results in significant reduction of wasted capacity. However, a trivial algorithm, setting the limit to 0, would result in even better results by these metrics – at the expense of frequent OOMs! In this section we show that Autopiloted jobs have higher reliability than the non-Autopiloted ones.

Figure 5 shows the cumulative distribution function (CDF) of relative OOMs by job-days. OOMs are rare: over 99.5% of Autopilot job-days see no OOMs. While the ML recommender results in slightly more OOM-free job-days than the moving window recommender, it also leads to slightly more relative OOMs (0.013 versus 0.002 per task-day). Both algorithms clearly dominate manual limit setting. With hard RAM limits, around 98.7% job-days are OOM-free; and the relative OOM rate is 0.069/task-day. Soft RAM limit jobs had a better relative OOM rate of 0.019, but slightly fewer OOM-free job-days (97.5%).

The number of OOMs naturally depends on the relative slack – higher slack means that more memory is available,

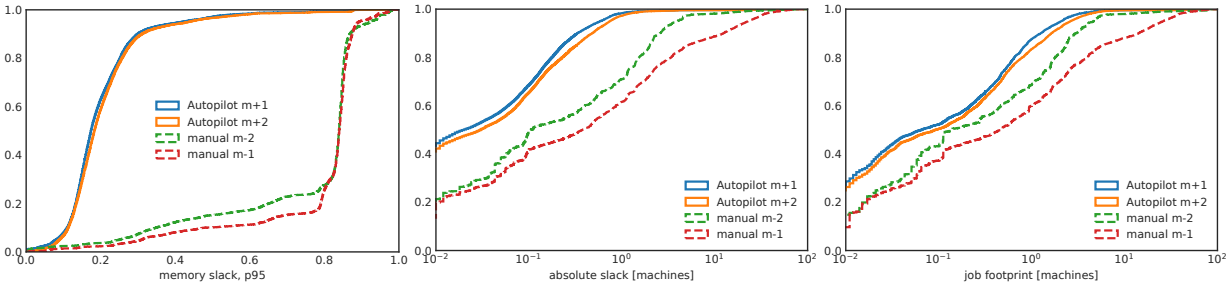


(a) CDF of relative slack (fraction of claimed, but unused memory).

(b) CDF of absolute slack (amount of claimed, but unused memory).

(c) CDF of the footprint (total limit).

Figure 3. Resource usage. CDFs over job-days of a random sample of 5 000 jobs in each category, drawn from across the entire fleet.



(a) CDF of relative slack (fraction of claimed, but unused memory)

(b) CDF of absolute slack (amount of claimed, but unused memory)

(c) CDF of the footprint (total limit)

Figure 4. Resource usage. CDF over job-days. 500 jobs that migrated to Autopilot.

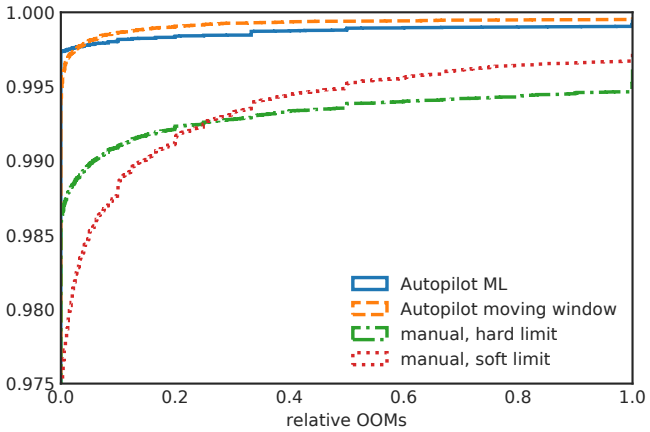


Figure 5. Cumulative distribution function of relative OOMs (number of OOMs per day normalized by the number of tasks) by job-days. Note non-zero y-axis offset – the vast majority of jobs-days have no OOMs, e.g., in the Autopilot cases, over 99.5% of job-days are OOM-free.

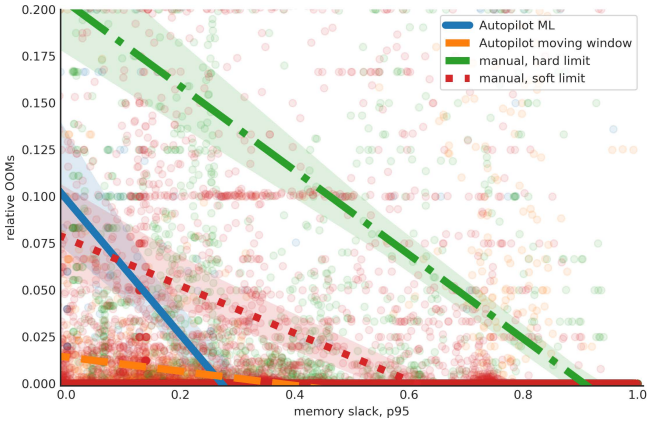


Figure 6. A scatterplot showing OOMs vs slack. A point corresponds to a single job-day; the point's color shows how limit is set for that job on that day. Lines (with a 95%ile confidence interval band) show linear regressions.

so a task should OOM more rarely. The line slopes in Figure 6 represent how strongly the OOM rates relate to slack, while the intercepts reflect the overall number of OOMs. The regression line for non-Autopiloted jobs with soft limits falls below non-Autopiloted jobs with hard limits; there is a similar strict dominance between Autopiloted jobs that use the moving window algorithm and the non-Autopiloted jobs using soft limits. However, the regression for the ML algorithm intersects the lines for jobs using a manually specified soft limit and those using the moving window Autopilot scheme, suggesting more OOMs for jobs with low slack – but also fewer OOMs for jobs with higher slack.

Table 2. Number of job-days seriously impacted by OOMs, by algorithm. Each row corresponds to a different threshold for classifying a job-day as being seriously impacted by OOMs: e.g., in the first row, a job-day is seriously impacted if 5, or 1/7th of the job's tasks (whichever is higher) are terminated with an OOM. Both "hard" and "soft" have manual limit settings.

thresholds		affected job-days			
# OOMs	fraction tasks	hard	soft	window	ML
5	1/7	807	994	131	98
4	1/5	813	945	120	108
4	1/7	862	1059	142	109
4	1/10	916	1235	152	113
3	1/7	931	1116	149	118

Because the ML recommender results in higher average relative OOMs rates than the sliding window recommender, it may be that the ML recommender reduces jobs' limits too aggressively. However, the ML recommender is designed to result in an occasional OOM for jobs that won't be impacted too much. As we explained in Section 2, our jobs are designed to absorb occasional failures – as long as there are sufficiently many surviving tasks able to absorb the traffic that the terminated task once handled. This is working as intended, and the benefit is bigger resource savings. Yet one might reasonably still be concerned that the recommender is being *too* aggressive.

To explore this concern, we categorized a job-day as being *seriously impacted by OOMs* when it experiences more OOMs during the day than the larger of a threshold number (e.g., 4) or fraction (e.g., 1/7) of its tasks. Table 2 shows the number of job-days that were seriously impacted by OOMs across some threshold settings ranging from more liberal (top) to more conservatitive (bottom). Although the absolute numbers slightly vary, the relative ordering of the methods stays the same.

Among non-Autopiloted jobs, we were surprised to find that the jobs with hard RAM limits, while having more relative OOMs (as discussed above), were less seriously affected by OOMs. We hypothesize that users may be manually specifying soft limits for jobs with an erratic memory usage pattern that are particularly difficult to provision for. Among Autopiloted jobs, as expected, while jobs using the moving window algorithm have less relative OOMs, they are somewhat more likely to be seriously affected by OOMs than jobs using the ML algorithm.

We also studied in more detail how concentrated OOMs are across jobs. If most OOMs occur in just a few jobs, this might point to a systematic problem with the autoscaling algorithm — we would rather have many jobs experiencing infrequent OOMs. We analyze jobs that had at least one OOM during the month. For each such job, we count the days with at least one OOM (we count OOM-days, rather than simply OOMs or relative OOMs, to focus on the repeatability, rather than magnitude, which we measured above). For Autopilot ML, 46% of such jobs OOM exactly once; and 80% of jobs OOM during 4 days or less. In contrast, in Autopilot moving window recommender, only 28% of jobs OOM exactly once; and 80% of jobs OOM during 21 days or less.

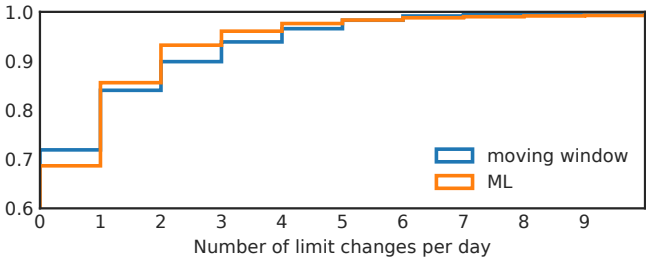


Figure 7. CDF of the number of limit changes per job-day. Note non-zero y-axis offset.

In our second sample population (jobs that migrated to Autopilot), the number of OOMs is too small for meaningful estimates of relative OOMs and serious OOMs. In the month before the migration, there were in total 348 job-days in which there is at least one OOM; after the migration, this number was reduced to just 48. Migration was successful for these jobs.

4.4 Number of limit changes

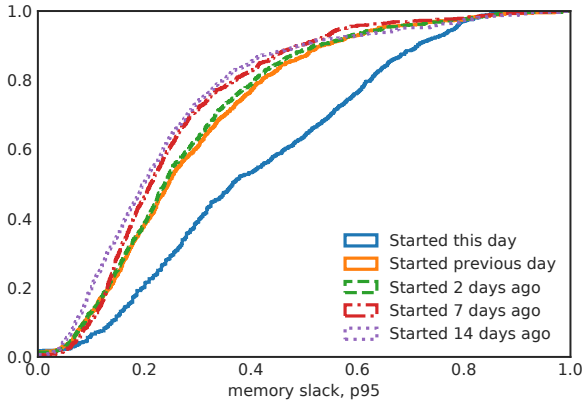
Manually-controlled jobs rarely have their limits changed: on our sample of 10 000 manually-limited jobs, we observed 334 changes during one month, or about 0.001 changes per job-day. Figure 7 shows how often Autopilot changes limits on our 10 000 job sample: a few hundred times more often per job than users do. However, it is still quite stable: in roughly 70% of job-days there are no changes; and the 99%ile job-day has only 6 (moving window) to 7 (ML) limit changes during a day. Given that it typically only takes a few tens of seconds to find a new place for a task even if it is evicted, this seems a reasonable price to pay for significant savings.

One might argue that the reduction of OOMs (and serious OOMs) reported in the previous section comes just from changing the limits more often than a human operator. In Figure 7, Autopilot ML and moving window have a similar number of limit changes; yet, as Table 2 shows, Autopilot ML uses this disruption budget more efficiently.

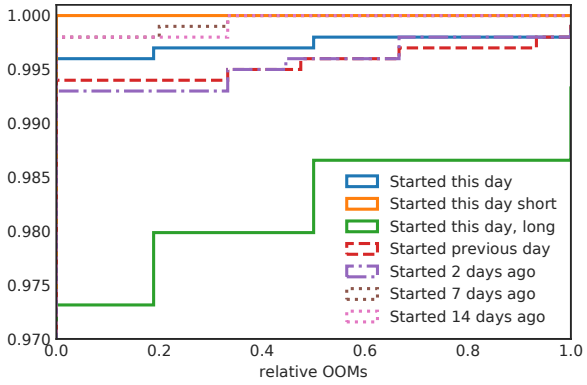
4.5 Behavior in time

In previous sections, we focused on long-running jobs: our jobs were serving continuously for at least a month. In this section, we analyze Autopilot performance as a function of the jobs' age. Figure 8a shows CDFs of the relative slack for 1 000 jobs of each different age range.

Jobs that have been running for less than a day have a significantly higher slack than jobs that have been running longer: this is a direct result of Autopilot being tuned for long-running jobs – the more history is available, the lower



(a) CDF of memory slack for different jobs ages.



(b) CDF of relative OOMs (number of OOMs per day normalized by number of tasks) for different job ages. Note non-zero y-axis offset.

Figure 8. Performance over jobs with short history. Each different age range has 1 000 jobs sampled from our whole fleet.

the slack. But even after 14 days the slack is still higher than for the steady state analyzed in the previous section.

The analysis of relative OOM rates (Figure 8b) shows that Autopilot is cautious with short jobs. For jobs with less than 24h duration, there are almost no OOMs: however, if we filter out short jobs (ones with total task duration of less than 1.5 hours), there are more OOMs than in the steady state. Once we consider jobs that started 7 or more days ago, the relative OOM rates are comparable to the steady state behavior.

5 Winning the users' trust: key features for increasing adoption

Our infrastructure serves thousands of internal users who have varied roles, experience and expectations. Smaller or newer services are typically run in production by software engineers who originally created them, while larger, more established services have dedicated dev/ops teams. To increase Autopilot's adoption we had to not only make sure the quality of our algorithms was acceptable, but also identify and answer needs our engineers have from the infrastructure. This section discusses these qualitative aspects. Our experience reinforces many of the lessons described in [5].

5.1 Evaluation process

Along with Autopilot, we developed a process to evaluate potential recommenders. A recommender is first evaluated in off-line simulations, using traces of resource usage of a representative sample of jobs. While such evaluation is far from complete (we detailed problems in Section 4.1), it is good enough to determine whether it is probably worth investing more effort in a recommender. If so, we proceed to using dry runs, in which the recommender runs as part of the production Autopilot service along side other recommenders – its recommendations are logged, but not acted upon. In both phases, we analyze the usual statistical aggregations such as means and high percentiles, but also pay particular attention to *outliers* – jobs on which the recommender performed particularly badly. These outliers have helped catch both bugs in implementation and unintended consequences of our algorithms.

Afterwards, we perform A/B tests in which the new recommender drives limits in production for a small fraction of users within a chosen cluster. Even a complete algorithm failure in this phase is unlikely to be catastrophic: if a job fails in one cluster, the service's load balancer will switch its traffic to other clusters, which should have enough capacity to handle the surge.

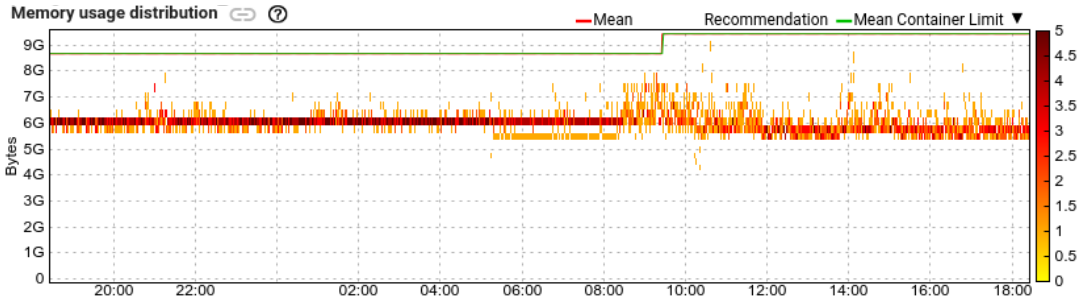
Finally, when the new recommender compares favorably in A/B tests, we gradually deploy it as a new standard for the entire fleet. To reduce the risk of possible failures, roll-outs are performed cluster by cluster, with multi-hour gaps between them, and can be rolled back if anomalies are detected.

5.2 Autopilot limits easily accessible to job owners

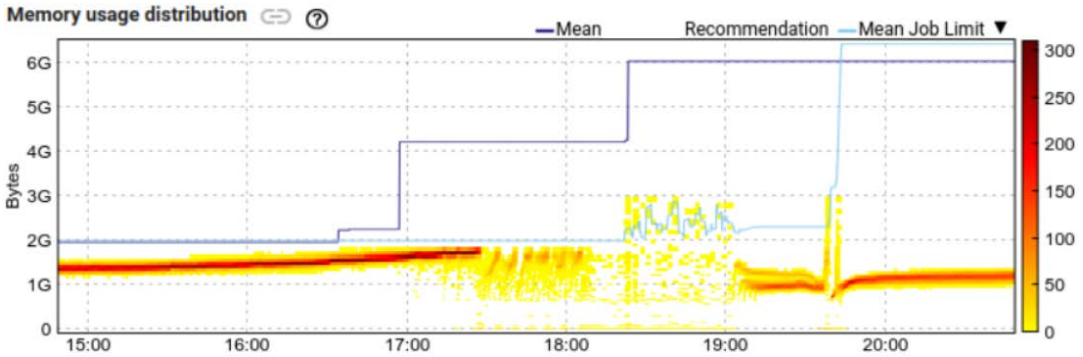
Our standard dashboard (Figure 9) for resource monitoring displays the distribution of job CPU and memory usage as well as the limits Autopilot computed – even for jobs that are not Autopiloted (for these jobs Autopilot runs in simulation mode). The dashboard helps the user to understand Autopilot's actions, and build trust in what Autopilot would do if it was enabled on non-Autopiloted jobs: the user can see how Autopilot would respond to daily and weekly cycles, new versions of binaries or suddenly changing loads.

5.3 Automatic migration

Once we had sufficient trust in Autopilot's actions from large-scale off-line simulation studies and smaller-scale A/B experiments, we enabled it as a default for all existing small jobs (with an aggregate limit of up to roughly 10 machines) and all new jobs. Users were notified well in advance and they were able to opt-out. This automatic migration trivially increased adoption with practically no user backlash.



(a) A trace of an Autopiloted job. Just after 8:00, some tasks started to exhibit an increased level of memory usage; Autopilot promptly increased the job's limit.



(b) A trace of a job using manually specified limits and with Autopilot running in simulation mode. A new roll-out had a memory leak that caused the memory usage to steadily increase. At about 17:30, the job's monitoring system started to page on-calls as the job started to OOM. It was not until 19:30 that the on-call dev/ops engineer managed to increase the job's memory limit (limit oscillations between 18:20 and 19:00 are an artifact of misalignment of time series data from the monitoring system). If the job had been Autopiloted, it would probably not have OOMed as the automatically increased limits would have given dev/ops more time to discover the memory leak and roll back to the previous version of the binary.

Figure 9. Screenshots of our monitoring dashboards for two production jobs. The dashboard includes a color-coded heatmap which indicates the number of a job's tasks with a certain usage at a particular time moment.

5.4 Overriding recommenders with custom recommenders

Autopilot's algorithms rely on historical CPU/memory usage to set future limits or task counts. However, for some jobs other signals are better predictors of the limits: for instance, the load of our filesystem servers depends almost linearly on the total size of the files controlled by the server. Additionally, some long-running services had already developed their own horizontal autoscalers before Autopilot, some of which contained sophisticated, finely-tuned logic that had been refined over many years (although they often had only a subset of Autopilot's features, and they did not always keep up with changes in Borg). Autopilot's custom recommenders permit users to preserve the critical parts of such algorithms – computation of the number of tasks or individual task resource limits – while delegating support functions such as actuation to the Autopilot ecosystem.

Custom recommenders proved popular: 3 months after they were made available, custom recommenders were managing 13% of our entire fleet's resources.

6 Reducing engineering toil

Google follows the dev/ops principle of reducing toil: tedious, repetitive work should be performed by machines rather than engineers and so we invest in automation to do so. Autopilot is one example.

A job's limits need to be increased as the job's workload increases. Popular services, even excluding the initial rapid growth phase, probably should be resized bi-weekly or monthly. And each roll-out of a new binary version may require limit adjustments. Suppose these were done manually. We assume that a manual resize requires on average 30 minutes of work: to change the configuration file, submit the change to the version control system, review the proposed change, initialize the roll-out and monitor its results. For our sample of 10 000 jobs with manual limits, even the 334 manual limit adjustments represent a total toil of roughly a person-month – and this is significantly less than the expected number of updates.

Autopilot's horizontal scaling – adding tasks to running jobs – automatically handles organic load growth. Autopilot's vertical scaling can handle both per-task load changes

and the effects of rolling out of a new binary. Both represent significant toil reduction.

We asked several owners of large jobs who migrated to Autopilot to estimate the reduction in toil they had experienced. One large service (composed of multiple jobs) reported that before migrating to Autopilot they performed roughly 8 manual resizes monthly. Another service estimated that Autopilot saves them 2 hours of human work per month previously needed for manual resizes. Another service, for which load varied significantly between clusters and in time, needed about 12 manual resizes per month.

Another benefit is reducing the interrupts (pages) that must be handled by on-call dev/ops engineers. With increased reliability, tasks fail less often, and the reporting system issues fewer alarms. This reduction is especially pronounced for jobs with loads that vary significantly across clusters: the toil of setting different manual limits is a frequent source of problems, and even complicates monitoring. One service reported that after migration, Autopilot *increased* memory limits in some clusters, which resulted in reducing the number of OOMs from roughly 2 000 a day to a negligible number. Another service reported no OOMs for almost a year following the migration; the number of on-call pages was reduced from 3 per week to less than 1 (the remaining pages were for unrelated problems).

Autopilot's tighter resource limits may expose bugs in jobs that went unnoticed with larger limits. Rare memory leaks or out-of-bounds accesses are notoriously difficult to find. While Autopilot works well in most cases, it may still require customized configuration for a few jobs. Thus, when a job is migrated to Autopilot and then starts to OOM frequently, it can be difficult to distinguish between Autopilot mis-configurations and a true bug. One group blamed Autopilot's memory limit setting algorithm for such a problem, and only discovered the root cause a few weeks later: a rarely-triggered out-of-bounds memory write.

And finally, Autopilot is used heavily by batch jobs (88% of such jobs by CPU enable it). We surmise that this is because Autopilot eliminates the need for a user to even specify limits for such jobs.

7 Related work

While Autopilot actuation, UX and some customisations are specific to Borg, the problem Autopilot solves is almost universal in cloud resource management.

The desired number of replicas and their resource requirements are expected to be provided by users in many cloud resource management systems, as the scheduler uses them to pack tasks to machines [14]. Borg [34], Omega [29] and Kubernetes [3] all require their users to set such limits when submitting jobs (Borg, Omega) or pods (Kubernetes). YARN [32] requires applications (jobs) to state the number of containers (tasks) and the CPU and RAM resource requirements of each container. In a somewhat different context, schedulers for HPC systems, such as Slurm [37], require each batch job to specify the number of machines.

Other studies confirm low utilization in private clouds that we observed in our infrastructure when jobs have their limits manually set. [30] analyzes 5-day usage of a 10 000-machine YARN cluster at Alibaba, and reports that 80% of time, RAM utilization is less than 55%. [23] analyzes a short (12-hour) Alibaba trace, showing that for almost all instances (tasks), the peak memory utilization is 80% or less. [26], analyzing the 30-days Google cluster trace [27], shows that although the mean requested memory is almost equal to the total available memory, the actual *usage* (averaged over one-hour windows) is below 50%.

An alternative to setting more precise limits is to oversubscribe resources, i.e., to deliberately assign to a machine tasks such that the sum of their requirements is higher than the amount of physical resources available locally. [30] shows a system oversubscribing resources in a YARN cluster. While oversubscription can be used in batch workloads that can tolerate occasional slowdowns, it may lead to significant increases in tail latency for serving workloads – which requires a careful, probabilistic treatment [2, 21].

Horizontal and vertical autoscaling requires the job to be elastic. In general, many classes of applications are notoriously difficult to scale. For example, JVMs, in their default configuration, are reluctant to release heap memory. Fortunately for Autopilot, the vast majority of jobs at Google were built with scaling in mind.

Autoscaling is a well-developed research area; recent surveys include [11, 16, 22]. The majority of research addresses horizontal autoscaling. [17] experimentally analyzes the performance of a few horizontal autoscaling algorithms for workflows. [10] builds probabilistic performance models of horizontal autoscalers in AWS and Azure. [24] measures the performance of horizontal autoscalers in AWS, Azure and GCE. While Autopilot also has a reactive horizontal autoscaler, this paper largely concentrates on vertical scaling (also called rightsizing, or VM adaptation in [16]). Kubernetes vertical pod autoscaler (VPA, [15]) sets containers' limits using statistics over a moving window (e.g., for RAM, the 99th percentile over 24h). Kubernetes' approach was directly inspired by our moving window recommenders (Section 3.2.2). [25] proposes an estimator which uses the sum of the median and the standard deviation over samples from a window.

We described two recommenders: one based on statistics computed from a moving window, with window parameters, such as length, set by the job owner (Section 3.2.2); and another one that automatically picks the moving window parameters based on a cost function (Section 3.2.3). An alternative to these simple statistics would be to use more advanced time series forecasting methods, such as autoregressive moving average (ARMA) (as in [28] that also uses a

job performance model), neural networks (as in [18]), recurrent neural networks as in [9, 20] or a custom forecast as in [13] which demonstrates that Markov-chain based prediction performs better than methods based on autoregression or autocorrelation. Our initial experiments with such methods demonstrated that, for the vast majority of Borg cases, the additional complexity of ARMA is not needed: jobs tend to use long windows (e.g., the default half life time for memory in moving window recommenders is 48 hours, Section 3.2.2); and the between-day trends are small enough that a simple moving window reacts quickly enough. For a recommender that is configurable by the user we believe that it is more important that parameters have simple semantics and that the recommender can be tuned predictably.

Autopilot refrains from building a job performance model: it does not try to optimize batch jobs' completion time or serving jobs' end-user response latency. We found that controlling the limits enables job owners to reason about job performance (and performance problems) and to separate concerns between the job and the infrastructure. This separation of concerns partly relies on the cluster and node-level schedulers. For example, Autopilot does not need to consider performance problems from so-called noisy neighbors as Borg handles them through a special mechanism [38]. To cover the few remaining special cases, Autopilot provides horizontal scaling hooks to allow teams to use custom metrics or even custom recommenders. In contrast, many research studies aim to directly optimize the job's performance metrics, rather than just the amount of allocated resources. For example, in Quasar [7] users specify performance constraints, not limits, and the scheduler is responsible for meeting them. In public clouds, in which jobs are assigned to VMs with strict limits, Paris [36] recommends a VM configuration given a representative task and its performance metrics. D2C [12] is a horizontal autoscaler that scales the number of replicas in each layer of a multi-tier application by learning the performance parameters of each layer (modeled with queuing theory). The learning process is on-line – the application does not have to be benchmarked in advance.

Even more specific optimizations are possible if the category of jobs is more narrow. Ernest [33] focuses on batch, machine learning jobs, while CherryPick [1] also considers analytical jobs. While this paper concentrates on serving jobs, Autopilot is also used by 88% of batch jobs at Google (measured by CPU consumption). [19] uses reinforcement learning (RL) to drive horizontal scaling of serving jobs (with a utility function taking into account the number of replicas, the throughput and any response time SLO violations). Autopilot's ML recommender borrows some ideas from RL, such as choosing a model and a limit based on its historical performance.

8 Conclusions

Autoscaling is crucial for cloud efficiency, reliability, and toil reduction. Manually-set limits not only waste resources (the average limits are too high) but also lead to frequent limit violations as load increases, or when new versions of services are rolled out.

Autopilot is a vertical and horizontal autoscaler used at Google. By automatically increasing the precision of limit settings, it has reduced resource waste and increased reliability: out-of-memory errors are less common, less severe and affect fewer jobs. Some of these gains were reachable with a simple time-weighted sliding window algorithm, but switching to a more sophisticated algorithm inspired by reinforcement learning enabled significant further gains.

Autopiloted jobs now represent over 48% of our fleet-wide usage. Achieving such a high adoption rate took significant development and trust-building measures to achieve, even in the presence of evident gains in reliability and efficiency. However, we demonstrated that this extra work has paid off, as users no longer have to resize their jobs manually, and the improved reliability results in fewer on-call alerts. We strongly commend this approach to other users of large scale compute clusters.

Acknowledgments

The authors of this paper performed measurements and wrote the paper, but many other people were critical to Autopilot's success. Those who contributed directly to the project include: Ben Appleton, Filip Balejko, Joachim Bartosik, Arek Betkier, Jason Choy, Krzysztof Chrobak, Sławomir Chyłek, Krzysztof Czaiński, Marcin Gawlik, Andrea Gesmundo, David Greenaway, Filip Grządkowski, Krzysztof Grygiel, Michał Jabczyński, Sebastian Kaliszewski, Paulina Kania, Tomek Kulczyński, Sergey Melnychuk, Dmitri Nikulin, Rafal Pytko, Natalia Sakowska, Piotr Skowron, Paweł Stradomski, David Symonds, Jacek Szmigiel, Michał Szostek, Lee Walsh, Peter Ward, Jamie Wilkinson, Przemysław Witek, Lijie Wong, Janek Wróbel, Tomasz Zielonka.

References

[1] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 469–482, 2017.

[2] D. Breitgand and A. Epstein. Improving consolidation of virtual machines with risk-aware bandwidth oversubscription in compute clouds. In *IEEE INFOCOM*, pages 2861–2865. IEEE, 2012.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes. *Queue*, 14(1):10, 2016.

[4] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. Long-term SLOs for reclaimed cloud computing resources. In *ACM Symposium on Cloud Computing (SoCC'14)*, pages 1–13. ACM, 2014.

[5] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, et al. Hydra: a federated resource manager for data-center scale analytics. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*,

pages 177–192, 2019.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *6th Conference on Symposium on Operating Systems Design & Implementation (OSDI'04)*, pages 10–10, San Francisco, CA, 2004. USENIX Association.

[7] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[8] F. Doshi-Velez and B. Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[9] M. Duggan, R. Shaw, J. Duggan, E. Howley, and E. Barrett. A multitime-steps-ahead prediction approach for scheduling live migration in cloud data centers. *Software: Practice and Experience*, 49(4):617–639, 2019.

[10] A. Evangelidis, D. Parker, and R. Bahsoon. Performance modelling and verification of cloud-based auto-scaling policies. *Future Generation Computer Systems*, 87:629–638, 2018.

[11] G. Galante, L. C. E. De Bona, A. R. Mury, B. Schulze, and R. da Rosa Righi. An analysis of public clouds elasticity in the execution of scientific applications: a survey. *Journal of Grid Computing*, 14(2):193–216, 2016.

[12] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, model-driven autoscaling for cloud applications. In *11th International Conference on Autonomic Computing (ICAC'14)*, pages 57–64, 2014.

[13] Z. Gong, X. Gu, and J. Wilkes. Press: predictive elastic resource scaling for cloud systems. In *International Conference on Network and Service Management*, pages 9–16. IEEE, 2010.

[14] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM'14*, pages 455–466. ACM, 2014.

[15] K. Grygiel and M. Wielgus. Kubernetes vertical pod autoscaler: design proposal. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md, kubernetes community, 2018. Accessed 2019-11-04.

[16] A. R. Hummaida, N. W. Paton, and R. Sakellariou. Adaptation in cloud resource configuration: a survey. *Journal of Cloud Computing*, 5(1):7, 2016.

[17] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup. An experimental performance evaluation of autoscaling policies for complex workflows. In *8th ACM/SPEC on International Conference on Performance Engineering*, pages 75–86. ACM, 2017.

[18] S. Islam, J. Keung, K. Lee, and A. Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2012.

[19] P. Jamshidi, A. M. Sharifloo, C. Pahl, A. Metzger, and G. Estrada. Self-learning cloud controllers: fuzzy q-learning for knowledge evolution. In *International Conference on Cloud and Autonomic Computing*, pages 208–211. IEEE, 2015.

[20] D. Janardhanan and E. Barrett. CPU workload forecasting of machines in data centers using LSTM recurrent neural networks and ARIMA models. In *12th International Conference for Internet Technology and Secured Transactions (ICITST'17)*, pages 55–60. IEEE, 2017.

[21] P. Janus and K. Rzadca. SLO-aware colocation of data center tasks based on instantaneous processor requirements. In *ACM Symposium on Cloud Computing (SoCC'17)*, pages 256–268. ACM, 2017.

[22] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

[23] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai. Imbalance in the cloud: an analysis on Alibaba cluster trace. In *IEEE International Conference on Big Data (BigData'17)*, pages 2884–2892. IEEE, 2017.

[24] V. Podolskiy, A. Jindal, and M. Gerndt. IaaS reactive autoscaling performance challenges. In *11th IEEE International Conference on Cloud Computing (CLOUD'18)*, pages 954–957. IEEE, 2018.

[25] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes. In *12th IEEE International Conference on Cloud Computing (CLOUD'19)*, pages 33–40. IEEE, 2019.

[26] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *3rd ACM Symposium on Cloud Computing (SoCC'12)*, page 7. ACM, 2012.

[27] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. Technical report at https://github.com/google/cluster-data/, Google, Mountain View, CA, USA, 2011.

[28] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *4th IEEE International Conference on Cloud Computing (CLOUD'11)*, pages 500–507. IEEE, 2011.

[29] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *8th ACM European Conference on Computer Systems (EuroSys'13)*, pages 351–364. ACM, 2013.

[30] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, and C. Li. ROSE: cluster resource scheduling via speculative over-subscription. In *38th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 949–960. IEEE, 2018.

[31] M. Tirmazi, A. Barker, N. Deng, Z. G. Qin, M. E. Haque, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: the Next Generation. In *15th ACM European Conference on Computer Systems (EuroSys'20)*, Heraklion, Crete, Greece, 2020.

[32] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *4th ACM Symposium on Cloud Computing (SoCC'13)*, page 5. ACM, 2013.

[33] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 363–378, 2016.

[34] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *10th ACM European Conference on Computer Systems (EuroSys'15)*, page 18. ACM, 2015.

[35] J. Wilkes. Google cluster usage traces v3. Technical report at https://github.com/google/cluster-data, Google, Mountain View, CA, USA, 2020.

[36] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the best VM across multiple public clouds: A data-driven performance modeling approach. In *ACM Symposium on Cloud Computing (SoCC'17)*, pages 452–465, 2017.

[37] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: simple Linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[38] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *8th ACM European Conference on Computer Systems (EuroSys'13)*, pages 379–391, 2013.