SJTU SAIL (Software Architecture and Infrastructure Lab)

SHANGHAI JIAO TONG UNIVERSITY

Resource Management

Yuchen Cheng and Yalun Lin Hsu

2020-09-21

Outline

- Yuchen Cheng
  - Large-scale cluster management at Google with Borg
  - Autopilot: workload autoscaling at Google
- Yalun Lin Hsu
  - Medea: scheduling of long running applications in shared production clusters
  - Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications

Background

- Growing number of data-intensive scientific applications
  - New cluster computing frameworks will continue to emerge
  - No framework will be optimal for all applications

Target

- **Run frameworks in the same cluster**
- Solutions
  - Statically partition the cluster and run one framework per partition
  - Allocate a set of VMs to each framework
- **Problem**
  - The mismatch between the allocation granularities of these solutions and of existing frameworks

Goals

- High cluster utilization
- Low latency
- Strict enforcement of scheduling invariants
- Diverse application framework support
- Fairness and fine-grained
- Fault tolerant
- Highly scalable

Scheduler architectures

- Monolithic scheduling
  - Borg
  - Kubernetes
- Two-level scheduling
  - Mesos
  - YARN
- Shared-state scheduling
  - Omega
  - Apollo
- Distributed scheduling
  - Sparrow
- Hybrid scheduling
  - Hawk

Monolithic scheduling

- All jobs that are to be executed must pass through a single scheduler
- **Pros**
  - Accurate view of the state of a cluster
  - Good scheduling decision
- **Cons**
  - Complexity
  - Not clear for future framework requirements
  - Expensive refactoring for existing frameworks

Two-level scheduling

- Separate the concerns of resource allocation and task placement
- Mesos: Resource offer
  - Mesos decides how many resources to offer each framework
  - Frameworks decide which resources to accept and which tasks to run on them
- **Pros**
  - Flexibility
  - Parallelism
- **Cons**
  - Limited parallelism due to pessimistic concurrency control
  - Restricted visibility of resources in a scheduler framework

Shared-state scheduling

- Allow different schedulers to update the state of the cluster
  - Lock-free optimistic concurrency control
- **Pros**
  - Have access to all cluster-wide resources
- **Cons**
  - Must work with outdated information
  - May experience degraded scheduler performance under high contention

Distributed scheduling

- No coordination between the schedulers
- Each scheduler use its own local copy of the state of the cluster
- Worker machines run tasks in a fixed number of slots
- **Pros**
  - Suitable for time-sensitive jobs
- **Cons**
  - Insufficient for long running jobs

Hybrid scheduling

- Combination of centralized and distributed scheduling
  - A distributed one for a workload
  - A centralized one for the rest

Large-scale cluster management at Google with Borg

Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes

Benefits

- Borg
    1. hides the details of resource management and failure handling so its users can focus on application development instead
    2. operates with very high reliability and availability, and supports applications that do the same
    3. lets us run workloads across tens of thousands of machines effectively

How does Borg work?

- Users submit jobs
  - Each job contains 1+ task that all run the same program/binary
  - Runs inside containers (not VMs as it would cost higher latency)
- Each job runs on one cell
  - A cell is a set of machines that run as one unit
- Two main types of jobs:
  - Prod job : long-running server jobs, higher priority
  - Non-prod job : quick batch jobs, lower priority

How does Borg work?

- Users submit jobs
  - Each job contains 1+ task that all run the same program/binary
  - Runs inside containers (not VMs as it would cost higher latency)
- Each job runs on one cell
  - A cell is a set of machines that run as one unit
- Two main types of jobs:
  - Prod job : long-running server jobs, higher priority
  - Non-prod job : quick batch jobs, lower priority

Stanford CS349D: Cloud Computing Technology. https://docs.google.com/presentation/d/1HHgMZ6WpRUj6RC_QCAPVQRPEwcXcRec53xAR5__JC6U

Borg architecture

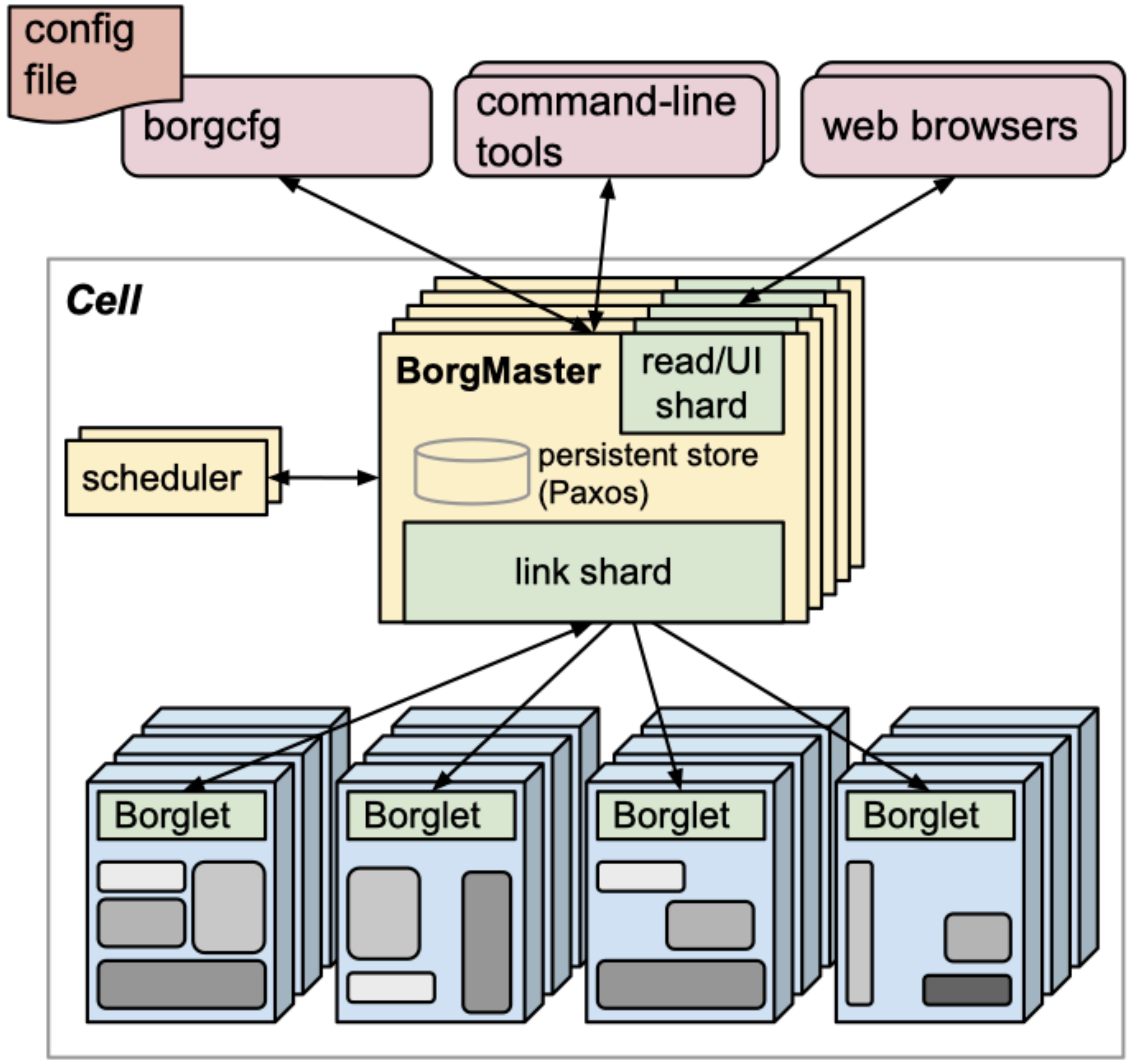


Figure 1: The high-level architecture of Borg. *Only a tiny fraction of the thousands of worker nodes are shown.*

Borgmaster

- Each cell contains a Borgmaster
- Each Borgmaster consists of 2 processes:
  - Main Borgmaster process
  - Scheduler
- Multiple replicas of each Borgmaster
- Role of (elected leader) Borgmaster:
  - submission of job, termination of any of job’s task

Scheduling

- Borgmaster adds new jobs to a pending queue after recording it in the Paxos store
- A scheduler (primarily operates on tasks) scans and assigns tasks to machines
  - Feasibility checking
  - Scoring
- E-PVM vs “best-fit”
  - E-PVM leaves headroom for load-spikes but has increased fragmentation
  - Best-fit fills machines as tightly as possible, but hurts “bursty loads”
- Current model is a hybrid of both
  - Borg will kill lower priority tasks until it finds room for an assigned task

Borglet

- Local Borg agent on every cell
  - starts/stops/restarts tasks
  - Manages local resources
  - Rolls over debug logs
- Polled by Borgmaster to get machine’s current state
- If a Borglet does not respond to several poll mesages, it is marked as down
  - Tasks re-distributed
  - If communication is restored, Borgmaster tells Borglet to kill rescheduled tasks

Scalability

- Ultimate scalability limit is unknown
  - Single Borgmaster can manage thousands of borglets
  - Rates above 10,000 tasks per minute
  - Busy Borgmaster uses 10-14 CPU cores and 50GiB RAM

Scalability

- **Score caching**
  - Feasibility and scoring is expensive, scores are cached until properties of the machine or task change
- **Equivalence class**
  - A group of tasks with identical requirements
  - Stems from tasks within a job having identical requirements and constraints
  - Easier than determining feasibility for every pending task and scoring every machine
- **Relaxed randomization**
  - Scheduler examines machines in a random order until it has found enough feasible machines to score, and selects the best from this set
  - Speeds up assignments of tasks to machines

Stanford CS349D: Cloud Computing Technology. https://docs.google.com/presentation/d/1HHgMZ6WpRUj6RC_QCAPVQRPEwcXcRec53xAR5__JC6U

Kubernetes Scheduler

Node selection

1. Filtering
   - Find the set of Nodes where it's feasible to schedule the Pod
   - Predicates
2. Scoring
   - Rank the remaining nodes to choose the most suitable Pod placement
   - Priorities

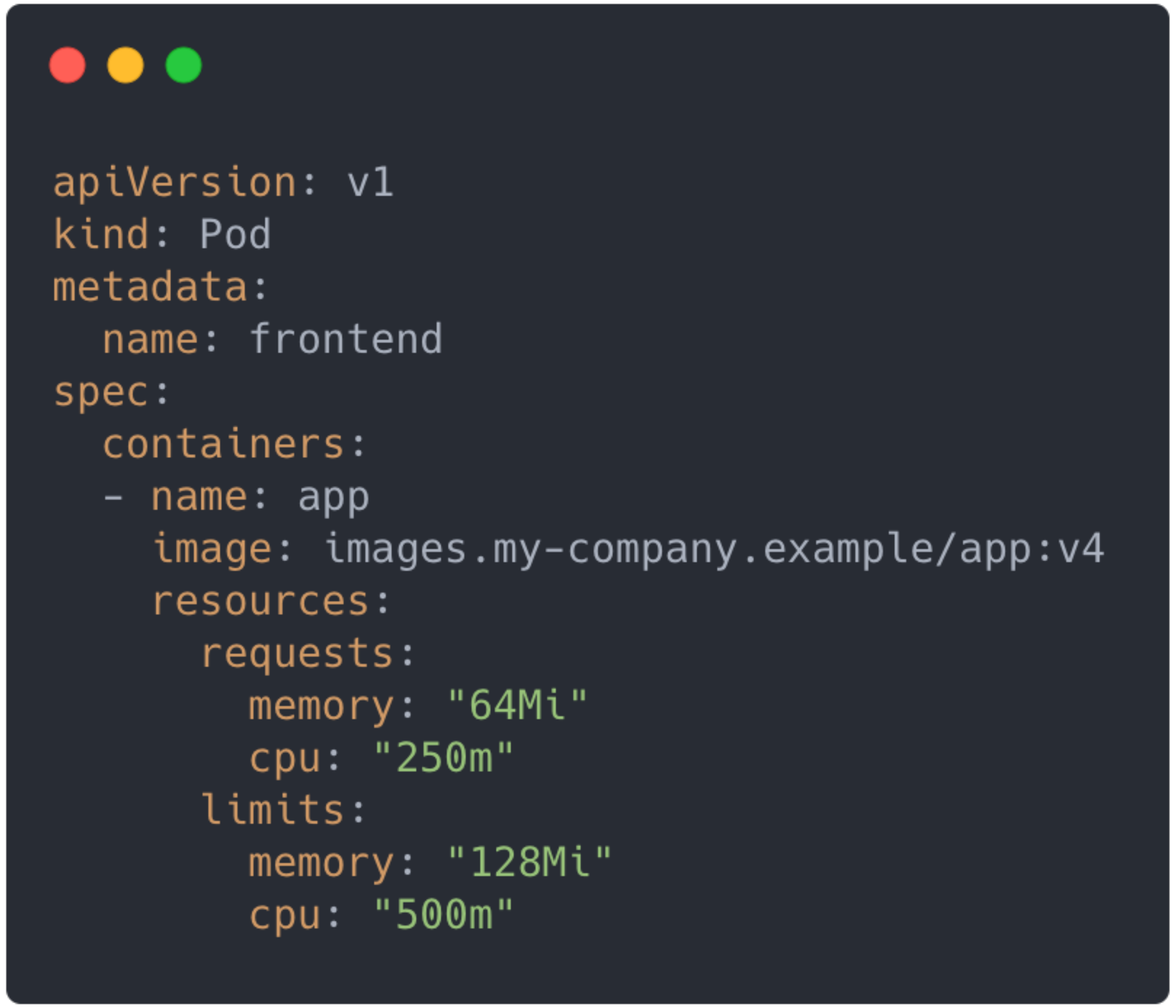
https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

Autopilot: workload autoscaling at Google

Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes

Motivation

- Cloud systems require their users to declare
  - how many instances their workload will need during execution **(horizontal)** , and
  - the resources needed for each **(vertical)**
- **Limits are (mostly) a nuisance to the user**



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Machines, jobs and tasks

- A job is composed of one or more tasks
- A task is executed on a single physical machine
- A single machine executes multiple tasks concurrently
- Workloads
  - Serving: aim at strict performance guarantees on query response time
  - Batch: 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

Autopilot dataflow

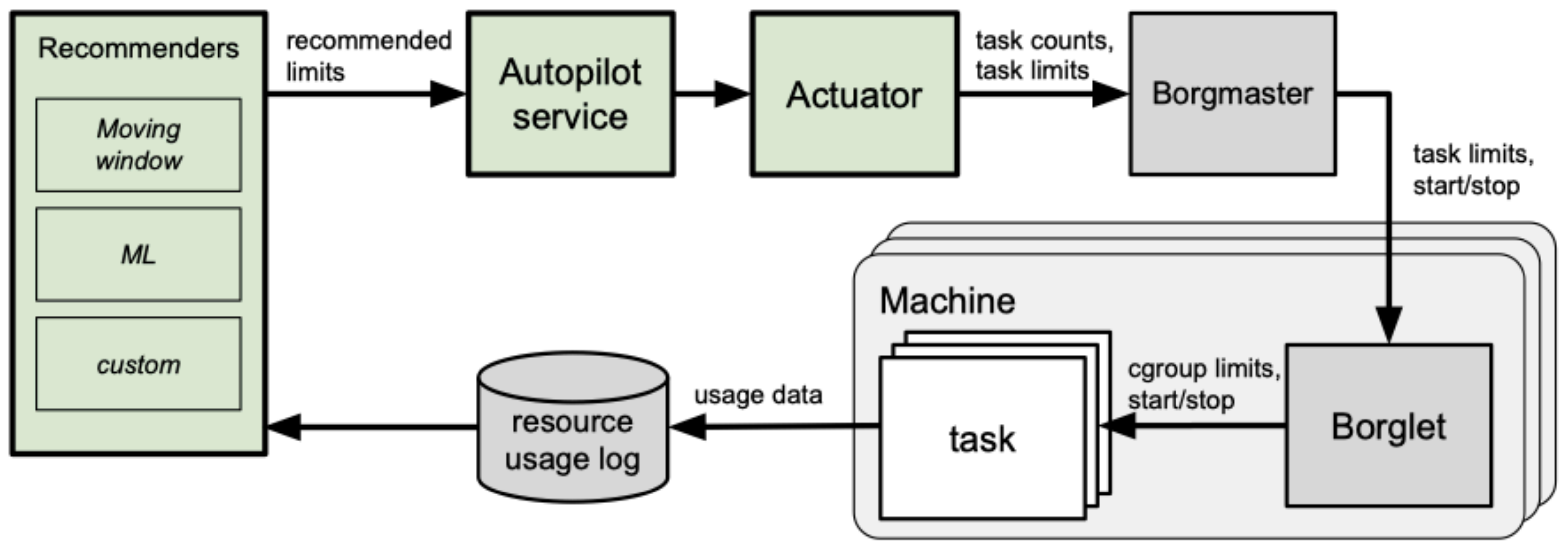
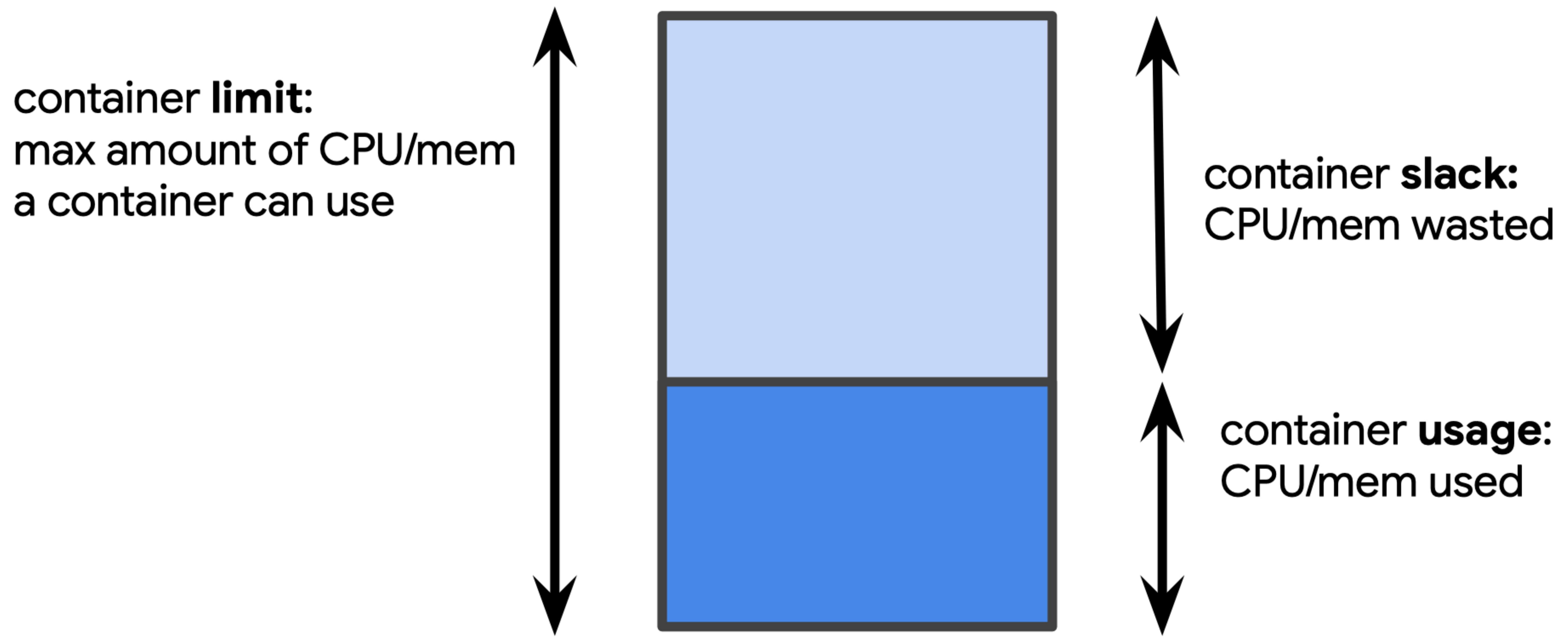


Figure 1. Autopilot dataflow diagram.

Resource management through task limits

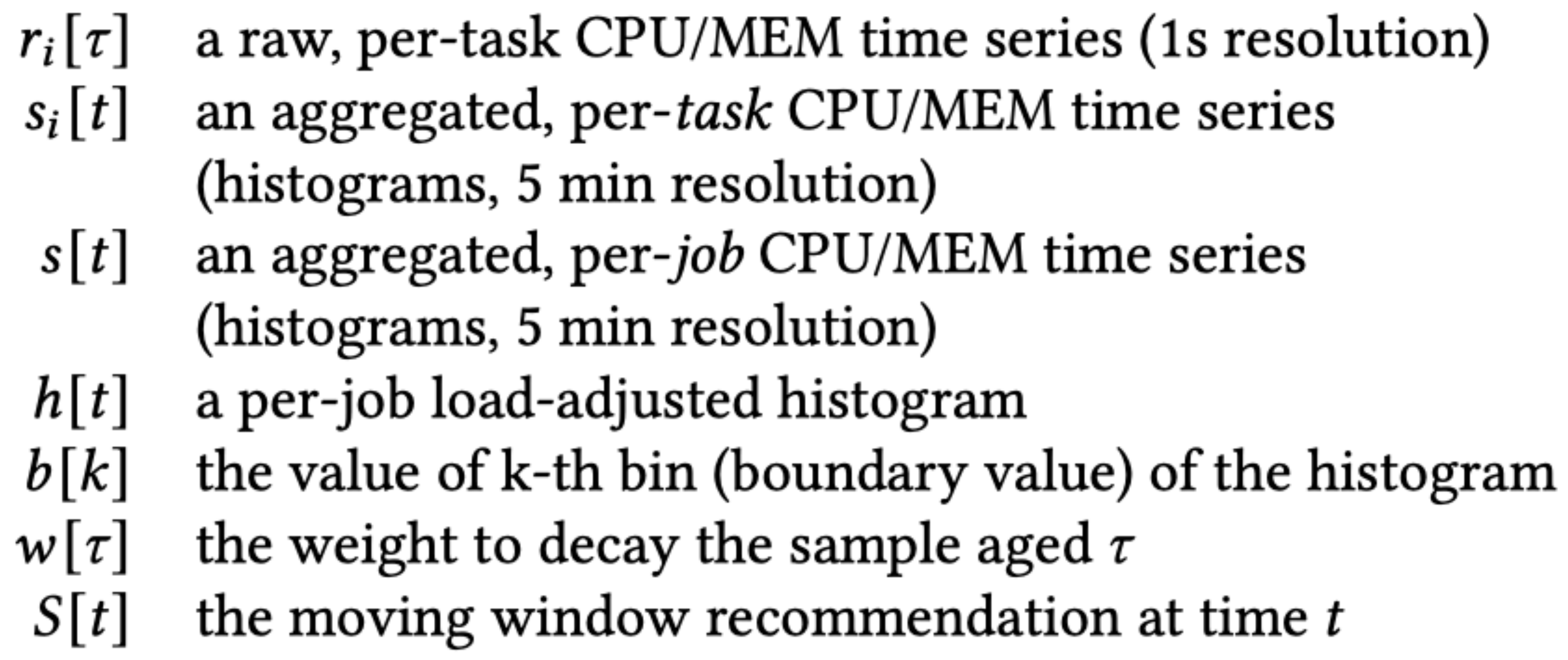
- CPU and RAM are key resources to manage



Automating limits with Autopilot

Preprocessing: aggregating the input signal

- CPU signal
  - the number of raw signal samples $r\ i[\tau]$ that fall into each of about 400 usage buckets
- Memory signal
  - the task's peak (maximum) request during the 5 minute window
- A task is more sensitive to underprovisioning memory (as it would terminate with an OOM) than CPU (when it would be just CPU-throttled)



$r_i[\tau]$	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

Moving window recommenders

- Goals
  - Increase swiftly in response to rising usage
  - Reduce slowly after the load decreases to avoid a too-rapid response to temporary downward workload fluctuations
- **Solution:** Exponentially-decaying
  - Exponentially-decaying weights $w[r] = 2^{-\tau/t_{1/2}}$
  - Half life
    - CPU: 12 hours
    - Memory: 48 hours

Statistics

- **Peak** (S_{max})
  - The maximum from recent samples
- **Weighted average** (S_{avg})
  - The time-weighted average of the average signal value
- **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 $b[k]$ and then returns a certain percentile $Pj\ (h[t])$ of this histogram

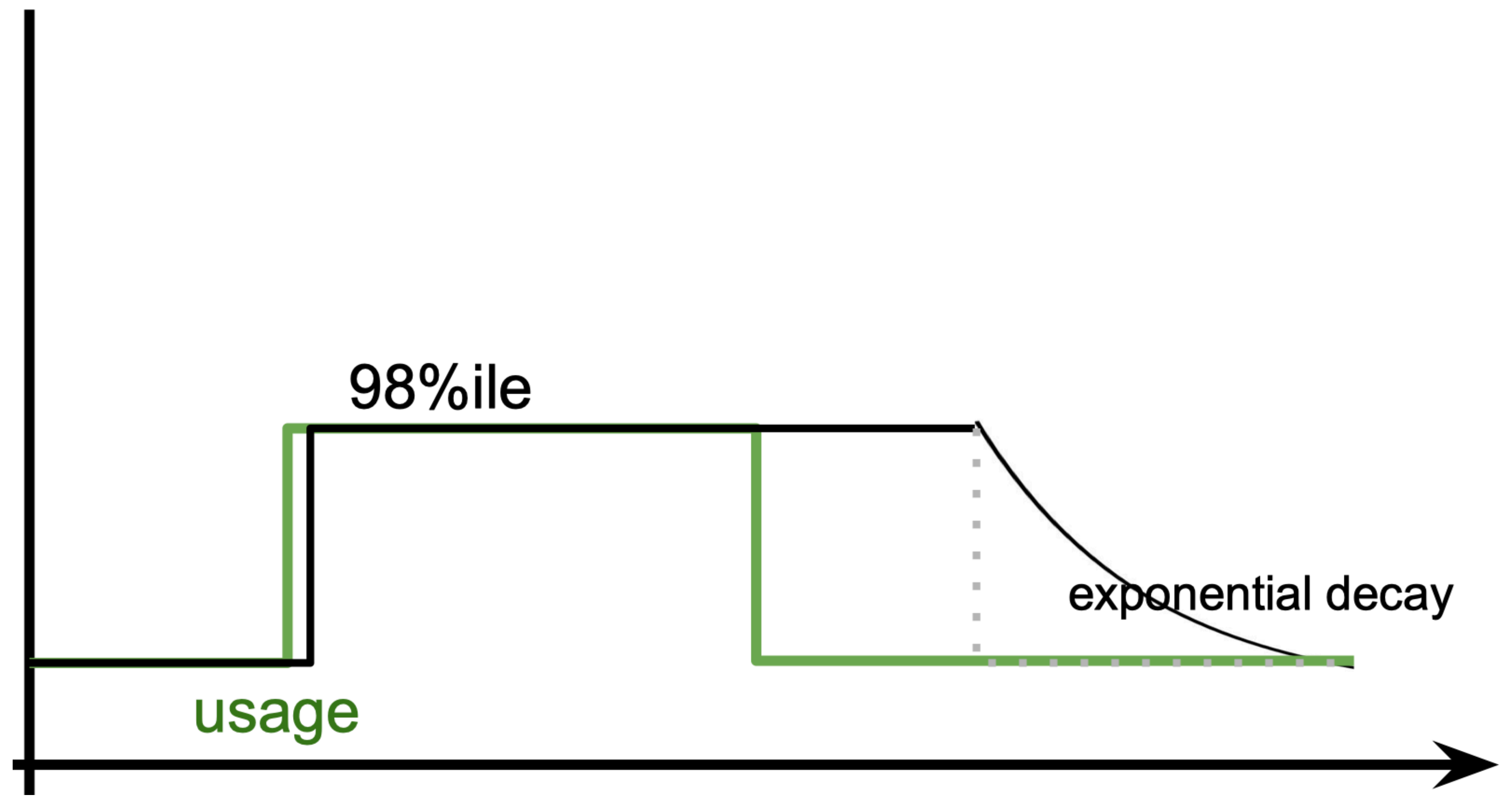
Recommenders based on machine learning

- **Definition:** 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
- **Overrun cost**

$$o(L)[t] = (1 - d_m)(o(L)[t-1]) + d_m \left(\sum_{j:b[j]>L} s[t][j] \right)$$

- **Underpin cost**

$$u(L)[t] = (1 - d_m)(u(L)[t-1]) + d_m \left(\sum_{j:b[j]<L} s[t][j] \right)$$

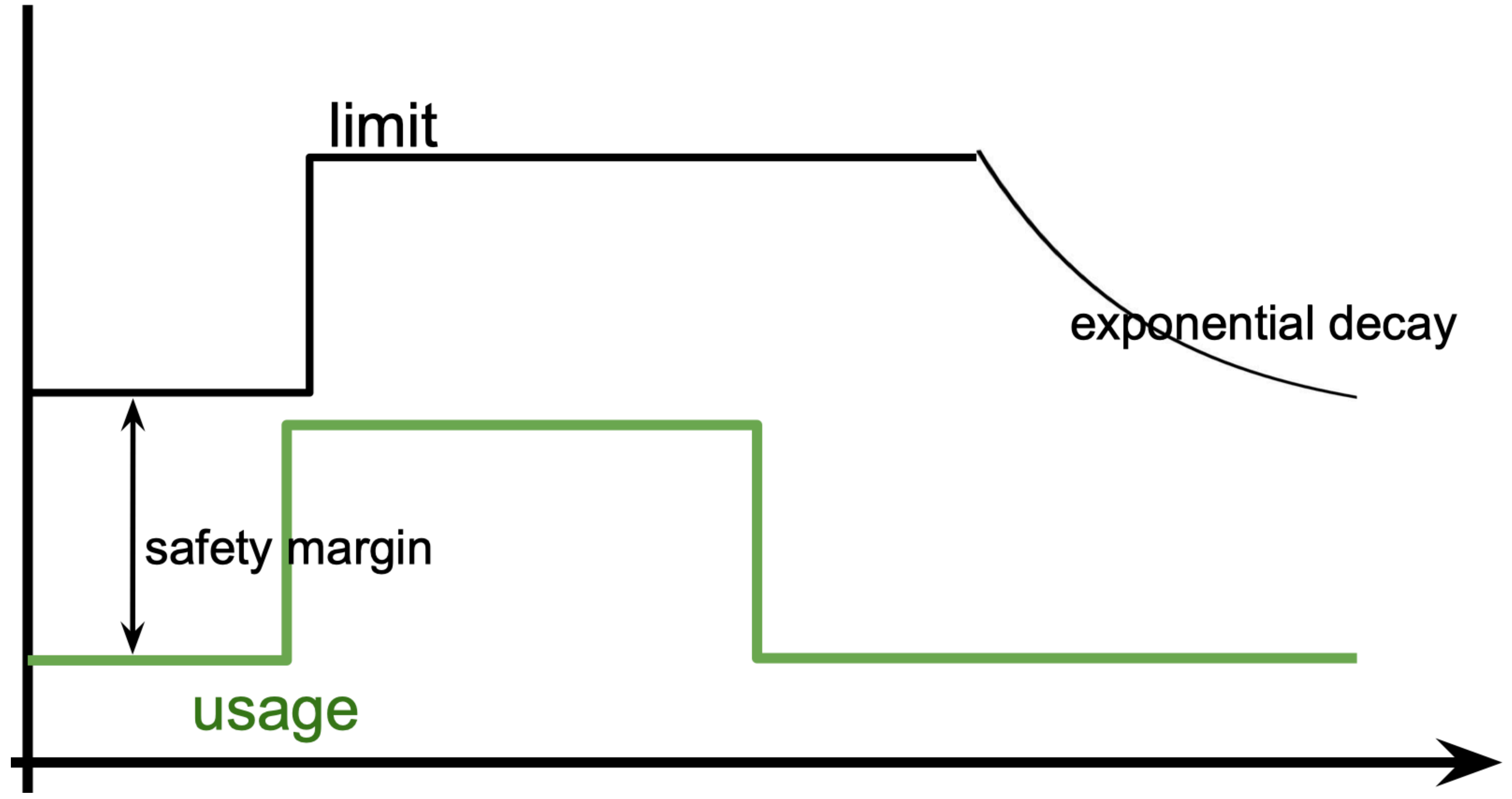


Recommenders based on machine learning

- $L'_m[t] = \arg\min_L \left(w_o o(L)[t] + w_u u(L)[t] + w_{\Delta L} \Delta \left(L, L'_m[t-1] \right) \right)$
- Overruns express the cost of the lost opportunity
- Underruns express the cost of the infrastructure
- The penalty term Δ helps avoid changing the limits too frequently

Recommenders based on machine learning

- The limit is increased by the safety margin $L_m[t] = L'_m[t] + M_m$



Recommenders based on machine learning

- The ML recommender maintains for each model its (exponentially smoothed) cost

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

- The value of a limit tested by the recommender

$$L[t] = \arg\min_m \left(c_m[t] + w_{\Delta m}\Delta(m[t-1], m) + w_{\Delta L}\Delta\left(L[t], L_m[t]\right)\right)$$

Hyperparameter optimization

- **Hyperparameters:** the weights in the cost functions
- **Goals**
  - 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

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

Horizontal autoscaling

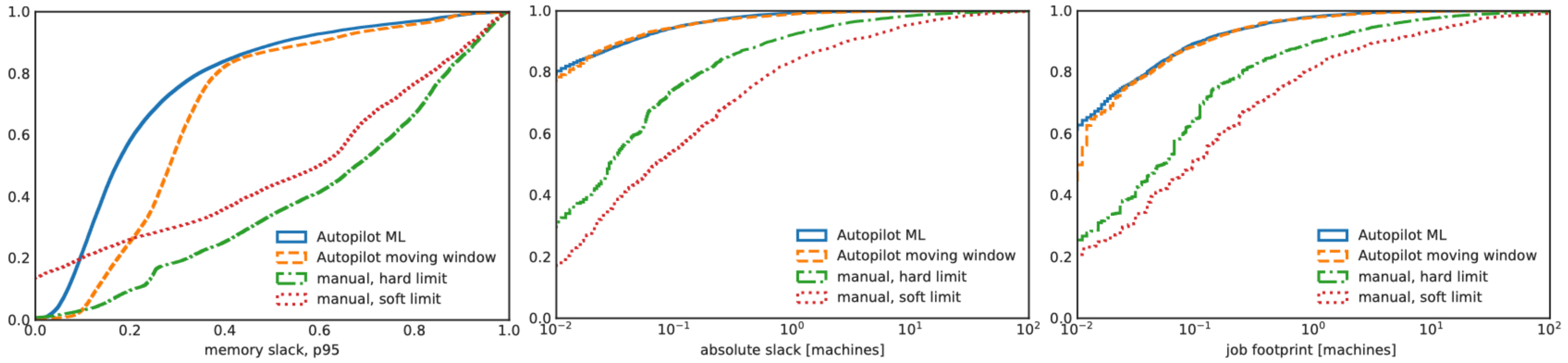
- Vertical autoscaling alone is insufficient
  - A single task cannot get larger than the machine it is running on
- The horizontal and vertical scaling mechanisms complement each other
  - Vertical autoscaling determines the optimal resource allocation for an individual task
  - Horizontal autoscaling adds or removes replicas as the popularity and load on a service changes
- Horizontal autoscaling is used mainly by large jobs

Recommender Quality

Metrics

- **Footprint**
  - Dividing the raw value in bytes by the amount of memory a single (largish) machine has
- **Relative slack**
  - The fraction of requested resources that are not used
- **Absolute slack**
  - The sum of limit-seconds minus usage-seconds over all tasks of a job, divided by 24×3600 (one day)
- **Relative OOM rate**
  - 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.

Resource usage (random sampling)



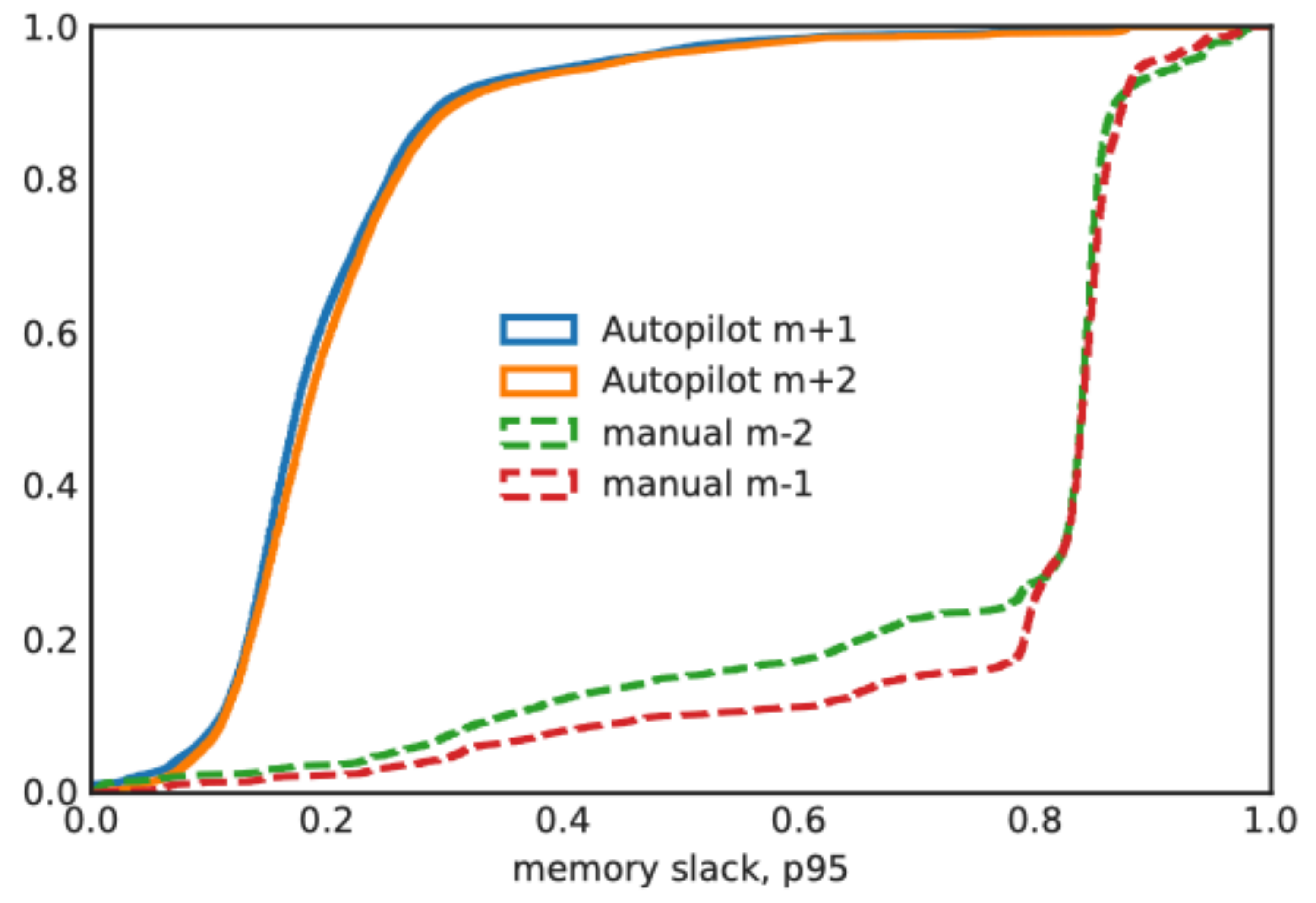
(a) CDF of relative slack (fraction of claimed, but unused memory).

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

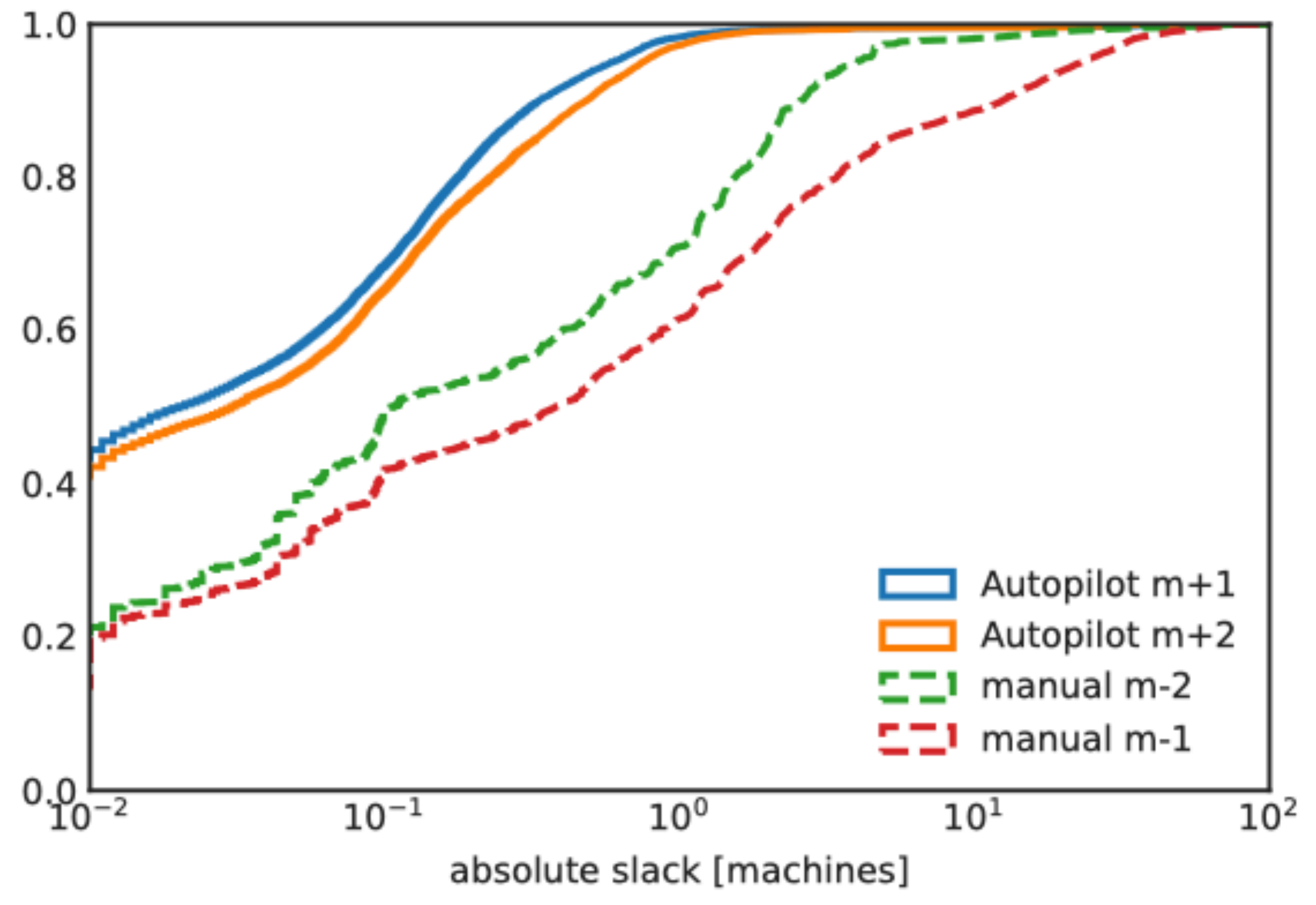
(c) CDF of the footprint (total limit).

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.

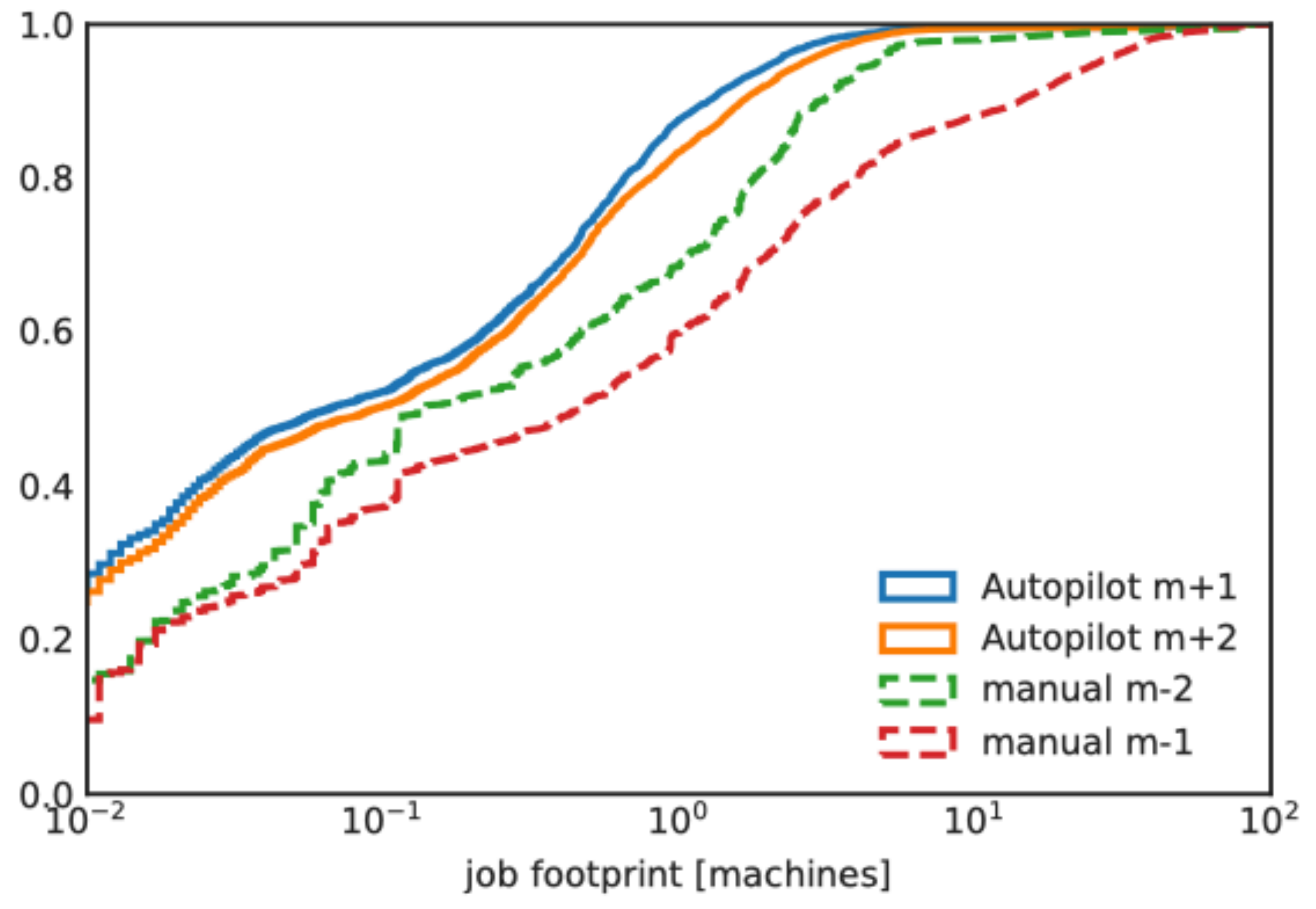
Resource usage (job-days)



(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.

Out-of-memory

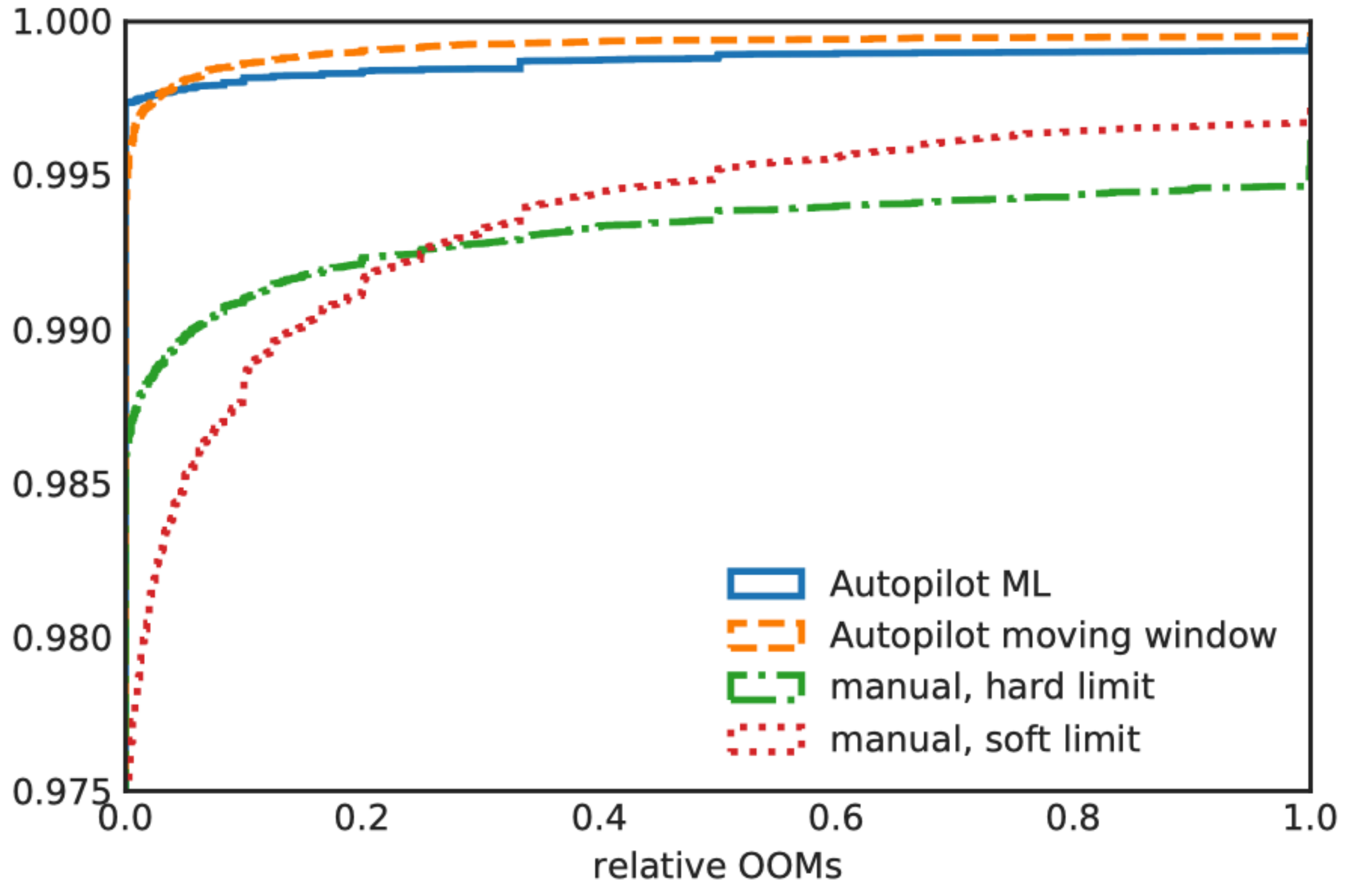


Figure 5. Cumulative distribution function of relative OOMs (number of OOMs per day normalized by the number of tasks) by job-days. Note non-zero y-axis offset – the vast majority of jobs-days have no OOMs, e.g., in the Autopilot cases, over 99.5% of job-days are OOM-free.

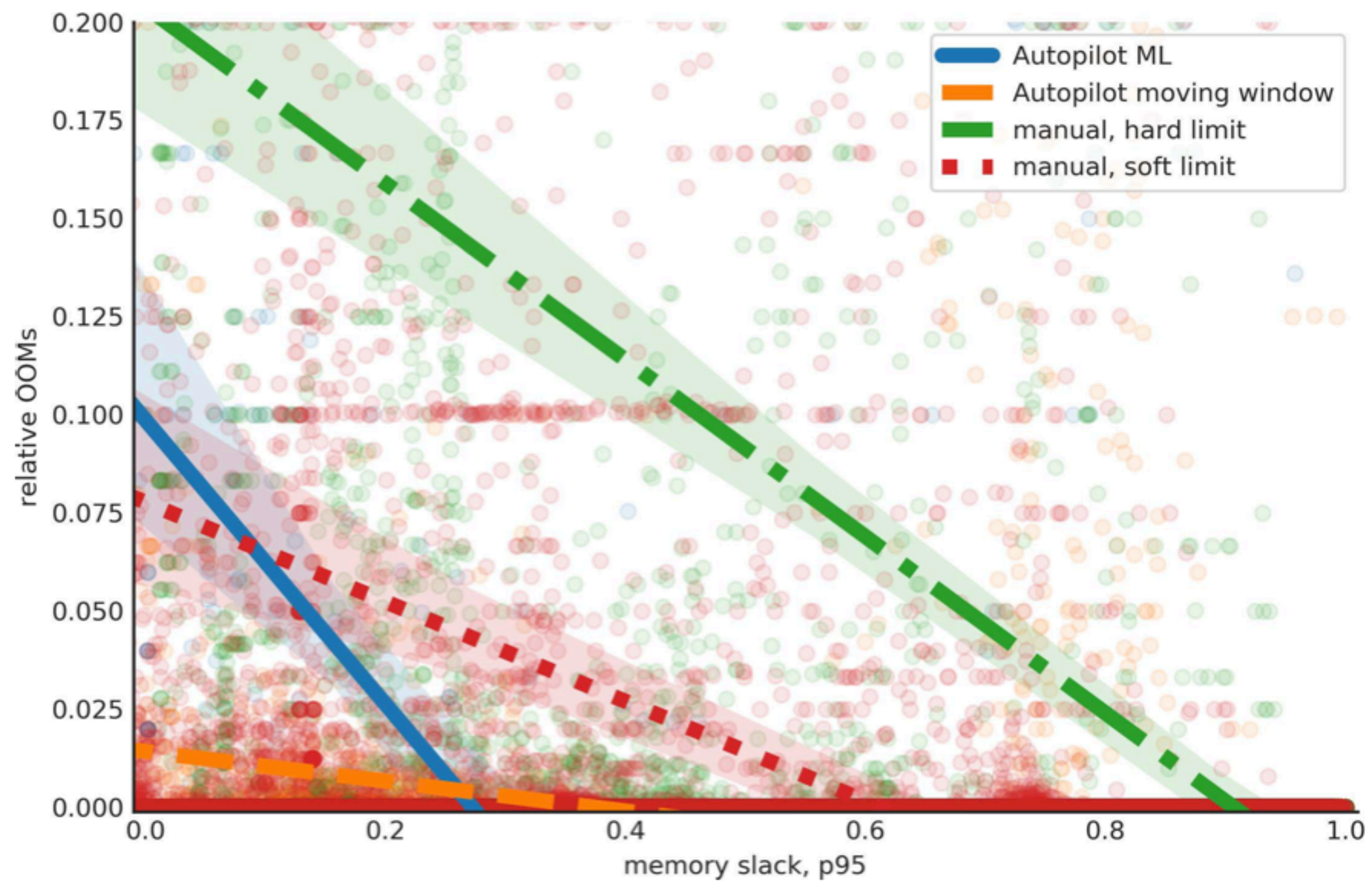


Figure 6. A scatterplot showing OOMs vs slack. A point corresponds to a single job-day; the point's color shows how limit is set for that job on that day. Lines (with a 95%ile confidence interval band) show linear regressions.

Autopilot: Summary

- Efficient scheduling requires fine-grained control of jobs' limits
- Humans are bad at setting the limits precisely.
- Autopilot uses past usage to drive future limits
- Autopilot reduces relative slack by 2x and it reduces the number of jobs severely impacted by OOMs 10x

Thanks for your attention.