CPU-Limits kill Performance: Time to rethink Resource Control

Chirag C Shetty
University of Illinois
Urbana-Champaign
Urbana-Champaign, Illinois, USA
cshetty2@illinois.edu

Sarthak Chakraborty
University of Illinois
Urbana-Champaign
Urbana-Champaign, Illinois, USA
sc134@illinois.edu

Hubertus Franke
IBM Research
Yorktown Heights, New York, USA
frankeh@us.ibm.com

Larisa Shwartz
IBM Research
Yorktown Heights, New York, USA
lshwart@us.ibm.com

Chandra Narayanaswami
IBM Research
Yorktown Heights, New York, USA
chandras@us.ibm.com

Indranil Gupta
University of Illinois
Urbana-Champaign
Urbana-Champaign, Illinois, USA
indy@illinois.edu

Saurabh Jha
IBM Research
Yorktown Heights, New York, USA
saurabh.jha@ibm.com

Abstract

Research in compute resource management for cloud-native applications is dominated by the problem of setting optimal *CPU limits* (c.limit) – a fundamental OS mechanism that strictly restricts a container's CPU usage to its specified c.limit. Rightsizing and autoscaling works have innovated on allocation/scaling policies assuming the ubiquity and necessity of c.limit. We question this. Practical experiences of cloud users indicate that c.limit harms application performance and costs more than it helps. These observations are in contradiction to the conventional wisdom presented in both academic research and industry best practices. We argue that this indiscriminate adoption of c.limits is driven by erroneous beliefs that c.limits is essential for operational and safety purposes. We provide empirical evidence making a case for eschewing c.limits completely from latency-sensitive applications. This prompts a fundamental rethinking of auto-scaling and billing paradigms and opens new research avenues. Finally, we highlight specific scenarios where c.limits can be beneficial if used in a well-reasoned way (e.g. background jobs).

CCS Concepts

• **Computing methodologies → Distributed computing methodologies**; • **Computer systems organization → Maintainability and maintenance**; **Cloud computing**; • **General and reference → Performance**; **Reliability**.

Keywords

Cloud Computing, Microservices, Autoscaling, Resource Optimization, Site Reliability Engineering, Container Orchestration

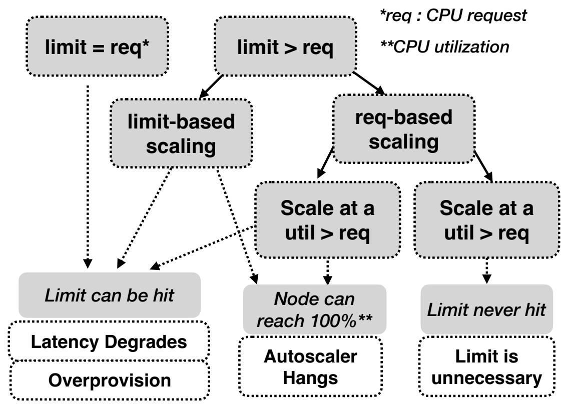


Figure 1: c.limits are either harmful or unnecessary: Summary of §3.

ACM Reference Format:
Chirag C Shetty, Sarthak Chakraborty, Hubertus Franke, Larisa Shwartz, Chandra Narayanaswami, Indranil Gupta, and Saurabh Jha. 2025. CPU-Limits kill Performance: Time to rethink Resource Control . In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3772052.3772219

1 Introduction

The problem of finding optimal resource allocation for containerized microservices (*rightsizing*) and automatically adjusting it with varying load (*autoscaling*) has been a major focus of research and industry practitioners over the past decade. The objective of autoscaling research is to devise algorithms to meet the application Service Level Objectives (SLO) [41] with minimal resource allocation [63, 64, 69, 72, 75, 81, 82]. These are built into orchestrators like Kubernetes (K8s) [34], Borg [80], Twine [77]. They then handle the distribution of shared resources like CPU & memory among the microservices. Management of CPU resource is of special interest since CPU usage closely correlates to request processing latency & is often the primary resource bottleneck [73].

We argue that these efforts are misguided, as the fundamental problem of CPU allocation for containers lies not in the policies

Properties	FIRM [72]	Cilantro [64]	Autothrottle [81]	Ursa [82]	SHOWAR [63]	Erlang [75]	H(V)PA [49, 59]
CPU alloc. mechanism	C.LIMITS	C.LIMITS	C.LIMITS	C.LIMITS	C.LIMITS	VM*	C.REQUEST
Works without C.LIMITS?	✗	✗	✗	✗	✗	✓	✓
What exactly fails on disabling C.LIMITS? [2]	§3.4, Reward fn. r_t assumes $\frac{RU_i}{RLT_i} \leq 1$, else $RLT_i = 0$ trivially.	§4.2, perf $p(a, l)$ is f(alloc 'a'), and usage > alloc not considered.	§3.2.1 Uses throttle count as the main metric. NA w/o C.LIMIT.	§MIP1 uses fixed R_i for each LPR, obtained by profiling w/ C.LIMITS.	§3.2 - 3.3: uses slack (e), algo devolves to HPA if slack is negative.	NA	NA
C.REQ scaled with C.LIMIT? [3]	✗[6]	✗[14]	✗[24]	NP**	NP**	NA	NA
Container sizing technique	Reinforcement Learning	Fixed	Contextual Bandit	Profiling & MIP	Three-sigma rule	Fixed	Fixed(Manual)

Table 1: Use of C.LIMITS in recent works on microservices CPU sizing/autoscaling. *Erlang uses one node(VM) per pod replica. [1]Horizontal scaling threshold manually set, not found in artifact. [2]Relevant sections in that paper. [3] Relevant code artifact cited. NP**: Code publicly not available.

(*how much* to allocate), but rather in the mechanism used today(*how* it is allocated). *CPU-Limits* (we will use C.LIMITS and C.LIM in short) is a widely adopted mechanism, built on top of Linux's `cpu.cfs_quota_us` [22, 23, 63, 64, 82]). Intuitively, C.LIMIT (typically specified in millicores) defines an upper bound on CPU usage beyond which the container is throttled. In the event of such throttling, application latency can dramatically degrade. Thus, "avoiding throttling" spawned academic autoscaling research focusing on algorithmic innovations to automatically adjust C.LIMITS (Tab. 1). Despite these efforts, the techniques, however, have seen low adoption in practice [9, 13]. Practitioners started noticing that the impact of throttling on SLO and predictability outweighs the benefits of C.LIMITS, if any (§3). So, we question, *do we really need C.LIMITS?*

C.LIMITS have a history. Initially, on individual machines, users could control CPU allocation using *shares* (`cpu.cfs_shares`). The Linux CFS Scheduler (or recent EEVDF [16]) allocates CPU time to processes proportional to their specified share [15, 19, 36]. *Quota* (`cpu.cfs_quota_us`, aka bandwidth control) was introduced in Linux 3.2 [1] for two purposes: ❶ To have a *conceptually simpler & predictable* way of specifying CPU allocation, in contrast to *share*, wherein CPU cycles a process gets is *relative* to CPU use of all other co-located processes, and ❷ To support pay-per-use – Operators needed a way to assign an explicit upper bound to CPU usage by tenants based on how much they paid. Subsequently, as containerization on cloud became popular, quotas & shares got adopted for container sizing (as C.LIM & C.REQUESTS in K8s terms).

At first, needing C.LIM seems logical. When deploying applications in multi-tenant environments, we do not want containers to use excessive CPU, thereby depriving co-located containers of CPU resources, causing unpredictable latencies and potentially expensive cloud bills. Thus, having C.LIM was (and still is) considered a "best practice". For instance, in K8s, for critical containers to have the highest Quality of Service, they must have C.LIM (and equal to its C.REQ) [8], in spite of the risks of throttling.

However, we raise two important considerations: First, use of C.LIMITS for ❶ is redundant today, because the orchestrators ensure the sum of C.REQUESTS of all containers placed on a node is less than the node's capacity [35]. This makes C.REQUESTS an absolute minimum CPU guarantee(§3.4) - we do not need C.LIM on top of C.REQ. Secondly, C.LIMITS is non-intuitive to set and has adverse impacts on *all the three* key metrics – application latency, reliability, and cost, offering no benefits (§3, summarized in Fig.1). So we posit that the community should not spend efforts on setting the right C.LIMITS, and provide directions on 'what else if not C.LIMIT'.

Our interviews with SREs (Site Reliability Engineers) revealed that the DevOps community is split too on the "limits vs. no limits" debate. Some insist CPU limits are essential to limit resource misuse, while others argue against it [4, 11, 12]. In this paper, (1) we discuss both sides, and provide empirical evidence to eliminate C.LIMITS from latency sensitive application management. (2) However, going C.LIMITS-free requires fundamental rethinking in existing autoscaling and billing paradigms. We elaborate research directions and demonstrate the potential by building a prototype autoscaler (§4). (3) Finally, judicious use of C.LIMITS in specific scenarios is warranted (e.g., background jobs). We highlight them while debunking several myths (§5) that promote the use of C.LIMITS.

2 Background

Microservice applications typically are deployed as containers on a cluster of *nodes* (physical machines or VMs) running Linux. Container orchestration systems like Kubernetes (K8s) [34], Borg [80], Twine [77], etc. co-locate multiple containers on the same node, with Linux's Completely Fair Scheduler (*CFS*) on each node multiplexing CPU time among the containers based on their CPU allocations. We use the K8s term **'pod'** interchangeably with 'container'.

CPU Shares and Quotas: *CFS* gives processes a minimum CPU time proportional to their *share* (`cpu.cfs_shares`) and no more than their specified *quota* (`cpu.cfs_quota_us`) [15, 78, 79]. A process's actual CPU time changes dynamically with more processes. In each scheduling period of 100ms, two processes $\mathcal{A}, \mathcal{B}$ with 400 and 600 shares get a guaranteed 40ms ($\frac{400}{400+600} \times 100$ms) and 60ms of CPU time respectively. If $\mathcal{A}$ uses only 20ms of its allocated time, $\mathcal{B}$ is free to use the remaining 80ms. However, if $\mathcal{B}$ had a quota of 60ms, it cannot exceed 60ms even if a residual 20ms is available. When $\mathcal{B}$ hits 60ms, it will be throttled until the next period.

CPU-Requests and limits: For each container, the user can specify the CPU allocation using CPU-Requests and CPU-Limits (we'll use C.REQ, C.LIM in short). Effectively, ***C.REQ is the minimum guarantee and C.LIM is the maximum allowance on a container's CPU usage.*** Millicores (m) units are used, where a millicore is 1ms worth of CPU time every second; so 1000m implies an allocation of one CPU core. Internally, K8s translates C.REQ and C.LIM to `cpu.cfs_shares` and `cpu.cfs_quota_us` respectively. Importantly, K8s scheduler [54] ensures that the sum of C.REQ of all containers on a node is less than the node's net available CPU to prevent over-allocation and safeguard the C.REQ's guarantee. Container can use more than its C.REQ when free CPU cycles are available, but if its CPU utilization (CPU.Util) reaches C.LIM it will be throttled. Note C.LIM ≥ C.REQ and it is optional to set C.LIM.

Horizontal Pod Autoscaler (HPA): HPA is K8s's native and most commonly used autoscaler [49]. To enable autoscaling, user sets a scaling threshold (CPU.Thres). When the container's CPU.Util

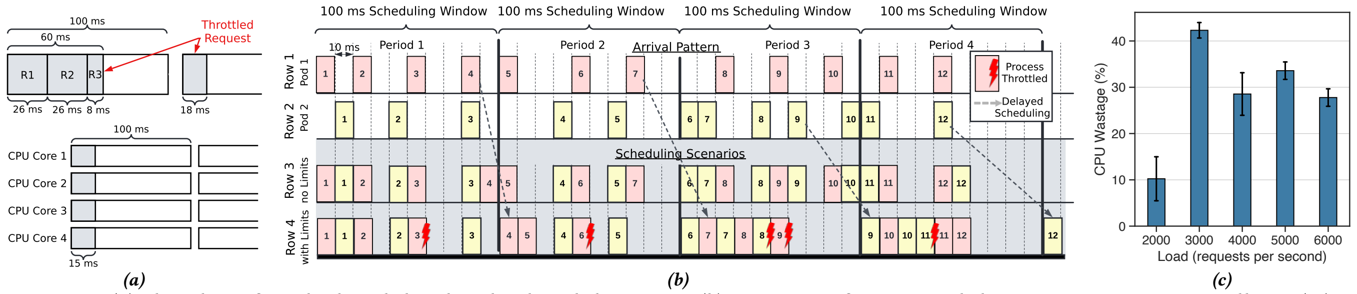


Figure 2: (a) Throttling of single-threaded and multi-threaded processes. (b) Formation of queues: Pods have c.requests = 300m millicore (m), i.e., 30ms per 100ms. In Row-4, c.limit of 300 millicores is applied on both. We assume fair scheduling (with no preemption, for the sake of simplicity). (c) Impact of c.limits on cost: Percentage increase in CPU required to meet SLO with c.limits specified using c.limits vs just c.request (SN app)

reaches a user-set scaling threshold (CPU.Thres), HPA adds another replica of the container with the same c.req. CPU.Thres is usually specified as a percentage (e.g. 70%) of the container's c.req.

2.1 Prevalence of CPU-Limits

The key idea of using c.lim to prevent containers from using more than the specified amount of resources is widespread.

In Industry: Examples of the use of c.limits in industry:

(1) K8s assigns Quality of Service (QoS) classes to containers based on c.limit. To have the highest QoS class, a container must have c.limit set and must be equal to its c.requests [53]. DevOps engineers thus add c.limit on critical pods [37, 47, 60].
(2) Multi-tenancy best practice considers c.limits (quota) as best practice [25, 26, 38, 39].
(3) Popular software projects, in attempts to become 'container friendly', use c.limit to infer internal parameters like thread-pool size, processor count etc [29, 31, 40].
(4) Managed services by cloud providers, like GCP's Autopilot [17, 74], automatically apply c.limits on customer pods.
(5) Deployment templates in the widely used Helm Chart library Bitnami come with pre-set c.limit values [27].

In Research: Significant research effort has gone into 'finding optimal c.limits to minimizes CPU allocations while meeting application SLO' [63, 64, 72, 74, 75, 82]. Table 1 lists the most recent autoscaling systems from top conferences. They rely on c.limits to specify container CPU allocations. Tuning c.limits is used as the primary mechanism of trading off CPU allocation for latency.

3 CPU-Limits Considered Harmful

We now present evidence of the disconnect between industry standards, academic research, and practitioners' experiences with using c.limits. We systematically argue for eliminating the use of c.limits completely in latency-sensitive application deployments.

3.1 The Ongoing Debate on Limits

This section is motivated by industry debates around the benefits of limits [5, 7, 10, 11, 17, 20, 61]. Our study of online discussion forums reveals that cluster administrators are *already getting rid* of c.limits [4, 20]. We also interviewed several DevOps personnel at KubeCon (premier K8s conference) and inside a large cloud provider company. We found that admins would *largely prefer not having to set c.limits* due to frequent latency degradation from c.limit's throttling. Many use manual autoscaling, causing high management overheads and underutilization, e.g., *over 65% of Kubernetes workloads use only 50% requested CPU resources [13].* We present select anonymized quotes from our interviews:

- (Major Insurance Provider) "*A service with 16 CPUs didn't utilize more than 10 CPUs due to throttling. We eliminated the issue by eliminating c.limits altogether.*"
- (Major SaaS Enterprise) "*c.limits is very non intuitive, especially for multi-threaded examples. We are forced to use c.limits...*"
- (Major Sports Company) "*(We) need a way for developers to not worry about resource requirements and CPU measure. We stopped using limits; we mostly only use HPA.*"

So, what is the *expected* benefit of using c.limits? We encountered three common reasons why DevOps personnel believe c.limits are useful: (1) for guaranteed CPU allocation & predictable performance, (2) to provide safety & cluster stability despite runaway containers that consume a lot of resources [32], especially in multi-tenant deployments [39], and (3) to limit dollar costs for their application. However, our measurements will show that c.limits lead to unpredictable performance and cluster instability. Fig. 1 is a decision tree that summarizes our arguments in the rest of this section. In all cases, the only way forward is to eliminate limits from autoscalers.

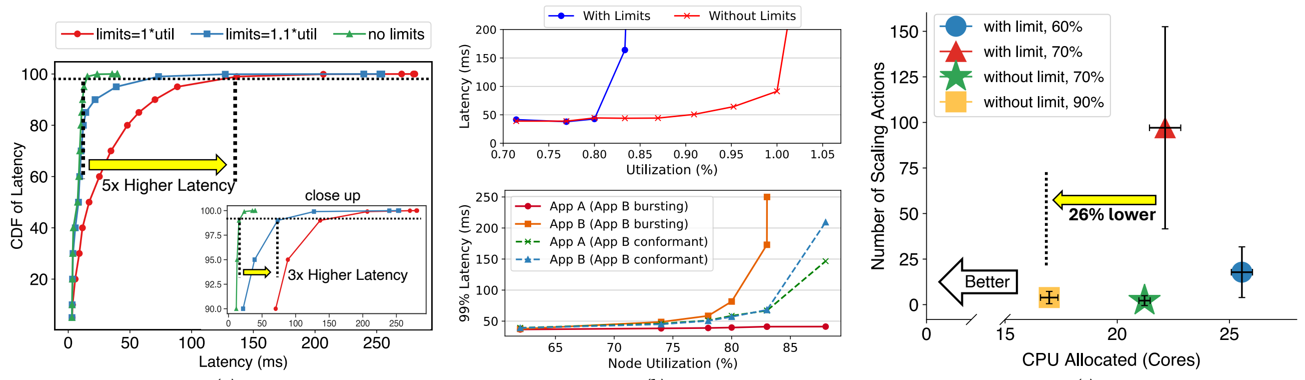
Setup: We deploy HotelReservation and SocialNetwork applications from DeathStarBench [68] on EC2 clusters. We optimize the CPU allocation vs latency over various c.limits, c.requests, and scaling thresholds (CPU.Thres) settings under varying loads.

3.2 c.limits Exacerbate Latency & Reliability

First, we observe that using c.limits to reduce CPU allocation by limiting the CPU utilization of pods can have adverse impacts.

Bad for Latency: For the same amount of CPU utilization of a pod, applying c.limits on it results in worse latency characteristics. In Fig. 3a, we put c.limit on only one (out of 19) pods of HotelReservation. Even *without* attempting to reduce its actual CPU utilization, the end-to-end tail latency degraded by 5×.

c.limits harm application latency in two distinct ways: (1) *throttling* of individual requests, and (2) formation of *long queues* across requests. Consider a pod with c.limits of 600 millicores, i.e., upto 60ms of CPU time within every 100 ms scheduling period. Fig. 2a (top) shows a single-threaded container, where requests take ∼26



(a) (b) (c)

Figure 3: (a) Impact on latency with C.LIMITS of 1× & 1.1× the CPU util (HR). (b) [top] Queuing with and without C.LIMITS. X axis is % utilization of pod's allocated C.REQ/C.LIM. [bottom] C.REQ protects app A against bursting app B (c) No. of scaling actions & CPU required to meet SLO on increasing load by 25% different scaling thresholds (60%, 70%, 90%) with & without C.LIMITS (SN) (HR = HotelReservation, SN = SocialNetwork [68]).

ms. Every one in three requests (e.g., R3) will be throttled [21] for 40ms, resulting in an execution time of 66ms. If the containers were multi-threaded, like Figure 2a (bottom), the limit constrains the *cumulative* CPU time across all threads (e.g., avg. of 15ms CPU runtime per core with four threads).

Further, throttling can form long queues as illustrated in Fig. 2b. Consider, two pods P_1 and P_2 co-located on a single-core node. Each pod gets a load with a Poisson arrival rate of 30 req/sec. Every request needs 10 ms of CPU time, translating to a net CPU utilization of 300 millicores (30ms per 100ms of scheduling period). We show two request execution schedules (1) when each pod is allocated 300 millicores C.REQUESTS (Row-3), and (2) when 300 millicores C.LIMIT is applied (Row-4). In both cases, pod CPU utilization (30%) and net node utilization (60%) are nearly the same, but average latency increases significantly with C.LIMIT. With C.LIMITS, typical microbursts of Poisson arrival form queues taking multiple scheduling periods to drain. E.g., in period 1, P_1 gets a microburst of 4 requests (instead of average of 3). Thus, request 4 gets delayed due to throttling, forming queues for requests 7 & 10.

But isn't this regular queuing that is expected when job arrival rate matches the processing speed? No. C.LIMIT-induced queuing is worse because (1) regular queuing happens only during high *node* CPU utilization. C.LIMIT causes queueing at each *pod*, and thus happens more often. (2) In regular queuing, a job in the queue waits for all the jobs in front to finish. With C.LIMIT, in addition, the job waits for cumulative throttling times. Thus, queues form more rapidly and at lower loads in C.LIMIT-ed pods as shown in Fig. 3b (top).

Bad for Reliability: In our experiments, tail latency degradations due to C.LIMITS occurred frequently and lasted multiple seconds–triggered even during routine cluster actions like scaling, updating deployments, or adding/removing nodes in the cluster. Few cases resulted in cascading failures to other parts of the system. For instance, with long queue formation, the pod's memory consumption exceeded its memory limit, causing pod restarts. This reduced the overall capacity and led to other pods hitting their limit or causing request timeouts. Others have noted similar behavior [28, 33].

Takeaway 1: *If a pod's CPU utilization hits C.LIM, latency drastically degrades & can cause cascading failures. C.LIM creates a false allocation-vs-latency tradeoff, without actually reducing the CPU.UTIL.*

3.3 Why Not Set Higher CPU-LIMITS?

A natural solution is to *always* ensure that the C.LIMITS is higher than the actual container utilization. However, we argue:

(1) If the C.LIMIT is ever hit, it causes severe latency degradation, and (2) If the C.LIMIT is never hit, then it is unnecessary. Two cases are possible depending on what we do with C.REQUESTS:

Case A : C.LIMITS = C.REQUESTS → Increased Cost:

To avoid throttling, we must maintain a *margin* between a pod's CPU utilization and C.LIMIT. Adding this margin to C.REQUEST translates to larger pod footprints, which in turn require more nodes for placement since pods are packed according to their C.REQUEST [54]. In Fig 2c, we manually optimized SocialNetwork to get the lowest C.LIMIT-based allocation while meeting an SLO. Further, to maintain this margin at all times, we configure `HPA` with a scaling threshold(CPU.THRES) of (100 minus margin).

To maintain the tail latency, it took 25-45% higher net C.REQUESTS than the overall observed CPU utilization. In other words, we had to increase the C.LIMITS by ~30%, roughly split as a 10% increase to handle Poisson microbursts, 10% for fluctuations from cluster actions (eg: pod updates, scaling) and configurations (eg: child and parent colocated on a node [83]), and 10% for load variations (e.g. [62]). This also explains why the industry rule of thumb for margin is 20-30% [42]. However, it varies by application, load levels etc–in Fig 2c, `nginx-thrift` pods used 30%, while `user-timeline-service` needed 45 %. Any attempts at reducing the margin (i.e, increasing scaling threshold) can lead to unpredictable, long convergence time to meet the SLO with varying load. In Fig. 3c, we increase the load on SocialNetwork by 25% and measure the time taken to meet SLO. With a larger scaling threshold, CPU allocation reduces but leaves a smaller margin between the utilization and C.LIM (which causes throttling), trading it off with predictably meeting SLO (Fig 3c). We observe a 4×increase in the time to meet SLO when the scaling threshold increases from 60% to 70%! This shows the sensitivity of autoscaling to the scaling threshold with C.LIM.

C.LIMITS are too fine-grained for Overall Dollar Costs: Developers who want their application to stay within a dollar budget today need to grapple with *multiple* C.LIMITS of individual containers. This is cumbersome, error-prone, and *too fine-grained* for application-level goals. The need to maintain enough margins when applying C.LIMITS and the difficulties in doing so cost-effectively explain the complaints by system admins as in §3.1.

Takeaway 2: *Avoiding performance impact of C.LIMITS leads to over provisioning. Minimizing that cost is non-trivial.*

Case B : C.LIMITS > C.REQUESTS → Autoscaler hangs

Unlike Case A, if we increase only C.LIM without increasing C.REQ, we can prevent throttling while not increasing the cost. This is, in fact, a common practice [72, 81]. Depending on what we set CPU.THRES to, either C.LIM will never be hit, rendering it unnecessary, or autoscaler may 'hang,' leading to long periods of SLO violation. If CPU.THRES is lower than C.REQUESTS–the autoscaler creates a replica before the container utilization reaches C.REQUESTS–the C.LIM will never be reached. In this case, C.LIMITS serves no purpose.

If CPU.THRES is higher than C.REQUESTS, it scales when container utilization > C.REQUESTS (e.g., all autoscalers in Tab. 1). But remember that a pod is only guaranteed its C.REQUEST. Thus, if the pod is on a tightly packed node running at high utilization, the pod's utilization may never exceed its C.REQUEST and thus not reach CPU.THRES. Autoscaler will not add additional pod replicas, and hence, the SLO can remain violated indefinitely. C.LIMITS > CPU.THRES > C.REQUESTS may thus lead autoscalers to get stuck.

Takeaway 3: *C.LIMITS higher than C.REQUESTS are unnecessary at best, and cause autoscaler to hang at worst.*

Conclusion: No matter how C.LIMITS are applied on latency sensitive pods, it can only harm the performance, cost, and reliability.

C.LIMITS is unnecessary and not the right tool for CPU resource management. Among the researchers who acknowledge this, a common refrain then is – "*But C.LIMIT is a 'necessary evil' to deal with adverse & failure scenarios like CPU hogging/bursting/runaway/buggy pods?*". Judicious use of C.LIM has valid use-cases, & we detail it in §5. However, we argue that effective use of C.REQUESTS is adequate in most of these scenarios & remains an unexplored direction. Use of C.LIM in fact should be an exception rather than the rule.

3.4 C.REQUESTS are sufficient

C.REQUEST is sufficient to ensure a container gets its CPU allocation, irrespective of its co-located containers' CPU usage. It follows from the OS's fairness promise & the orchestrator's gate-keeping :

CFS's Promise: CFS provides proportional fairness guarantee: If n pods $\{P_1, P_2, ...P_n\}$ with C.REQUESTS of $\{r_1, r_2...r_n\}$ are running on a node with C CPUs, CFS ensures that each P_i gets a *minimum* CPU time of $\frac{r_i}{\sum_1^n r_j}C$, no matter what the co-located pods are consuming [58]. Internally, CFS uses C.REQUESTS to weigh the container's usage while deciding the scheduling priority.

Orchestrator's Gate-keeping: Placement algorithms in container orchestrators like K8s ensure $\sum_1^n r_j \leq C$. *Thus C.REQUESTS r_i is the absolute minimum guarantee of CPU that P_i will get.*

Consequently, CPU allocated using C.REQ is guaranteed even without using any C.LIM. **We call a pod *conformant* if its CPU.UTIL ≤ C.REQUESTS, and as *bursting* otherwise**. Pods are free to use CPU beyond their C.REQ allocation . But they can not 'steal' CPU from conformant pods. A bursting pod can drive up the node utilization. This however, will only impact its latency.

We demonstrate this in Figure 3b(bottom). We co-locate two SocialNetwork apps A & B on a 4 core node. App A has a constant load and is conformant with C.REQ of 1100m. App B has a growing load, but a fixed C.REQ of 700m. As the node CPU utilization increases due to bursting B, the latency of B degrades. A is not impacted. From A's viewpoint, the node is only 45% $\frac{(1100+700)}{4000}$ occupied at all times. Note however, that the node has spare C.REQ of 1200m ($= 4000 - (1100 + 700)$). Thus, the orchestrator can pack more conformant pods on the node. Then A may see increased latency with higher node utilization. Dotted lines shows the latency if B were conformant as its load increased. Remember that this degradation would happen even if the conformant pods had C.LIMITS– C.LIMITS would only degrade the latency further due to throttling.

4 Beyond C.LIMITS: Problems & Solution Sketch

Abandoning the use of C.LIMITS compels a course correction in CPU resource management practices. Here we call attention to two prominent areas that can benefit from fundamental rethinking: (1) Autoscaler designs, and (2) Billing paradigms. We build an illustrative prototype "Yet Another AutoScaler".

4.1 Rethinking Autoscaler Design

Today, the concept of C.LIMITS is inherent in the modeling of autoscaling as an optimization problem. Removing limits renders many past works inapplicable for C.LIMIT-less design (Tab. 1). On the other hand, new opportunities arise once we shift focus away from "avoiding throttling". In this section, we answer three questions: *Why not simply replace C.LIMITS with C.REQUEST? Without C.LIMIT to control CPU usage of containers, what does the latency-resource tradeoff look like? Finally, is all this effort in redesigning worth it?*

Current autoscaling models break with C.REQUEST: Autoscaling systems heavily used C.LIM because C.LIM provides an useful invariant : *for any container C, the actual CPU utilization (U) is less than the C.LIMIT (L) at all times*. With C.LIM, as U approaches L, the latency degrades due to throttling. Thus autoscaling can be conveniently formulated as "minimizing the slack ($L - U$) while keeping the latency under the SLO". L was the knob used to tradeoff CPU allocation and application latency. Allocation minimization algorithms were designed, assuming the presence of C.LIMITS.

On removing C.LIM however, this model breaks: U is no longer bound by the C.REQUEST-based allocation R. For instance, in FIRM, the reward function used by Reinforcement Learning (RL) agent is $\alpha(\frac{SLO}{curr\_latency})+(1-\alpha)(\frac{U}{L})$. With C.LIM, $\frac{U}{L} < 1$ always. On replacing C.LIM (L) with C.REQ (R), pod can burst and $\frac{U}{R} > 1$ is possible. Thus the RL will converge to degenerate solution of setting $R = 0$ to maximize the reward (more failure examples in Tab. 1). With C.LIM, throttling dominated the latency vs CPU allocation tradeoff. Impact of node utilization & C.REQ values were secondary (e.g. [72, 81] in Tab. 1 solely optimized C.LIM without modifying C.REQ), but will now come forth and must be actively managed.

The C.LIMITS-less Design Space: Firstly, we need a way to trade off CPU allocation for latency of a pod. There are two such *knobs*. (1) **Overage** ($U - R$) of a pod's utilization compared to its allocated C.REQ, (2) **Net node CPU utilization** (**N**) of the node on which the pod is placed . As shown in Fig. 3b (and §3.4), having a smaller C.REQ on a pod (i.e. a higher overage) will sacrifice latency for resource and vice versa. Similarly, placing the pod on nodes with higher N (i.e. tighter packing) degrades its latency. Fortunately, the latency varies

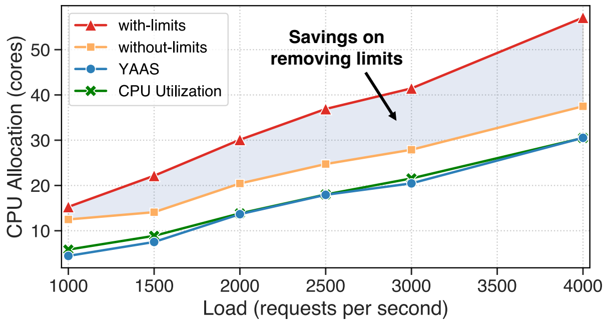


Figure 4: Savings on removing c.lim & with YAAS (HR app)

smoothly on tuning these knobs unlike with c.lim (ref. Fig. 3b(top)). This opens up new design opportunities for adaptive algorithms.

To grasp the rich optimization space, consider the basic problem of finding smallest node that fits two pods C1 and C2. Suppose C1 needs an SLO of 100ms and has an average CPU utilization of 2000m. C2 has a 25ms SLO with 1000m utilization. One simple solution is to fix the overage at 0 i.e. set c.req of each pod equal to their utilization. Then we can search what node utilization (N) can meet the SLO of C2, the stricter of the two (stricter the SLO, lower the node util. has to be). Let's say a 4 core node (4000m) at 75% ($\frac{2000m+1000m}{4000m}$) node utilization meets C2's SLO. However, with that large 25% margin C1's SLO may be met too well (<<100ms). We can optimize further by tuning overages. Redistributing c.req to be say 1500m for both C1 , C2 i.e ($-500m$ & $500m$ overage respectively), we can possibly place them on a smaller 3 core node at 100% node utilization. We built YAAS to show the potential of c.lim-less designs.

A simple 'Yet Another Autoscaler' (YAAS): YAAS optimizes allocation by keeping overage ≥ 0 for all containers at all times. We built YAAS atop K8s's HPA and made simple optimizations. We let the overage grow until SLO is slightly violated, and then set the overage to 0 by allocating more resources (either by increasing c.req or by spawning new replicas). Furthermore, when a new replica is added, the c.req of each replica is reduced proportionally and split equally across all active replicas to maintain overage ≥ 0 and save resources compared to HPA. This is because the utilization U is split between the replicas. To reduce the node utilization N, YAAS moves containers across nodes if a container is scheduled on a *congested node*–whose utilization crosses a threshold (T_{cong}). Fig.5 shows the full operation. Fig.4 shows the CPU resources required to meet SLO (20ms) of the HotelReservation benchmark. *Simply removing c.limits achieves an average 38% savings, and applying the above optimizations with YAAS further increases savings to 51%.*

Several improvements are possible. In doing so, the designers must consider the following: Firstly, the two knobs are not independent. The same overage will result in different latencies on nodes with different N. On a node with a higher N, latency is more sensitive to overage. Secondly, vertical scaling is easier in the c.limit-free domain. Pods are free to use spare CPU cycles on the node as the load fluctuates. With no threat of throttling, vertical scaling no longer requires the autoscaler to micro-manage the CPU allocations during every small load change. Thirdly, combining pod scaling with node scaling becomes necessary when using N as one of the knobs. This differs from the status quo where pod scalers like K8s HPA or those in Tab. 1 work independently of node scalers

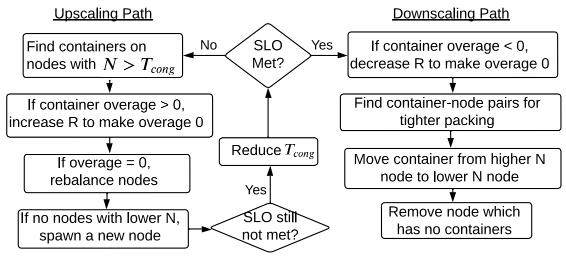

Upscaling Path
Downscaling Path
Find containers on nodes with $N > T_{cong}$
No
SLO Met?
Yes
If container overage < 0, decrease R to make overage 0
If container overage > 0, increase R to make overage 0
Find container-node pairs for tighter packing
If overage = 0, rebalance nodes
Reduce T_{cong}
Move container from higher N node to lower N node
Yes
If no nodes with lower N, spawn a new node
SLO still not met?
Remove node which has no containers

Figure 5: YAAS's scaling policies.

like ClusterAutoscaler [43], Karpenter [52], etc. Finally, safeguards such as hysteresis are needed to ensure the stability of multiple interacting control loops & upscaling-downscaling logic.

4.2 c.lim-free Cloud Billing

c.limit is an essential mechanism for usage-based billing in multi-tenant cloud services [55, 56]. c.lim make billing easy for operators by offloading the responsibility of rightsizing to the users. Users are billed for the resource they ask for. We call this *'resource-based billing'*. Fair c.lim-based billing is an active area of research [65]. Here we specifically ask how to support billing without using c.lim?

Resource-based billing does not work c.limits-free: Consider an obvious approach to bill based on the c.req users specify. This has two issues: (1) the operator can not police the users' pods without c.lim. Automatically adding c.lim (=c.req) can escalate latency/cost for the users (e.g. [17]), (2) It is easy to exploit – user can launch many pods with very small c.req, landing a few in emptier nodes. Once there, the pods burst out using more than they are billed for. Borg reported having this issue which they fixed with placement heuristics [44]. Alternatively, we could bill by the actual CPU.Util of the pods. But this can lead to large increase in user bills. Not all CPU.Util contributes to useful performance. For instance, containers that rely on spinning (e.g., based on Erlang [2, 48] or Go [30]) can exhibit high CPU.Util without proportional performance gains. Further, contentions such as the cache can increases pod CPU.Util (processes spend longer on the CPU due to cache misses). Users will end up paying more for worse performance.

Hence, we propose **Performance Based Billing**. Users only specify what performance they desire & a maximum cost limit - e.g: maintain SLO $\leq$ 100ms with net cost under $XXX. Operators are free to manage resources as needed. With controls over node utilizations, pod placements, observability etc, operators have significantly more knobs to control the latency of applications. We show how such a scheme can be built on top of a c.lim-free autoscaler using YAAS.

A sketch of c.lim-free Billing: Recall that YAAS (§4.1) increases c.req conservatively (overage $\geq$ 0) – only when the SLO is about to be breached – and decreases it eagerly (Fig. 5). Thus c.req set by YAAS is indicative of the minimum c.req needed to meet the SLO. We can use sum of c.req set by YAAS to bill the user. This however is susceptible to an adversarial attack: the user can set a very strict SLO. In an attempt to meet the SLO, YAAS will move the pod to lightly-loaded nodes (low T_{cong}) thus enjoying superior latency without paying for it. To plug this gap, pricing should also reflect the node utilization N the pod needs to meet SLO. The lower the node utilization that is needed, the higher the price per c.req.

5 Debunking Myths & Valid C.LIM Usecases

Despite the evidences & opportunities presented thus far, researchers and SREs alike expressed skepticism about abandoning C.LIM. We address them here. Use of C.LIM is driven by erroneous beliefs that C.LIM is essential for operational and safety purposes. In reality, C.REQUEST is largely sufficient. C.LIM must be used judiciously only in specific scenarios. We elaborate.

Myth 1: Multi-tenancy: *C.LIMITS is needed in multi-tenant clusters to ensure applications get their assigned allocation*
Reality: False. C.REQUEST is sufficient (§3.4)
Valid Usecase: Benchmarking: Latency with C.LIM is a worst-case estimate of application's performance. Such benchmarking can help capacity planning. However,C.LIMITS should be limited to offline profiling and not be used in production.

Myth 2: Performance Isolation: *C.LIMITS provides performance isolation. It is a protection against noisy neighbors.*
Reality: False, with Caveats. Co-located containers can affect each other through CPU resources in two ways: 1) High node utilization: C.REQ already provides a level of performance isolation by preventing bursting containers from affecting neighbors. C.LIM has no additional benefit. 2) Contention on micro-architectural resources, primarily cache & memory bandwidth (MB). Effectively managing these interferences requires hardware support like Intel's CAT, MBA [50, 51]. C.LIM does not provide these.
Valid Usecase: Background (BG) Jobs: In the absence of CAT/MBA, C.LIM is often useful to control non-latency sensitive, throughput-oriented jobs like ML [18], or momentarily bursty events like GC processes which exhaust the MB/cache [67]. First step is to set the lowest C.REQ (1m) for the BG jobs or use lower scheduling priority settings like SCHED_IDLE [3]. But it may not be enough [71], and we may need to explicitly limit the BG jobs using C.LIM.
Valid Usecase: CPU Pinning: Some workloads benefit from being pinned to CPUs [71, 76]. C.LIM has recently been modified to allow pinning if C.LIM is an integer [46]. But pinning has a negative impact on a highly threaded application [66]. Also, rounding to an integer can accumulate into significant wastage (e.g: as noted in Borg [44])

Myth 3: Safety: *Runaway/buggy pods may consume excessive CPU. C.LIMITS is a must to contain such behavior & control cloud bill.*
Reality: False. C.LIM can not be configured to prevent runaway pods. C.LIM has no way to distinguish 'useful' CPU consumption (e.g., due to increased workload) from a bug-induced one. Two scenarios: if autoscaling is enabled on that pod, then as CPU.UTIL approaches C.LIM, the autoscaler will scale up the pod, irrespective of whether the container is buggy or not. If autoscaler is not enabled, then the impact is limited to one node, and C.REQ protects co-located containers. Further, to limit the cloud bill, a *full-application level limit* suffices [45]. Applying C.LIM on every pod is overkill.

Myth 4: Cluster Stability: *Critical system components will become unresponsive at very high node utilization without C.LIMITS.*
Reality: False. Small amount of C.REQ must be reserved on each node for system components [57] C.REQ will guarantee the system pods gets CPU time no matter what the node utilization is.
Other Valid Use Cases: Power capping [70]. Sometimes, throttling the containers is intentional in using C.LIMITS. E.g., we spoke to a company that hosts Minecraft servers, and they want their free-tier users to see their CPU utilization and throttle when they hit the quota (thus prompting them to upgrade).

6 Conclusion

In this paper, we showed that the use of C.LIM can be counterproductive and we recommend removing C.LIM from CPU resource management for latency-sensitive applications. We can and should base new autoscaling and billing systems on C.REQUEST only. The design space here is rich, and even simple policies have potential for large resource savings, while being more reliable and predictable. C.LIM must only be used in specific, unavoidable scenarios.

Acknowledgments:

This research was funded in part by a grant from the IBM-Illinois Discovery Accelerator Institute. We also thank Cheuk Lam at IBM and Thrivikraman Varadharajan, Tianyin Xu, Aishwarya Ganesan and Ram Alagappan at UIUC for useful discussions and suggestions. We also thank our reviewers for their thoughtful comments.

References

[1] Prominent features in linux 3.2. https://kernelnewbies.org/Linux_3.2/, 2012.
[2] The curious case of beam cpu usage. https://stressgrid.com/blog/beam_cpu_usage/, 2019.
[3] Fixing sched_idle. https://lwn.net/Articles/805317/, 2019.
[4] Optimizing kubernetes resource requests/limits for cost-efficiency and latency. https://www.youtube.com/watch?v=eBChCFD9hfs&ab_channel=HighLoadChannel, 2019.
[5] Tim hockin, top-level kubernetes maintainer, google. https://x.com/thockin/status/1134193838841401345?lang=en, 2019.
[6] Artifact of "firm: An intelligent fine-grained resource management framework for slo-oriented microservices". https://gitlab.engr.illinois.edu/DEPEND/firm/-/blob/master/actions.py?ref_type=heads, 2020.
[7] Kubernetes: Make your services faster by removing cpu limits, hackernews discussion. https://news.ycombinator.com/item?id=24351566, 2020.
[8] Kubernetes: Pod quality of service classes. https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/, 2020.
[9] 10 trends in real-world container use. https://www.datadoghq.com/container-report-2021/, 2021.
[10] 73,000 pods a day misadventures in multi-tenant. https://static.sched.com/hosted_files/kccncna2022/d5/misadventure_Multitenant.pdf, 2022.
[11] For the Love of God, Stop Using CPU Limits on Kubernetes. https://home.robusta.dev/blog/stop-using-cpu-limits, 2022.
[12] Why You Should Keep Using CPU Limits on Kubernetes. https://dnastacio.medium.com/why-you-should-keep-using-cpu-limits-on-kubernetes-60c4e50dfc61, 2022.
[13] 10 insights on real-world container use. https://www.datadoghq.com/container-report/, 2023.
[14] Artifact of "cilantro: Performance-aware resource allocation for general objectives via online feedback". https://github.com/romilbhardwaj/cilantro/blob/main/experiments/microservices/starters/hotel-res/hotel-res-core/frontend/frontend-deployment.yaml, 2023.
[15] Control groups (cgroups) for limiting resource usage on linux. https://www.ibm.com/docs/en/spectrum-symphony/7.3.0?topic=limits-control-groups-cgroups-limiting-resource-usage-linux, 2023.
[16] An eevdf cpu scheduler for linux. https://lwn.net/Articles/925371/, 2023.
[17] First experiences with GKE autopilot in production - question about requests/limits inefficiency. https://www.reddit.com/r/googlecloud/comments/r2pzpz/first_experiences_with_gke_autopilot_in/, 2023.
[18] How to optimize kubernetes performance for machine learning workloads. https://www.ivinco.com/blog/how-to-optimize-kubernetes-performance-for-machine-learning-workloads?utm_source=chatgpt.com, 2023.
[19] Introduction to control groups. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01, 2023.
[20] Kubernetes cpu throttling: The silent killer of response time – and what to do about it, ibm. https://community.ibm.com/community/user/aiops/blogs/dina-henderson/2022/06/29/kubernetes-cpu-throttling-the-silent-killer-of-res, 2023.
[21] Kubernetes CPU throttling: The silent killer of response time. https://www.ibm.com/blog/kubernetes-cpu-throttling-the-silent-killer-of-response-time/, 2023.

[22] Maximizing reliability, minimizing costs: Right-sizing Kubernetes workloads. https://cloud.google.com/blog/products/containers-kubernetes/proactive-kubernetes-workload-management, 2023.
[23] Practical tips for rightsizing your Kubernetes workloads. https://www.datadoghq.com/blog/rightsize-kubernetes-workloads/, 2023.
[24] Artifact of "autothrottle: A practical bi-level approach to resource management for slo-targeted microservices". https://github.com/microsoft/autothrottle/blob/main/worker-daemon.py, 2024.
[25] Best practices for achieving isolation in kubernetes multi-tenant environments. https://www.loft.sh/blog/best-practices-for-achieving-isolation-in-kubernetes-multi-tenant-environments, 2024.
[26] Best practices for multi-tenant environments, oracle. https://docs.oracle.com/en-us/iaas/Content/ContEng/Tasks/contengbestpractices_topic-Multi-Tenancy-best-practices.htm, 2024.
[27] Bitnami resource templates. https://github.com/bitnami/charts/blob/main/bitnami/common/templates/_resources.tpl, 2024.
[28] Can cpu throttling results in oomkilled states in k8s? https://stackoverflow.com/questions/76153774/can-cpu-throttling-results-in-oomkilled-states-in-k8s, 2024.
[29] Containerizing java applications. https://learn.microsoft.com/en-us/azure/developer/java/containers/kubernetes, 2024.
[30] Elixir vs go vs node. https://stressgrid.com/blog/benchmarking_go_vs_node_vs_elixir/, 2024.
[31] Erlang quota awareness. https://www.erlang.org/blog/otp-23-highlights/#take-cpu-quotas-into-account, 2024.
[32] How to prevent a run away pod or container from using all resources on the node. https://platform9.com/kb/kubernetes/how-to-prevent-a-run-away-pod-or-container-from-using-all-resou, 2024.
[33] Kubernetes liveness probes and cpu limit risks self-reinforcing crashloopbackoff. https://heiioncall.com/blog/kubernetes-liveness-probes-and-cpu-limits-risks-self-reinforcing-crashloopbackoff, 2024.
[34] Kubernetes: Production-grade container orchestration. https://kubernetes.io/, 2024.
[35] Kubernetes scheduler. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/, 2024.
[36] Linux cpu management. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/resource_management_guide/sec-cpu, 2024.
[37] Managing compute resources with openshift/kubernetes, redhat. https://www.redhat.com/en/blog/managing-compute-resources-openshiftkubernetes, 2024.
[38] Multi-tenancy, google cloud. https://cloud.google.com/kubernetes-engine/docs/best-practices/enterprise-multitenancy, 2024.
[39] Multi-tenancy, kubernetes. https://kubernetes.io/docs/concepts/security/multi-tenancy/, 2024.
[40] Rabbitmq runtime schedulers. https://www.rabbitmq.com/docs/runtime#scheduling, 2024.
[41] Service level objectives : Google sre handbook. https://sre.google/sre-book/service-level-objectives/, 2024.
[42] The surprising economics of horizontal pod autoscaling tuning, gke. https://cloud.google.com/blog/products/containers-kubernetes/tuning-the-kubernetes-hpa-in-gke, 2024.
[43] About cluster autoscaling. https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler, 2025.
[44] Cluster management at google with borg. https://www.youtube.com/watch?v=0W49z8hVn0k, 2025.
[45] Configure memory and cpu quotas for a namespace. https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/quota-memory-cpu-namespace/, 2025.
[46] Control cpu management policies on the node. https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/, 2025.
[47] Datadog cpu request, limit deep dive. https://www.datadoghq.com/blog/kubernetes-cpu-requests-limits/, 2025.
[48] Erlang performance issues with limit. https://elixirforum.com/t/performance-issue-when-running-in-kubernetes/31239/11, 2025.
[49] Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2025.
[50] Introduction to cache allocation technology. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, 2025.
[51] Introduction to memory bandwidth allocation. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html, 2025.
[52] Karpenter.sh. https://karpenter.sh/, 2025.
[53] Kubernetes pod quality of service classes. https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/, 2025.
[54] Kubernetes scheduler. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/, 2025.
[55] Managing the noisy neighbor problem in kubernetes. https://www.spectrocloud.com/blog/managing-the-noisy-neighbor-problem-in-kubernetes-multi-tenancy?utm_source=chatgpt.com, 2025.
[56] Multi-tenancy best practices. https://docs.oracle.com/en-us/iaas/Content/ContEng/Tasks/contengbestpractices_topic-Multi-Tenancy-best-practices.htm, 2025.
[57] Reserve compute resources for system daemons. https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/, 2025.
[58] Resource management for pods and containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/, 2025.
[59] Vertical pod autoscaling. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler, 2025.
[60] Why are you setting cpu limits ? https://www.reddit.com/r/kubernetes/comments/utvr4e/why_are_you_setting_cpu_limits/, 2025.
[61] Natalia Angulo and Carlos Sanchez. Lessons learned migrating an existing product to a multi tenant cloud native environment - natalia angulo & carlos sanchez, adobe, kubecon north america 2023. https://youtu.be/RmZh67vSjNY?si=koTvH6aWLbQszYfU&t=1361, 2023.
[62] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, Renton, WA, April 2018. USENIX Association.
[63] Ataollah Fatahi Baarzi and George Kesidis. SHOWAR: Right-Sizing And Efficient Scheduling of Microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 427–441, New York, NY, USA, 2021. Association for Computing Machinery.
[64] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Cilantro: Performance-Aware resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 623–643, Boston, MA, July 2023. USENIX Association.
[65] Tingjia Cao, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Tyler Caraza-Harter. Making serverless Pay-For-Use a reality with leopard. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 189–204, Philadelphia, PA, April 2025. USENIX Association.
[66] Shuang Chen, Shay GalOn, Christina Delimitrou, Srilatha Manne, and Jose F Martinez. Workload characterization of interactive cloud services on big and small server platforms. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 125–134. IEEE, 2017.
[67] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.
[68] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.
[69] Md Rajib Hossen, Mohammad A. Islam, and Kishwar Ahmed. Practical Efficient Microservice Autoscaling with QoS Assurance. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22, page 240–252, New York, NY, USA, 2022. Association for Computing Machinery.
[70] Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sreekumar Kodakara, David Lo, and Parthasarathy Ranganathan. Thunderbolt: throughput-optimized, quality-of-service-aware power capping at scale. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.
[71] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 450–462, New York, NY, USA, 2015. Association for Computing Machinery.
[72] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.
[73] Benjamin Reidys, Pantea Zardoshti, Íñigo Goiri, Celine Irvene, Daniel S. Berger, Haoran Ma, Kapil Arya, Eli Cortez, Taylor Stark, Eugene Bak, Mehmet Iyigun, Stanko Novakovic, Lisa Hsu, Karel Trueba, Abhisek Pan, Chetan Bansal, Saravan Rajmohan, Jian Huang, and Ricardo Bianchini. Coach: Exploiting temporal patterns for all-resource oversubscription in cloud platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 164–181, New York, NY, USA, 2025. Association for Computing Machinery.

[74] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[75] Vighnesh Sachidananda and Anirudh Sivaraman. Erlang: Application-aware autoscaling for cloud microservices. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 888–923, New York, NY, USA, 2024. Association for Computing Machinery.

[76] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Mxfaas: Resource sharing in serverless environments for parallelism and efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[77] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, et al. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803, 2020.

[78] Paul Turner, Bharata B Rao, and Nikhil Rao. Cpu bandwidth control for cfs. In *Proceedings of the Linux Symposium*, pages 245–254, 2010.

[79] Paul Turner, Bharata B Rao, and Nikhil Rao. CPU bandwidth control for CFS. In *Linux Symposium*, volume 10, pages 245–254. Citeseer, 2010.

[80] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[81] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y. Yan. Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices. In *NSDI (USENIX Symposium on Networked Systems Design and Implementation)*. USENIX, April 2024.

[82] Yanqi Zhang, Zhuangzhuang Zhou, Sameh Elnikety, and Christina Delimitrou. Analytically-Driven Resource Management for Cloud-Native Microservices. *arXiv preprint arXiv:2401.02920*, 2024.

[83] Yuqiu Zhang, Tongkun Zhang, Gengrui Zhang, and Hans-Arno Jacobsen. Lifting the fog of uncertainties: Dynamic resource orchestration for the containerized cloud. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 48–64, New York, NY, USA, 2023. Association for Computing Machinery.