PerfectScale by doit

The Kubernetes Optimization Playbook:

Cost, Reliability, and Scale Without the Guesswork

You've deployed your Kubernetes cluster, and your app is running.

But now comes the hard part: keeping it efficient, without overspending.

Cut too much, and you risk outages. Overprovision, and you pay for resources you don't need.

In this guide, we'll break down where K8s costs come from, common mistakes that waste budget, and a simple 6-step framework to help you optimize without downtime or guesswork.

BONUS!
Production-Ready Optimization Assessment

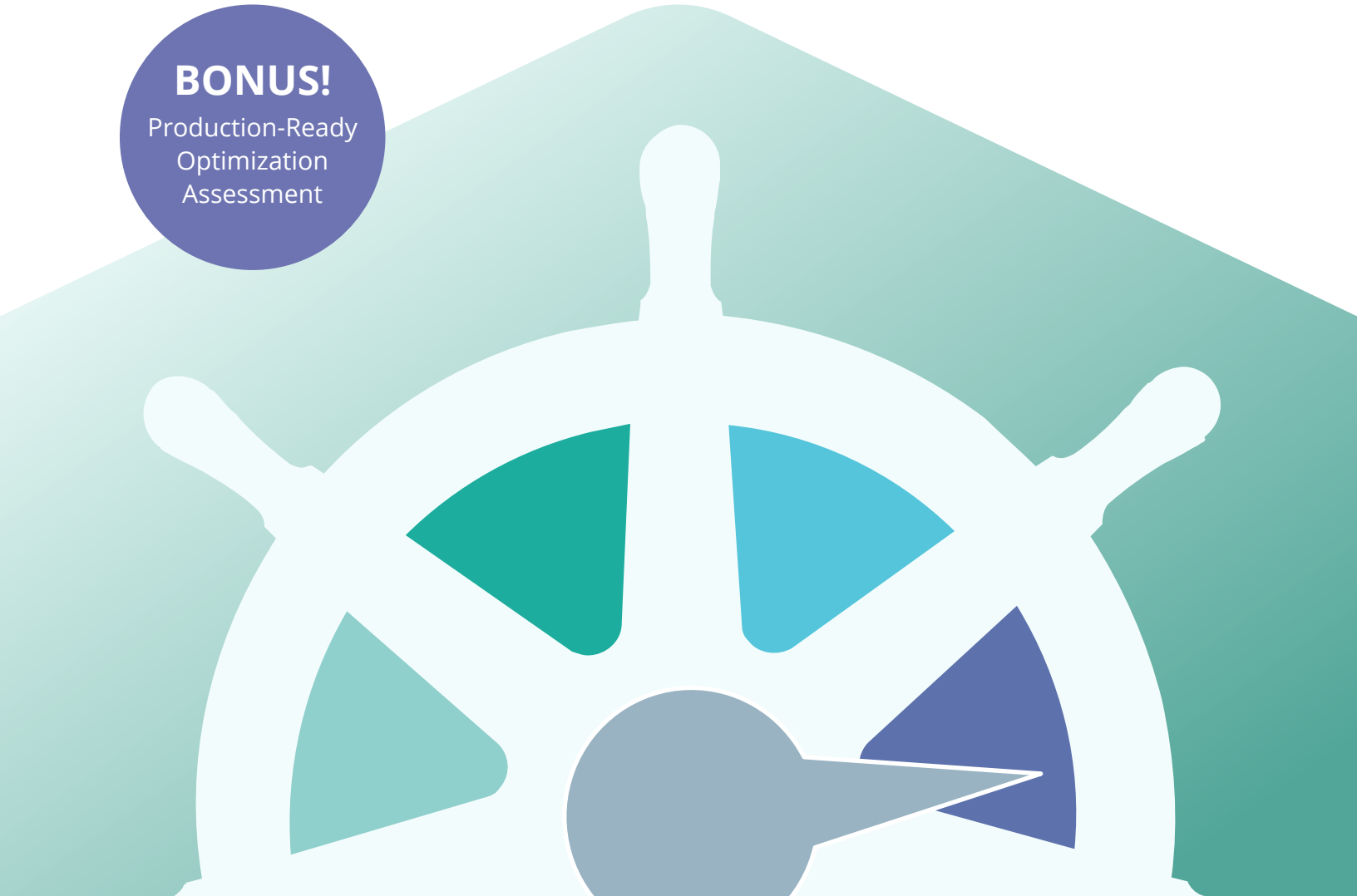


Table of Contents

Understanding the Kubernetes Cost Components

Kubernetes cost management starts with understanding what you're paying for—and why those costs can be unpredictable.

In traditional infrastructure, you buy or rent a fixed number of servers. In Kubernetes, you define what your applications need, and the system tries to meet those requests. This abstraction is powerful, but it also creates a disconnect between what you allocate and what you actually pay for.

Here are the four major cost drivers:

Compute Resources

This is the largest and most direct cost in most Kubernetes environments. Each container defines CPU and memory requests and limits, which Kubernetes uses to schedule and allocate resources across the cluster. These values determine how much compute capacity is reserved, even if it's never used.

- Overprovisioning leads to significant waste, as unused CPU and RAM are still billed.
- Underprovisioning leads to throttling, latency, or even downtime.

Compute costs scale quickly across hundreds or thousands of pods and nodes, making this the highest-leverage area for optimization.

Storage Costs

Storage costs come from Persistent Volumes (PVs) and the StorageClasses backing them, like standard HDD or high-performance SSD.

- Block storage (e.g., for databases) is fast but expensive.
- Object storage (like Amazon S3 or Google Cloud Storage) is cheaper, easier to scale, and ideal for logs, backups, and static assets. While object storage is cost-effective, mounting it into pods with CSI drivers introduces performance overhead and isn't ideal for latency-sensitive workloads.

Storage costs also grow based on retention policies, replication, and snapshotting, which are often overlooked in early configurations.

Networking Costs

Network charges in Kubernetes include:

- **Data transfer between pods and services**, especially across zones or regions.
- **LoadBalancer services**, which provision and maintain expensive cloud-managed endpoints.
- **Ingress traffic** is often routed through dedicated ingress controllers or third-party gateways.

Nodes also have limits on IP addresses and network interfaces. When those limits are hit, the cluster may spin up more nodes just to handle networking, which quietly drives up costs even further.

Additional Services

Kubernetes clusters often include a growing ecosystem of tools and agents that run alongside your core workloads:

- **Monitoring and observability tools** (e.g., Prometheus, Datadog, Grafana agents)
- **Logging systems** (e.g., Fluentd, Loki, ELK stack)
- **Security, CI/CD, or service mesh components** (e.g., Istio, Argo CD, Linkerd)

These services consume CPU, memory, and storage just like your app. Third-party platforms may also charge based on metrics volume, data retention, or number of agents, adding hidden costs to each deployed workload.

Why 51¢ of Every Kubernetes Dollar Goes to Waste

Storage, networking, and observability services all contribute to Kubernetes costs…but compute is where the biggest waste lives. CPU, memory, and GPU resources account for the bulk of spend, and they're also the most directly controllable.

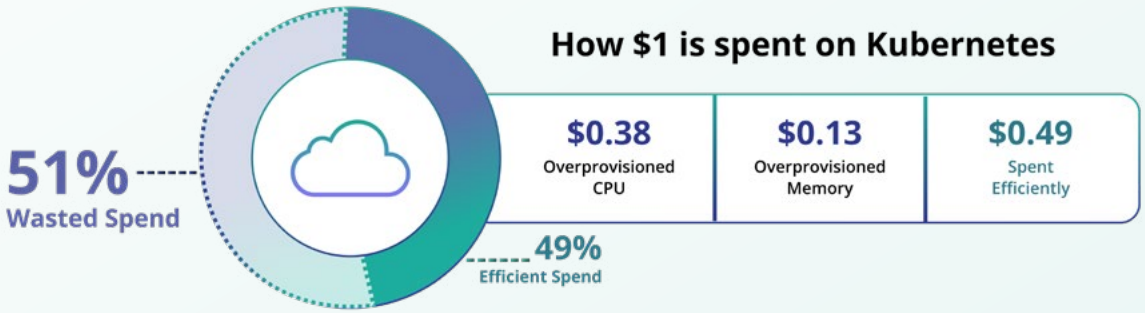
It is common for teams to overprovision compute to play it safe. The logic seems sound: higher spending must mean higher reliability. But in practice, even with generous buffers, teams still run into latency, performance issues, and scheduling failures.

The problem isn't a lack of effort or skill. It's a lack of visibility.

Even experienced DevOps teams are forced to guess. Without real-time insight into how workloads behave, developers rely on copy-paste defaults, outdated manifests, or overcautious estimates. That guesswork leads to a costly mix of inefficiency, fragility, and bloat.

The result?

According to the Kubernetes Optimization Trends Report 2024 by PerfectScale**, 25% of Kubernetes clusters are at risk, and 51¢ of every $1 spent on Kubernetes is wasted.**



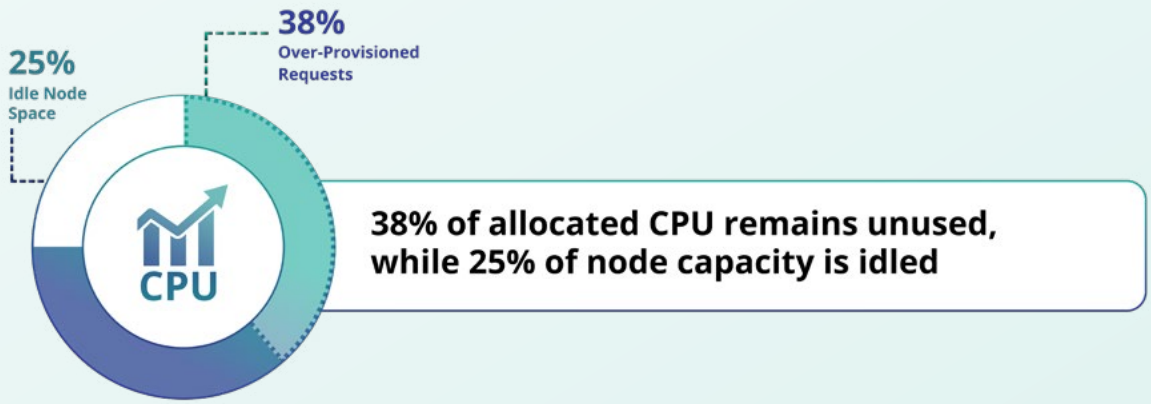
Wasted vs Idle Resources

Even experienced teams overspend on Kubernetes. Not because they're careless, but because they're cautious.

Here's why it happens:

- **Fear of downtime:** Overprovisioning feels safer than risking a crash.
- **Limited infrastructure expertise:** Developers rely on default settings or outdated manifests.
- **Lack of visibility:** Without clear usage data, teams can't rightsize confidently.
- **Risk aversion:** Manual tuning feels safer than trusting autoscalers or telemetry.

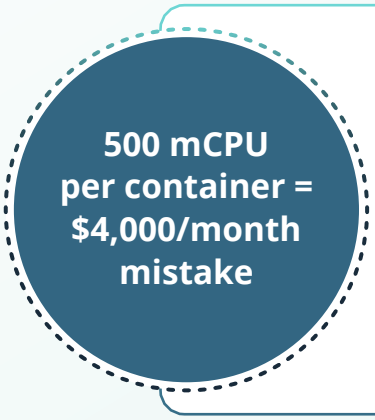
The result? Massive amounts of wasted and idle resources—often in the same cluster.



Let's break down the difference between waste and overprovisioning:

Wasted Resources

Wasted resources are allocated but unused. They show up when CPU or memory requests are set too high "just in case." At scale, this adds up to thousands (sometimes millions) of dollars in unused capacity.



PerfectScale Customer Story

One PerfectScale customer had a workload with 2 containers per pod and scaled it to 100 replicas. Each container was overprovisioned by 500 mCPU beyond what it actually used.

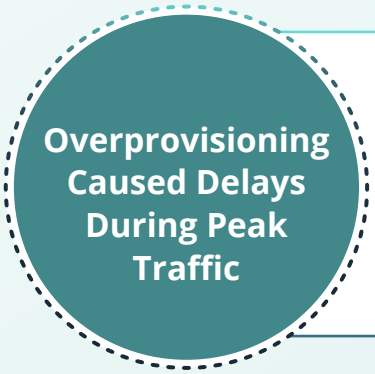
- **500 mCPU × 2 containers × 100 replicas = 100 vCPUs of unused compute**
- **That's ~$120/day, or $44,000/year in waste, for just one workload**

Overprovisioning

It's hard to guess the right CPU and memory requests for a workload. So, engineers play it safe and go high.

Now imagine 200 pods, each over-requesting by 100 MB of memory. That's 20 GB of wasted RAM across the cluster, leading to:

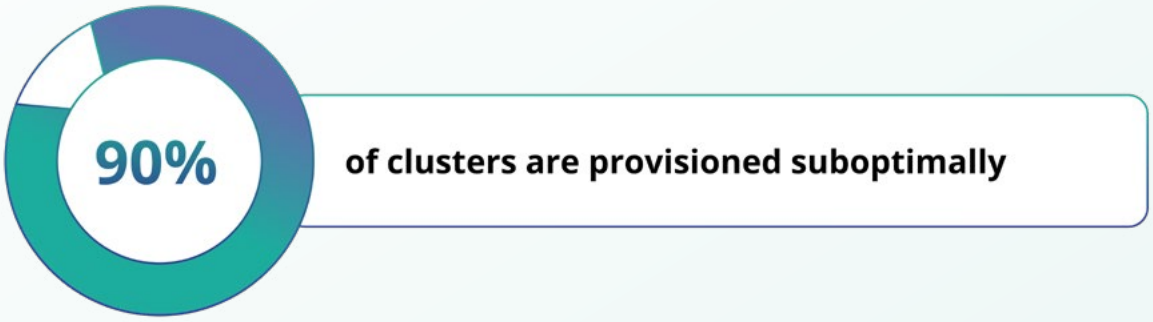
- Higher node counts
- Slower autoscalers
- Inefficient scheduling



PerfectScale Customer Story

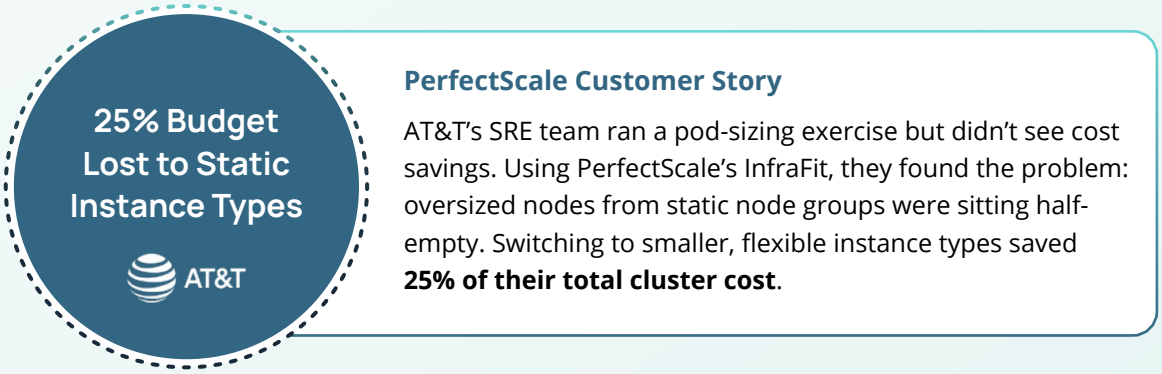
A large e-commerce company set pod limits to 1 CPU each, even though usage was much lower. During a traffic spike, KEDA scaled the pods up to 250, hitting the regional CPU quota. The cluster couldn't bring up new nodes, pods sat pending, and **orders were delayed**.

Idle Resources



These are provisioned but unallocated resources sitting unused at the node level. Sometimes they exist to absorb bursts. But often they result from:

- Poor bin-packing
- Suboptimal autoscaler configs
- Using static or oversized instance types

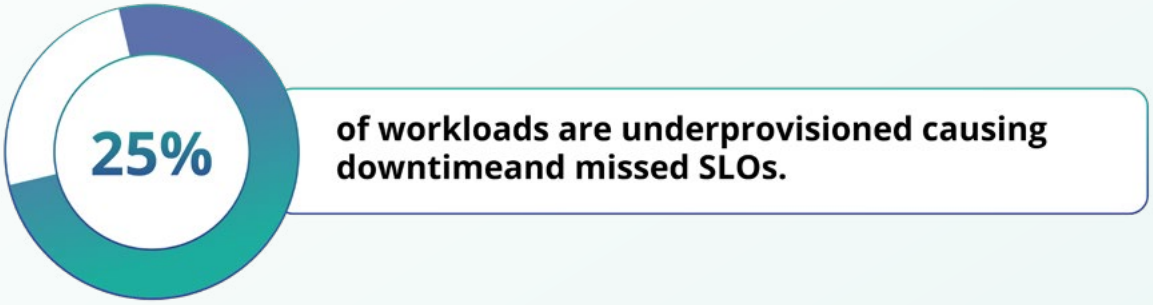


Underprovisioning

Trying to cut costs too aggressively can backfire. When workloads don't get the resources they need, the result is:

- Increased latency
- OOMKills and evictions
- Degraded performance
- Production outages

PerfectScale data shows that 25% of workloads face performance risks due to underprovisioning or configuration errors. These issues are often invisible until production breaks.



Common causes:

- Developer misconfigurations (YAML typos, copy-paste limits)
- Shifting load patterns (traffic spikes, user growth)
- Over-optimization (cutting too close to the edge)

Scheduling Matters (A Lot)

Right-sizing your containers and nodes is essential for cost-efficiency, performance, and reliability. But how workloads are scheduled onto nodes can make or break your optimization efforts.

Kubernetes scheduling is highly flexible. You can control where and how pods are placed using parameters like node selectors, affinity rules, and disruption budgets. This flexibility is powerful, but misconfigurations can inadvertently boost your cluster costs.

The Costly Scheduling Mistakes

Here are a couple of examples of how misconfigured scheduling can generate waste:

The selfish pod

A pod may be perfectly right-sized for CPU and memory, running on a well-configured node, but a misapplied constraint (like an unnecessary node taint or anti-affinity rule) can block other pods from being scheduled to the same node. The result: idle node resources and pods waiting longer before they can run, which can also delay services.

The dog in the manger

Node Taints and tolerations are useful for protecting expensive resources, like GPUs or premium storage. But a misconfigured taint–tolerant pair can let pods that don't need those resources occupy those nodes, while the workloads that do need them sit pending. This can trigger your autoscaler to add more costly nodes, driving up expenses and delaying critical work.

Scheduling Control	Description
nodeSelector	Simple key-value label matching to constrain pods to specific nodes
Node Affinity	More expressive node selection with required/preferred rules and operators
Pod Affinity	Schedule pods near other pods based on labels and topology
Pod Anti-Affinity	Schedule pods away from other pods to spread workloads
Taints & Tolerations	Repel pods from nodes unless they have matching tolerations
Priority Classes	Assign scheduling priority levels to pods for preemption decisions
Pod Disruption Budgets (PDBs)	Limit voluntary disruptions during maintenance/scaling operations
Topology Spread Constraints	Evenly distribute pods across failure domains (zones, nodes, etc.)

Crafting the Optimal Node Group

Kubernetes orchestrates application containers, but those containers still need to run on nodes—virtual or physical machines with defined sizes and prices. The size and cost of your nodes directly affect your cluster's performance, reliability, and budget.

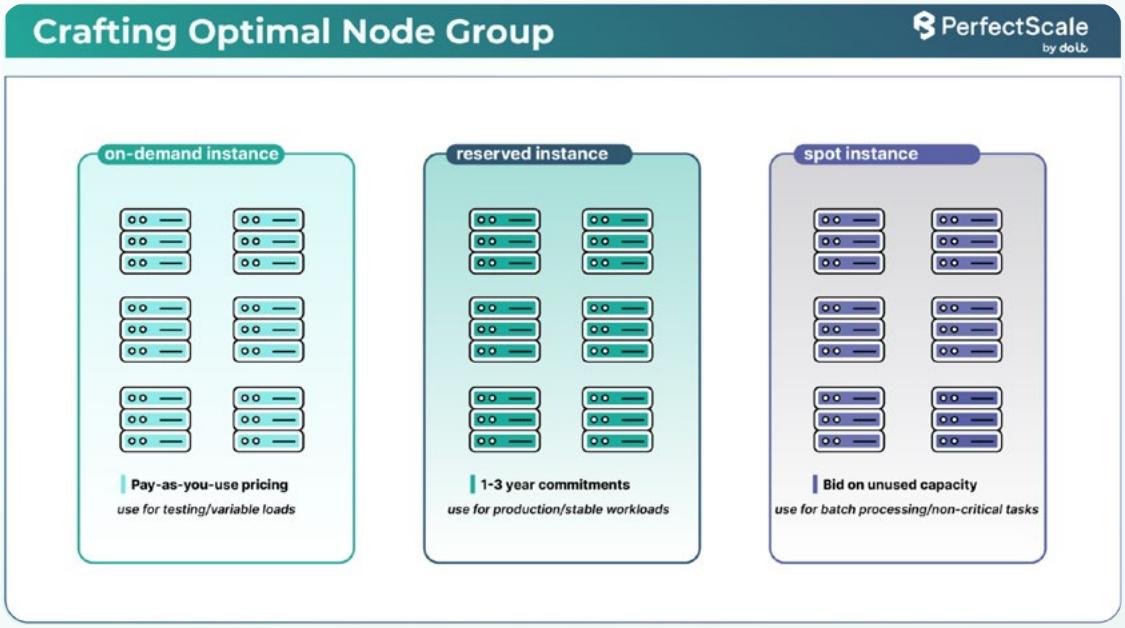
Choosing the right node types is rarely simple. Major cloud providers offer hundreds of options, each with different CPU, memory, storage, and networking capabilities. The challenge is finding the mix that delivers the best performance per dollar for your workloads.

This choice is harder with Cluster Autoscaler, where you must predefine node groups in advance. Even with modern autoscalers like Karpenter, the way you configure node pools can significantly influence cost and reliability.

Purchase model options

Cloud providers also offer different pricing models that can be combined for maximum savings:

- **On-demand:** Pay-as-you-go with no commitments. Most flexible but also the most expensive. Best for variable or unpredictable workloads.
- **Reserved instances:** Commit to 1–3 years for 30–70% discounts. Best for steady, predictable workloads.
- **Spot instances:** Use spare capacity for up to 90% savings. Can be interrupted with short notice when demand rises. Best for fault-tolerant or non-critical workloads.



Picking nodes requires aligning node choice with scheduling constraints and workload needs. For example:

- Use node labels and taints to ensure GPU nodes are only scheduled for GPU workloads.
- Build mixed pools that combine spot, reserved, and on-demand nodes for resilience and cost efficiency.
- Size nodes to balance bin-packing efficiency with scaling speed.

Getting this balance right requires understanding both your workloads and their SLOs. Done well, it reduces waste, prevents performance bottlenecks, and keeps costs predictable.

Quick Rule of Thumb

- Use **on-demand** for testing, staging, or highly variable workloads.
- Use is **reserved** for baseline production workloads that run continuously.
- Use **spot** for batch processing, stateless services, or other fault-tolerant jobs.

Manual vs. Automated K8s Optimization

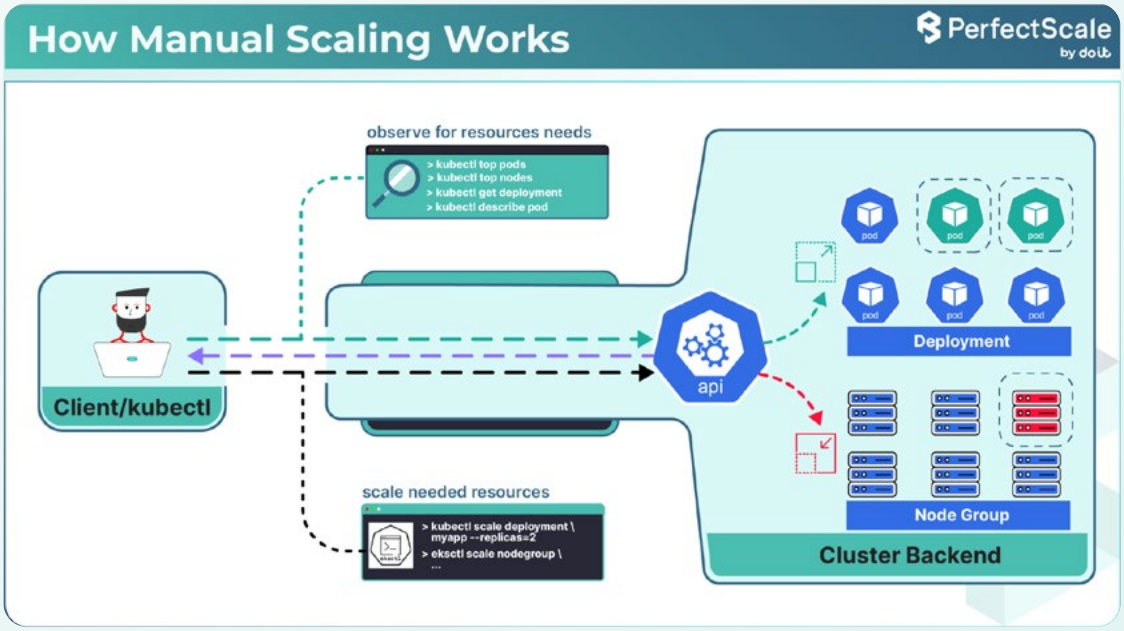
Manual Optimization: What you Can Do (But Probably Shouldn't)

Manual optimization means tuning your Kubernetes environment by hand—adjusting replicas, node pools, resource requests and limits, scheduling rules, and governance policies yourself.

It's technically possible, but it comes with serious drawbacks:

- **Reactive, not proactive:** Teams respond to problems after they occur, often too late or too early to make a meaningful difference.
- **Error-prone:** A single typo in a kubectl scale command or a missed manifest update can trigger overprovisioning, under-replication, or outages.
- **High effort, high cost:** Reviewing dashboards, analyzing metrics, and manually adjusting settings consumes valuable engineering time.
- **Outage risk:** Misjudging workloads or relying on static thresholds often leads to resource misallocation. Without precise, real-time data, finding the true "optimized" line is guesswork.
- **Doesn't scale:** As clusters, workloads, and regions multiply, manual tuning becomes unsustainable.

Manual approaches may work for small dev/test clusters or static, predictable workloads. In production, they're risky, inefficient, and expensive.



Autonomous Optimization: Safer, but Needs Guardrails

Automation is a strategic necessity for meeting cost, reliability, and scalability goals. Automated systems continuously monitor workloads, dynamically adjust compute at both the pod and node level (via HPA, VPA, Cluster Autoscaler, Karpenter), and enforce policies across environments.

Benefits include:

- Reaction times in milliseconds instead of hours
- Workloads meeting performance targets without delay
- Continuous rightsizing based on actual utilization
- Lower cloud costs and reduced latency
- SREs focusing on architecture, not firefighting

Automation scales seamlessly across clusters, clouds, and regions—but it's not foolproof. Misconfigurations or overly aggressive scaling can destabilize workloads.

The YoYo attack

One risk of blind automation is the YoYo attack (Bremler-Barr et al., 2017), where an attacker sends periodic traffic bursts to trigger scaling up and down. This creates an Economic Denial of Sustainability (EDoS)—driving up cloud costs and degrading performance as the cluster oscillates between scale-up and scale-down phases.

Guardrails for safe automation

To prevent instability and runaway costs, build safety checks into your automation:

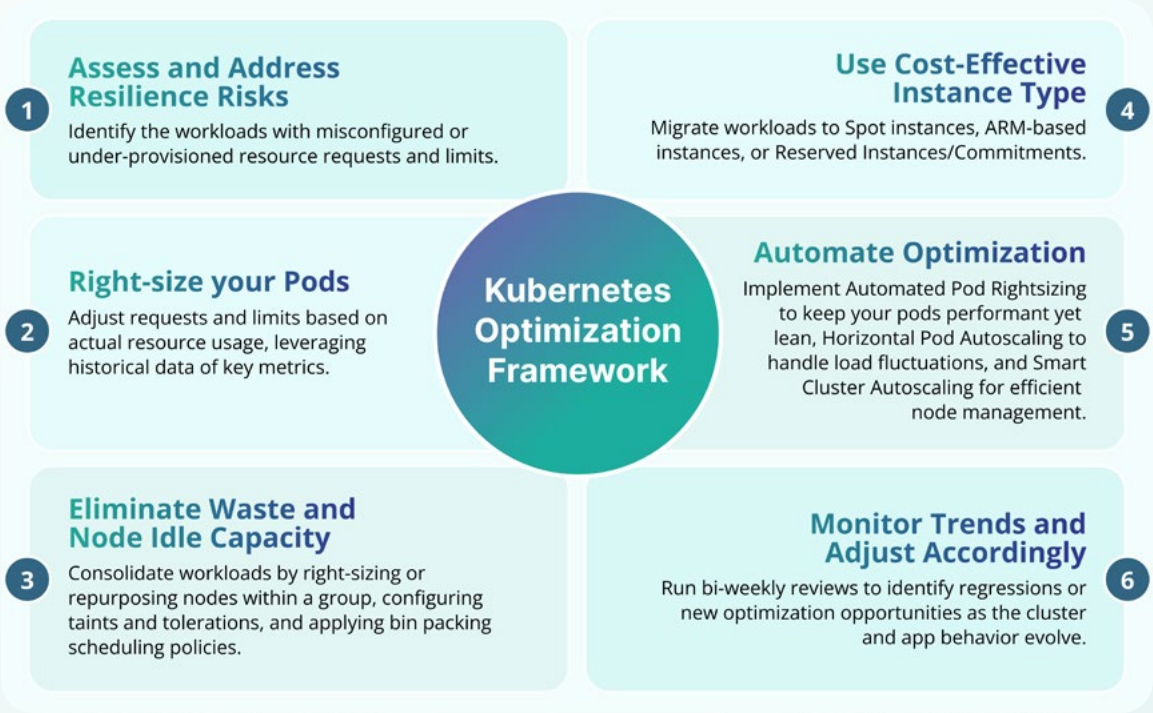
- **Liveness and readiness probes:** Ensure automation only acts on healthy pods. Unstable pods that fail probes won't be counted in scaling decisions.
- **PodDisruptionBudgets (PDBs):** Set limits on how many pods can be taken down at once, preventing aggressive scale-in from causing service gaps.

With these guardrails, automation can safely deliver continuous optimization without pushing your systems into danger.

The 6-step Kubernetes Optimization Framework

Once your Kubernetes environment is live, the real work begins. This is Day-2 Operations—where stability, performance, and cost efficiency must be maintained continuously. Optimization at this stage isn't about one-off fixes. It's about creating a repeatable and proactive process.

Here's a 6-step framework you can apply to cut waste, improve reliability, and keep infrastructure lean and responsive.



1. Patch up the Holes – Assess and Eliminate Resource Risks

Before chasing savings, ensure your cluster is free from misconfigurations that could cause outages or performance issues:

- Unset or underprovisioned memory limits lead to OOM crashes
- Unnecessary CPU limits lead to throttling
- Requests are too low to handle the production load
  Scan for these risks first.

2. Trim the Fat – Rightsize your Pod

Pods often request far more CPU and memory than they use. This inflates node counts and leaves resources idle.

Use historical metrics (Prometheus, CloudWatch, GKE cost tools) to align requests and limits with real usage. PerfectScale's Community Edition can provide this analysis for free.

3. Fill up the Bins – Eliminate Idle Node Waste

Optimized pods make excess node capacity visible. Idle CPU, memory, or GPU may be spread across nodes due to poor bin-packing.

Reduce waste by:

- Resizing or repurposing node groups
- Adjusting taints/tolerations
- Using bin-packing scheduling policies
- Leveraging autoscalers like Karpenter or Cluster Autoscaler with tuned thresholds

4. Pick the Smart Metal – Use Cost-effective Instance Types

After rightsizing your cluster, ensure you're on the right hardware:

- Spot instances for stateless, retry-safe jobs
- ARM-based instances for supported workloads
- Reserved Instances/Commitments for predictable workloads

Use node affinity rules and multi-zone spot pools to balance cost and resilience.

5. Automate Relentlessly – Optimize While you Sleep

Manual optimization doesn't scale. Implement:

- Automated pod rightsizing via vertical pod autoscaler to keep your pods performant and lean.
- Horizontal pod autoscaling for load fluctuations (HPA or KEDA)
- Smart cluster autoscaling for efficient node management

PerfectScale can integrate these for seamless multi-dimensional autoscaling.

6. Optimize in Motion – Monitor Trends and Adjust Accordingly

Optimization is ongoing.

Track these metrics over time:

- Cost per workload
- Utilization-to-request ratios
- Anomalies over time

Use dashboards (PerfectScale, Grafana, FinOps tools) and run bi-weekly reviews to catch regressions and spot new opportunities.

Cut your K8s Spending by 50% with PerfectScale by DoiT

PerfectScale by DoiT helps engineering teams automatically optimize Kubernetes environments, reduce costs while maintaining peak performance, and full visibility.

It works across **EKS, GKE, AKS, KOPS, and private cloud** deployments, integrating seamlessly to give you precise, data-driven control without guesswork.

Rapyd Solved K8s Observability Gaps to Cut K8s Costs by 40%.

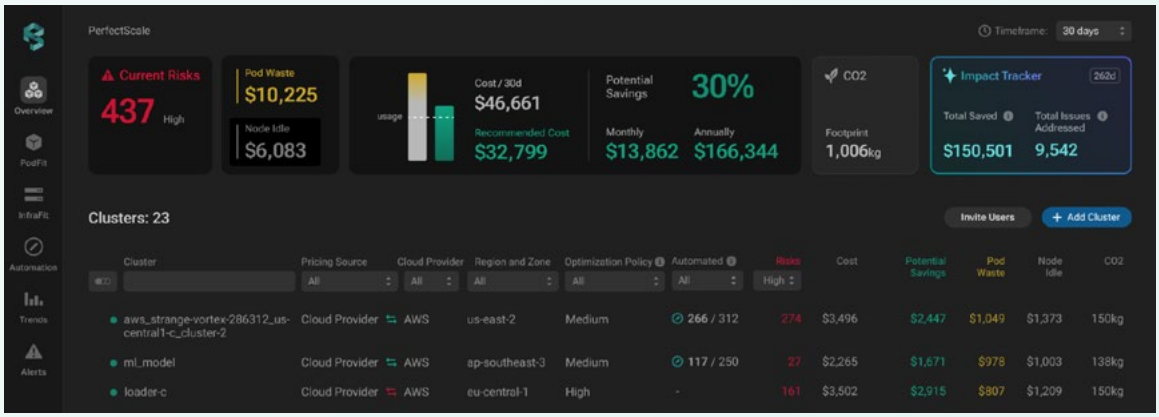
Rapyd

PerfectScale Customer Story

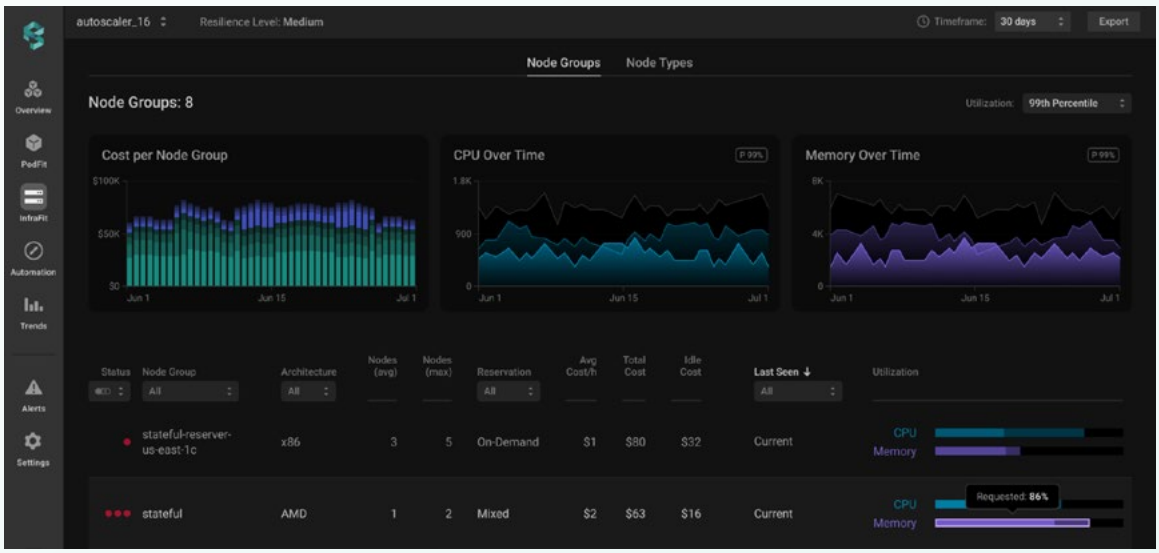
Rapyd, a global fintech leader operating in 100+ countries, migrated from EC2 to EKS and quickly saw the limits of traditional optimization tools. PerfectScale provided the visibility and precision they needed, **reducing cloud costs by up to 40%** while improving performance across 15+ AWS clusters.

With PerfectScale, you can:

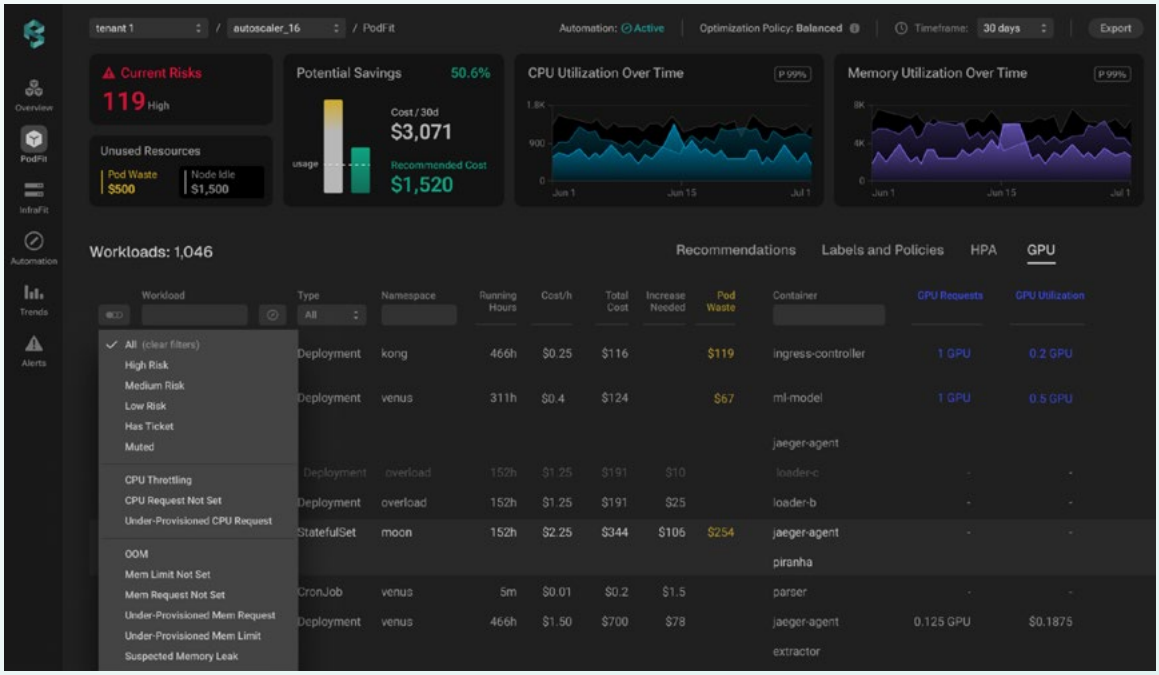
- **Deploy in minutes:** Get a full risk and cost breakdown within 5 minutes using a single Helm command.



- **Get actionable insights:** Receive safe-to-apply recommendations based on actual usage.
- **Autonomously tune resources:** Adjust CPU and memory requests/limits for immediate results.
- **Monitor continuously:** Track cost, risk, and performance across all workloads from one dashboard.



- **Prioritize fixes:** Identify up to 30 resilience risks and address them before they escalate.



Industry leaders like **Rapyd, Paramount Pictures, and Creditas** are already using PerfectScale to run Kubernetes at peak efficiency.

Sign up **or** book a demo **with our technical experts today.**

PerfectScale by doit

The Production-Ready Optimization Assessment

See this assessment to validate that your cluster is not just running, it's optimized, scalable, and cost-efficient.

How It Works:

Each checklist item = 1 point. Total Possible 23 pts.

Score-Based Maturity Levels

Score Range	Maturity Level	Meaning
0–6	Observer	You're monitoring, but there's a lot of waste and risk.
7–13	Proactive	You're applying manual best practices, but not automating.
14–18	Automated	You've implemented tools and automation, but it's not yet integrated into workflows.
19–23	Integrated	Optimization is part of how your team builds and ships software.

Workload Setup Total /7

- [ ] All workloads have appropriate **CPU/memory requests**.
- [ ] All workloads have appropriate **memory limits.**
- [ ] **No workloads have CPU limits** - unless explicitly justified.
- [ ] Requests reflect **actual usage with headroom**, not copy-paste defaults.
- [ ] **No critical workload** is running on **Spot nodes** without a fallback.
- [ ] **Liveness/readiness** probes are defined and tested.
- [ ] PodDisruptionBudgets and scheduling constraints are used **strategically** (not blindly).

Node Group Configuration Total /5

- [ ] **Node groups are segmented** by workload type (e.g., Spot vs On-Demand, GPU workloads).
- [ ] **Node labels and taints** are actively used for smarter scheduling.
- [ ] **Autoscaling node groups** are correctly sized for bin-packing.
- [ ] **Spot instance fallback** strategies are in place (e.g., mixed pools or interruption handlers).
- [ ] Alternative processors are used where applicable (i.e, ARM, Graviton, Ampere, Altra).

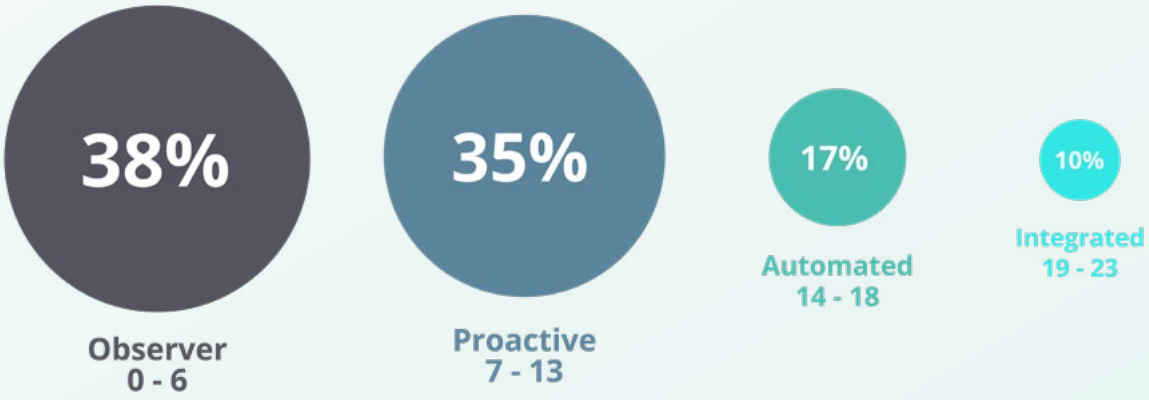
Autoscaling Total /4

- [ ] **HPA** is configured for all scalable stateless services.
- [ ] Cluster Autoscaler/Karpenter/Node Auto Provisioning is tested and tuned.
- [ ] HPA and pod right-sizing do **not** conflict (mutually exclusive when both are enabled).
- [ ] **PerfectScale** is implemented for continuous optimization.

Monitoring & Governance Total /7

- [ ] CPU/memory usage **dashboards** are in place (e.g., PerfectScale, Grafana, Datadog, etc.).
- [ ] **Alerting** is configured for overprovisioning or throttling.
- [ ] **Cost visibility** per namespace or workload is enabled (e.g., PerfectScale).
- [ ] Policies prevent default/no resource configuration from going to prod.
- [ ] CI/CD includes validation for resource configurations.
- [ ] Resource optimization is reviewed **at least monthly**.
- [ ] Automation tool(s) are in place to apply safe changes (e.g., PerfectScale).

See how you compare to industry averages:



Notes/action items

The PerfectScale Vision

We realize that the initial resource allocation is the foundation of all the described autoscaling techniques. It may be the most basic, but it's also the least trivial problem to solve. It requires not only understanding the utilization patterns of your software but also balancing them against the unique business goals and environmental constraints of your organization.

That's why we're providing a full-cycle continuous optimization solution on top of our vertical pod autoscaler. Our platform allows you to optimize Kubernetes workloads continuously, taking into account each application's individual performance, reliability, and horizontal scaling objectives. This happens in full alignment with optimizing the underlying infrastructure, helping you get the most out of your chosen node autoscaling solution.

Defining autoscaling is not a set-and-forget event. Yes, PerfectScale automation takes a lot of guesswork and operational burden off engineers' shoulders. Yet, we must continuously ask if we're on the right path. That's why our platform provides ultimate visibility into the efficiency of your autoscaling configuration and their impact on your cost and reliability goals.

We sincerely believe that continuous delivery of value requires continuous improvement. That's why PerfectScale keeps your Kubernetes clusters both reliable and continuously optimized—so that your engineers can focus on innovation, creativity, and delivering value.

Anton Weiss, Chief Cluster Whisperer, PerfectScale

PerfectScale by doit

perfectscale.io