aws

EKS cost optimization: Scaling, sizing, and savings

AWS Prescriptive Guidance

AWS Prescriptive Guidance: EKS cost optimization: Scaling, sizing, and savings

Table of Contents

EKS cost optimization: Scaling, sizing, and savings

abhay kumar, Akash Kumar, Vinodkumar Mandalapu, and Viyoma Sachdeva, Amazon Web Services

Amazon Elastic Kubernetes Service (Amazon EKS) makes running Kubernetes easier, but without deliberate cost management, compute spend can grow quickly. Idle nodes, over-provisioned pods, extended support charges, and high-cardinality metrics all add up silently. This guide walks through practical strategies to identify and eliminate that waste. From Karpenter consolidation and Graviton adoption to right-sizing pod requests and cutting observability overhead, we cover the full cost surface. Every recommendation includes real kubectl commands, AWS CLI examples, and CloudWatch metrics so you can measure impact immediately. Whether you're running 3 clusters or 300, these patterns apply at any scale.

Code repository: The shell scripts and Kubernetes manifests for this guide are available in the eks-cost-optimization-guide repository on GitHub.

Intended audience

This guide is intended for platform engineers, DevOps engineers, and cloud architects who manage Amazon EKS clusters and are responsible for controlling compute and observability costs.

Business challenge

Organizations running Amazon EKS clusters often face significant and compounding cost challenges that erode cloud ROI:

- **Idle resources** : Clusters running without active application workloads continue consuming compute capacity, accumulating charges with zero business value.
- **Over-provisioning** : Applications configured with excessive CPU and memory requests reserve capacity they never use, preventing efficient bin-packing across nodes.
- **Inefficient autoscaling :** Traditional Cluster Autoscaler configurations with conservative or suboptimal settings respond slowly to demand changes, keeping unnecessary nodes running for extended periods.
- **Legacy architectures** : Workloads remain pinned to x86 instances by default, missing the 20% price-performance improvement available through AWS Graviton processors.

- **Monitoring overhead** : High-cardinality custom metrics, verbose logging, and unfiltered Container Insights generate substantial CloudWatch and observability platform costs that grow linearly with cluster size.
- **Manual management** : Without automation for resource optimization, node lifecycle, and off-hours scheduling, cost inefficiencies persist undetected and unresolved.

These challenges compound across environments. Typical Kubernetes clusters operate at only 20–30% resource utilization while paying for 100% of provisioned capacity, meaning up to 70% of compute spend delivers no workload value. At scale, this translates to tens or hundreds of thousands of dollars in annual cloud waste per organization.

Service limitations

Some AWS services aren't available in all AWS Regions. For Region availability, see the Service endpoints and quotas page in the AWS documentation, and choose the link for the service.

Pricing disclaimer

The pricing examples in this guide are based on prices at the time of publication. Prices are subject to change. Additionally, your costs may vary depending on your AWS Region, AWS service quotas, and other factors related to your cloud environment.

Objectives

Implementing the strategies in this guide helps organizations achieve measurable improvements across cost, reliability, and operations:

- **Reduced infrastructure costs:** Reduce compute spend by up to 30-60% through intelligent node provisioning, Spot and Graviton adoption, right-sizing pod requests, and eliminating extended support charges.
- **Improved resource utilization:** Drive cluster utilization from the typical 20–30% up to 60–70% by eliminating idle capacity, consolidating underutilized nodes, and enforcing resource quotas.
- **Maintained application reliability:** Balance cost savings with availability guarantees using Pod Disruption Budgets (PDBs), topology spread constraints, graceful termination policies, and multi-AZ (Availability Zone) Spot diversification.
- **Operational efficiency:** Reduce manual intervention through automated scaling, Karpenter-driven deprovisioning, off-hours scheduling, and workload placement policies that make cost-optimal decisions without human input.
- **Observability cost control:** Prevent monitoring spend from scaling linearly with cluster growth by reducing metric cardinality, filtering at source, and disabling unnecessary insights in non-production environments.

Prerequisites and limitations

Prerequisites

Before implementing the strategies in this guide, ensure you have:

Technical requirements:

- An active AWS account with Amazon EKS clusters running
- Familiarity with Kubernetes concepts (Pods, Nodes, Deployments, DaemonSets)
- Karpenter v0.33+ or Cluster Autoscaler installed on EKS clusters
- Access to Amazon CloudWatch for metrics and monitoring
- kubectl configured for EKS cluster access

Organizational requirements:

- Executive sponsorship for cost optimization initiatives
- Defined process for testing changes in non-production environments
- Communication plan for workload migrations and disruptions
- Cost allocation and chargeback model (recommended)

Tools and utilities:

- Helm 3.x for installing Kubernetes applications
- AWS CLI version 2.x
- Monitoring tools (CloudWatch Container Insights, Prometheus, or similar)
- Cost visibility tools (AWS Cost Explorer, Kubecost, or similar)

Limitations

Be aware of the following limitations:

Technical limitations:

- Karpenter requires Amazon EKS version 1.23 or later

- Graviton migration requires multi-architecture container images
- Some third-party software may not support ARM64 architecture
- Spot instances may be interrupted with 2-minute notice
- VPA and HPA cannot target the same metrics simultaneously

Operational considerations:

- Node consolidation may cause temporary pod disruptions
- Cluster upgrades require careful planning and testing
- Some optimizations may require application code changes
- Initial implementation requires time investment from engineering teams

Service quotas:

- Amazon EC2 instance limits per region
- Amazon EKS cluster limits (100 clusters per region by default)
- Auto Scaling group limits
- Spot instance capacity availability varies by region and instance type

Idle cluster cleanup: Decommission or consolidation

Before optimizing how your clusters run, eliminate what shouldn't be running at all. Idle clusters and abandoned workloads are pure waste as they consume resources 24/7 while delivering no business value. A cluster is considered idle when it has fewer than 5 application pods running (excluding kube-system, karpenter, monitoring namespaces) and node CPU utilization remains below 10% for 7+ consecutive days.

Identify idle clusters

Find clusters with little to no application workload running, these are candidates for decommissioning or consolidation. For sample commands, see this GitHub repository to identify idle clusters.

CloudWatch: Detect Idle Clusters via Metrics

For organizations managing multiple clusters across regions, CloudWatch metrics provide a centralized, automated way to flag idle clusters without requiring kubectl access to each one. Refer to this GitHub repository for sample commands to identify idle clusters.

Action

If a cluster shows <10% CPU utilization, zero application pods, or no deployments outside kube-system for 7+ days, escalate for decommissioning or consolidate workloads into another cluster. Every idle cluster costs you the full control plane fee plus node compute.

Before deleting an idle cluster, verify these items:

- No application pods running outside kube-system
- No active CronJobs with future schedules
- No external services pointing to this cluster's load balancers
- DNS records updated or removed
- Persistent data backed up or migrated
- IAM roles and policies cleaned up
- CloudWatch log groups archived or deleted

For the complete scripts and manifests, see the 01-idle-cluster-cleanup folder in the code repository.

The cheapest resource is the one you don't run. Eliminating idle clusters delivers immediate savings with zero risk to active applications.

Application cleanup: Removing redundant workloads

Once idle clusters are addressed, clean up abandoned workloads within active clusters. Leftover deployments, failed jobs, and orphaned resources prevent efficient bin-packing and inflate node count.

Identify redundant and abandoned applications

Find deployments, jobs, and services that are no longer needed, leftover from testing, failed rollouts, or decommissioned features. Key signals include:

- Deployments scaled to zero or with zero available replicas
- Pods in CrashLoopBackOff with high restart counts (>10)
- Completed/failed Jobs not cleaned up
- Orphaned Services with empty endpoints (no backing pods)
- Unbound PVCs (storage provisioned but not mounted)
- Stale namespaces with zero pods and no recent events

For scripts, see this GitHub repository to identify redundant and abandoned applications.

Identify stale namespaces

Entire namespaces can become abandoned after projects end or teams reorganize, they accumulate resources silently. For sample commands, see this GitHub repository to identify stale namespaces.

Actions

Once you've identified redundant resources, take action to reclaim costs. The cleanup script covers:

- Deleting completed/failed Jobs older than 7 days
- Removing pods stuck in Failed state
- Scaling down deployments in unused namespaces
- Deleting orphaned PVCs (after confirmation)

For cleanup commands, see this GitHub repository *## Always run cleanup scripts in* `--dry-run` *mode first and confirm with the owning team before deleting any resource in production.*

Abandoned workloads fragment your cluster capacity and prevent autoscalers from consolidating nodes. Cleaning them up often frees enough resources to scale down one or more nodes immediately, typically saving $70–$200/month per reclaimed node.

Recommendations

- Implement a "traffic audit" dashboard showing per-application request rates
- Set alerts for deployments with zero ingress traffic for more than 72 hours
- Use Kubernetes `scale-to-zero` solutions (e.g., KEDA (Kubernetes Event-driven Autoscaling) with minReplicaCount: 0) for intermittent workloads
- Establish an application lifecycle policy requiring periodic revalidation of deployed services
- Set `ttlSecondsAfterFinished: 86400` on all Job specs to auto-clean completed jobs after 24 hours

For the complete scripts and manifests, see the 02-application-cleanup folder in the code repository.

EKS control plane upgrade cost savings

Why upgrades save money

AWS charges a flat rate for the EKS control plane, but the rate increases significantly once your cluster version moves from **standard support** into **extended support**. For current pricing details, see the Amazon EKS pricing page.

Beyond the direct control plane charge, outdated versions also lead to:

- **Incompatibility with newer, cheaper instance types:** newer Kubernetes versions support the latest EC2 families (M7g, C7g, R7g) that offer better price-performance
- **Missing performance improvements:** scheduler enhancements, memory management fixes, and networking optimizations in newer versions can reduce the number of nodes required
- **Security patching overhead:** backporting fixes to older versions increases operational cost
- **Addon drift:** older versions lock you into older addon versions that may lack cost-relevant features (e.g., Karpenter improvements, EBS CSI (Container Storage Interface) gp3 defaults)
- **Extended support exit pressure:** once extended support ends, the cluster is auto-upgraded, potentially causing unplanned disruption if not proactively managed

Amazon EKS version lifecycle reference

For the current Kubernetes version calendar including standard support end dates and extended support windows, refer to the official Amazon EKS Kubernetes release calendar.

Identify clusters at risk

Scan all your clusters across regions to flag which ones are running on extended support and costing you extra. For sample commands, see this GitHub repository to identify clusters at risk.

Estimate your extended support waste

Quantify how much you're overpaying so you can prioritize upgrades with a clear dollar figure for leadership. For sample commands, see this GitHub repository

CloudWatch: Monitor Control Plane Health Pre-Upgrade

The health check validates: API server availability (no 5xx spikes), zero failed nodes, no deprecated API usage that would break post-upgrade, and addon compatibility with the target version. Confirm your cluster is healthy before upgrading, resolving existing issues first prevents failed upgrades and unplanned downtime. For sample commands, see this GitHub repository

Actions

Always perform upgrades in non-production environments first. Ensure you have tested addon compatibility and verified no deprecated API usage before upgrading production clusters.

Pre-upgrade compatibility check

Verify that your addons, APIs, and workloads are compatible with the target version to avoid breaking changes during the upgrade. The script checks addon version compatibility, deprecated API usage, PodSecurityPolicy presence, and workload-level feature dependencies.

For pre-upgrade compatibility check script, see this GitHub repository

Upgrade procedure (cost-aware, zero-downtime)

Execute the upgrade in the correct order: control plane first, then addons, then nodes, to maintain availability while moving to a cheaper support tier. The script handles:

- Control plane version upgrade (15–30 minutes, no workload impact)
- EKS-managed addon upgrades (vpc-cni, kube-proxy, coredns) to compatible versions
- Node rotation via managed node group rolling update or Karpenter drift annotation
- Post-upgrade health verification

For upgrade procedure script, see this GitHub repository

Key takeaway: Upgrading clusters out of extended support is often the single highest-ROI cost action, it requires no application changes and delivers immediate savings. Use the estimation script above with current EKS pricing to quantify your specific savings.

For the complete scripts and manifests, see the 03-eks-control-plane-upgrade folder in the code repository.

Karpenter cost optimization

Karpenter is the most powerful tool for EKS compute cost optimization. It replaces Cluster Autoscaler with faster, smarter provisioning that directly reduces costs through better instance selection, better consolidation, and Spot-aware scheduling. The sections below cover how to configure Karpenter for maximum savings.

For teams that prefer a fully managed alternative with less control, EKS Auto Mode is also available. However **Karpenter is the recommended approach,** for organizations that want full control over cost optimization, workload isolation, and scaling behavior.

Amazon EKS Auto Mode (managed option)

Amazon EKS Auto Mode (launched December 2024) automates compute management entirely — no Karpenter installation, no NodePool configuration, no autoscaler upgrades. AWS handles instance selection, Spot/On-Demand mixing, consolidation, and node patching automatically. EKS Auto Mode is best suited for teams without dedicated platform engineering capacity, or for dev/test environments where operational simplicity outweighs fine-grained cost control. It does not support separate NodePool configurations, custom disruption budgets, workload-specific taints, or explicit Spot/On-Demand ratio control.

When to use Amazon EKS Auto Mode

- **Greenfield clusters:** Get cost-optimal node provisioning out of the box without upfront tuning effort.
- **Lean teams**: Reduce the platform engineering burden by letting AWS manage node lifecycle, instance selection, and patching decisions.
- **Homogeneous workloads:** Standard web services and APIs that don't need dedicated GPU, Spark, or high-isolation node pools.
- **Simplicity-first environments**: Accept slightly less granular control in exchange for zero operational overhead on node management.

For sample commands, see this GitHub repository to create a new cluster with Auto Mode or enable it on an existing cluster.

When to use self-managed Karpenter

- You need **separate NodePools** per workload type (PDB-protected, non-PDB, GPU, Spark) with distinct cost strategies for each.
- You require **explicit control over disruption budgets**, expiry periods, and termination grace periods.
- Your **Spot/On-Demand ratio** must be governed by specific policies (for example, minimum 20% On-Demand for production, 100% Spot for batch).
- You run **Spark or batch workloads** needing consolidationPolicy: WhenEmpty to avoid disrupting in-progress jobs.
- **Regulatory or compliance requirements** mandate explicit instance family selection and audit trails for node provisioning decisions.

Migrating an existing cluster to Amazon EKS Auto Mode

For clusters currently using self-managed Karpenter or Cluster Autoscaler, migration to Auto Mode is a phased process — enable Auto Mode first, let it provision new nodes, then drain the old ones. The migration script handles:

- Enabling Auto Mode and waiting for new nodes to provision
- Cordoning and draining old self-managed nodes gracefully
- Verifying all pods are running on Auto Mode nodes
- Removing old node groups, Karpenter, or Cluster Autoscaler

For complete migration script, see this GitHub repository

Validation after migration

Migration considerations:

- PodDisruptionBudgets — Auto Mode respects PDBs during node rotation. Ensure PDBs are configured before draining.
- DaemonSets — Auto Mode nodes run standard EKS-managed DaemonSets. Custom DaemonSets will be scheduled automatically.

- Taints and tolerations — Review any custom taints on existing nodes. Auto Mode uses its own scheduling logic; workloads relying on custom taints may need toleration updates.
- Spot interruption handling — Auto Mode manages Spot lifecycle internally. Remove any custom Spot interruption handlers (AWS Node Termination Handler) as they're no longer needed.

For the complete migration and validation scripts, see the 04-karpenter-cost-optimization folder in the code repository.

Key takeaway: EKS Auto Mode is ideal for teams that want zero operational overhead on node management. For clusters requiring workload-specific NodePools, custom disruption budgets, or explicit Spot/On-Demand control, continue with self-managed Karpenter as described in the sections below.

Workload tiered Nodepools: Right capacity for the right workload

Isolate workloads into dedicated NodePools based on their characteristics — this prevents low-priority batch jobs from blocking expensive on-demand capacity meant for production, and allows tailored cost strategies per workload type.

Actions

Create separate NodePools based on workload criticality to manage disruption and keep blocking nodes isolated from other workloads. The sample manifest provides three NodePool configurations:

- **critical-workloads** — On-Demand, Graviton, with time-based disruption budgets (conservative during business hours, zero during peak, aggressive on weekends)
- **non-pdb-workloads** — Spot-first for stateless workloads with 50% disruption tolerance
- **gpu-workloads** — On-Demand GPU instances with `WhenEmpty` consolidation and `nvidia.com/gpu` taint

For complete NodePool manifests, see the workload-tiered-nodepools.yaml and to validate workloads are landing on the correct pool see this verify-nodepool-placement.sh

Key takeaway: Map your workloads into the correct nodepool based on workload requirements. You can also create nodepools based on tiers (production, staging, batch, dev). This will help to

explore Spot savings options for non-production workload as well. For the complete scripts and manifests, see the 04-karpenter-cost-optimization folder in the code repository.

Node expiry and termination grace period

Force periodic node replacement to pick up cheaper instance types, newer AMIs, and prevent long-running nodes from accumulating drift. Control how gracefully pods are evicted to balance cost reclaim speed with application stability.

Why this matters for cost

- **expireAfter**: Nodes running for weeks may be on older, more expensive instance types. Periodic rotation lets Karpenter re-evaluate and pick the cheapest option available now.
- **terminationGracePeriod":** Without this, pods with long shutdown hooks can keep a node alive (and billing) for hours. Setting a cap ensures nodes are reclaimed within a predictable window.

For sample NodePool configuration with expiry and termination grace period settings, see the node-expiry-termination.yaml , and verify-node-expiry.sh to check node ages and monitor expiry-driven replacements.

Actions

Both *expireAfter* and *terminationGracePeriod* directly affect running workloads — set them too aggressively and you risk disrupting active traffic; set them too conservatively and you pay for idle compute. Tune these values based on your workload's tolerance for restarts.

TerminationGracePeriod Recommendation

Workload Type	TerminationGracePeriod	Rationale
Stateless web services	2–6 hours	Quick drain, pods reschedule fast
Stateful applications	24–48 hours	Time for graceful shutdown and data sync
Spark/batch jobs	72+ hours	Allow in-progress jobs to complete

GPU training	Match job duration	Avoid wasting GPU hours

Key takeaway: Without expiry, nodes can run for weeks on instance types that are no longer the cheapest option. Periodic rotation lets Karpenter continuously re-optimize your fleet, and capping termination grace prevents dying pods from holding nodes hostage.

Disruption budgets

Disruption budgets control how many nodes Karpenter can disrupt simultaneously — this prevents aggressive consolidation from causing availability issues while still allowing cost optimization to proceed.

For sample NodePool configuration with schedule-aware disruption budgets, see the disruption-budgets.yaml that:

- Limits disruption to 10% of nodes during normal operations
- Blocks all disruptions during peak traffic hours (Mon–Fri 9–11am UTC)
- Allows aggressive 40% consolidation on weekends

Why this matters for cost

- **Too restrictive (e.g.,** `nodes: "1"`**):** consolidation happens so slowly that underutilized nodes linger for hours, wasting money.
- **Too aggressive (e.g.,** `nodes: "50%"`**):** mass disruption can trigger unnecessary scale-ups as pods reschedule, temporarily increasing cost.
- **Scheduled budgets** let you consolidate aggressively during off-hours when traffic is low and risk is minimal.

Identify nodeclaims blocking consolidation

Verify that disruption budgets are being respected and identify any NodeClaims preventing consolidation from proceeding. For sample commands , see the verify-disruption-budgets.sh

For the complete scripts and manifests, see the 04-karpenter-cost-optimization folder in the code repository.

Actions

Start with `nodes: "10%"` as your default budget, add a scheduled `nodes: "30%"` window during off-hours for aggressive consolidation, and set `nodes: "0"` during known peak events. Review disruption-blocked events weekly — if you see frequent blocks, your budget may be too restrictive and costing you money.

Key takeaway: Disruption budgets are the throttle on your cost savings. Too tight and consolidation stalls, leaving you paying for underutilized nodes. Too loose and you risk availability incidents that cost more than the compute you saved. Schedule-based budgets give you the best of both worlds.

Disruption preference policy

Disruption Preference Policy tells Karpenter which nodes to disrupt first when consolidating — prioritize removing the most expensive or least utilized nodes to maximize savings per disruption event.

The configuration covers:

- NodePool-level budget `reasons` (Underutilized, Empty, Drifted) to target on-demand nodes first for Spot replacement
- Pod-level `karpenter.sh/do-not-disrupt` annotations to control which workloads move first during consolidation
- Topology spread constraints to prevent consolidation from concentrating pods in a single zone

For the NodePool and Deployment configuration, see the disruption-preference-policy.yaml and verify-disruption-preference.sh for verification commands.

Actions

Combine budgets with preference policies to create a disruption strategy that removes the most expensive nodes first, during the safest windows, at a rate that doesn't impact availability.

Key takeaway: Not all disruptions save the same amount. Prioritizing on-demand node removal over Spot, and empty nodes over underutilized ones, maximizes the dollar value of each disruption event.

Cluster Autoscaler optimization

For clusters still running Cluster Autoscaler, the default configuration prioritizes safety over cost. Tuning these parameters controls how quickly underutilized nodes are reclaimed and how efficiently new capacity is provisioned.

Cost-Optimized Cluster Autoscaler Configuration

The key tuning changes vs defaults:

- `scale-down-delay-after-add`: 5m (default 10m): faster reclaim after scale-up
- `scale-down-unneeded-time`: 5m (default 10m): detect idle nodes sooner
- `scale-down-utilization-threshold`: 0.5: scale down if node is <50% utilized
- `expander`: least-waste: choose node group with best bin-packing fit
- `balance-similar-node-groups`: true: spread across AZs for Spot availability

For the complete deployment manifest,see the cluster-autoscaler-cost-optimized.yaml

Verify autoscaler behavior

Confirm that scale-down is actually happening, misconfigured flags or pod disruption budgets can silently block node removal. The verification script checks:

- Autoscaler status ConfigMap for current decisions
- Nodes with `scale-down-disabled` label (blocked from removal)
- Pending pods that may be triggering unnecessary scale-up
- Node utilization levels (nodes below 50% should be consolidated)
- CloudWatch node count trend (flat count despite variable traffic = paying for unused capacity)

For the verification commands, see the verify-autoscaler-behavior.sh

For the complete scripts and manifests, see the 04-karpenter-cost-optimization folder in the code repository.

Actions

Set `scale-down-utilization-threshold` to 0.5, `scale-down-unneeded-time` to 5m, and use the `least-waste` expander. Monitor node count trends weekly, if it's flat while pod count fluctuates, your scale-down settings are too conservative.

Key takeaway: The default Cluster Autoscaler settings are designed for safety, not cost. Tuning scale-down aggressiveness and using least-waste bin-packing can reduce node count by 20–40% without impacting workload availability.

Use single Cluster Autoscaler across AZs

Run one Cluster Autoscaler deployment that manages node groups across all Availability Zones — this gives the autoscaler full visibility into cluster-wide capacity and prevents AZ-isolated scaling decisions that lead to imbalanced, over-provisioned clusters.

For the deployment manifest, see the single-ca-multi-az.yaml and verify-ca-az-distribution.sh to verify node distribution across AZs.

Actions

Consolidate multiple per-AZ Cluster Autoscaler deployments into a single instance with `--balance-similar-node-groups=true`. This prevents one AZ from scaling up while another has idle capacity, eliminating cross-AZ over-provisioning.

Cost impact

Configuration	Min Nodes	Monthly Min Cost (m6g.large)
3 CAs × 1 min node	3	~$210/month
1 CA × 1 min node	1	~$70/month
Savings	**2 nodes**	**~$140/month per cluster**

Key takeaway: Multiple autoscaler instances per AZ can't see each other's capacity. A single CA with cross-AZ visibility makes globally optimal scaling decisions and avoids redundant nodes sitting idle in one zone while another scales up.

Set min-node-count appropriately

The minimum node count per node group determines your cost floor — set it too high and you pay for idle nodes 24/7, set it too low and you risk cold-start delays during traffic spikes.

Assessment criteria for min-node-count:

- Calculate total resource requests for always-on workloads (monitoring, logging, mesh)
- Add headroom for DaemonSets (each node runs kube-proxy, VPC CNI, monitoring agent)
- Factor in PDB requirements — if a deployment requires 2 replicas across 2 AZs, minimum is 2 nodes

min_nodes = ceil(total_pod_requests_at_lowest_traffic / allocatable_per_node)

For the commands to assess your current min-node-count, check off-peak utilization, and calculate the optimal minimum based on DaemonSet overhead and PDB requirements verify-min-node-count.sh .

Actions

Audit your node group minimums against actual off-peak utilization. If nodes at minimum count run below 40% CPU/memory, reduce the minimum. For non-production clusters, set `minSize=0` with Karpenter or `minSize=1` with Cluster Autoscaler to allow full scale-down outside business hours.

```
# Update min size for a managed node group
aws eks update-nodegroup-config --cluster-name my-cluster \
  --nodegroup-name my-nodegroup \
  --scaling-config minSize=2,maxSize=20,desiredSize=3
```

Key takeaway: Every node in your minimum count runs 24/7/365 regardless of demand. An over-provisioned minimum of just 2 extra nodes costs you the equivalent of those instances running all year — often thousands of dollars per node group.

Use expander strategy

The expander determines how Cluster Autoscaler chooses which node group to scale up when multiple options can fit pending pods — the wrong strategy leads to consistently picking expensive or oversized instances.

Expander	Behavior	Best For
`least-waste`	Chooses group that will have least idle CPU/memory after scheduling	**Cost optimizat ion** (recommended)
`most-pods`	Chooses group that can schedule the most pending pods	Batch processing
`priority`	Uses user-defined priority list	Graviton-first, then x86 fallback
`random`	Random selection (default in some versions)	Not recommended for cost optimization

For the deployment manifest with `least-waste` expander and a priority-based ConfigMap for Graviton-first strategy expander-strategy.yaml

Check expander

Verify which expander is active and confirm scale-up decisions match your priority configuration. For the verification commands, see the verify-expander-strategy.sh

For the complete scripts and manifests, see the 4-karpenter-cost-optimization folder in the code repository.

Actions

Switch from the default `random` expander to `least-waste` for general cost optimization, or `priority` if you have mixed Spot/On-Demand/Graviton node groups and want explicit control over which scales first. Never use `random` in production — it ignores cost entirely.

Key takeaway: The expander strategy is a one-line change that determines whether every scale-up event picks the cheapest option or an arbitrary one. `least-waste` alone can reduce per-scale-up cost by 20–30% by avoiding oversized instances.

CPU and memory right-sizing

Over-provisioned requests are the #1 hidden cost in Kubernetes. Pods requesting more than they use prevent bin-packing and force unnecessary nodes.

Identify Over-Provisioned Workloads

Compare resource requests against actual usage to find pods consuming significantly less than they reserve. Pods using less than 30% of their CPU request are candidates for right-sizing.

For kubectl-based commands to compare requests vs usage and identify waste, see the identify-over-provisioned.sh

CloudWatch: Container Insights for Right-Sizing

For a more accurate 7-day view (vs point-in-time `kubectl top`), use CloudWatch Container Insights metrics to measure average and peak CPU/memory utilization per namespace.

For sample commands, see the cloudwatch-right-sizing-metrics.sh

Actions

Implement Vertical Pod Autoscaler (VPA) for Recommendations, Refer to this GitHub repository for sample commands. For sample commands, to right-size pod resource requests, see this GitHub repository

Horizontal Pod Autoscaler (HPA) for Efficient Scaling, For sample commands to configured HPA for cost-efficient scaling, see this GitHub repository

Deploy VPA in `"Off"` mode first to collect recommendations for 7 days, then right-size requests to p95 usage + 20% buffer. Combine with HPA to scale horizontally at 70% utilization, this keeps pods lean while handling traffic spikes.

Key takeaway: Over-provisioned requests are invisible waste, they don't show up as errors or alerts, but they force extra nodes to exist. Right-sizing is often the single biggest cost win within an active cluster.

Graviton instance migration

AWS Graviton (arm64) instances, deliver up to 40% better price-performance compared to equivalent x86 instances.

Identify workloads ready for graviton

For sample commands to identify readiness for Graviton (ARM64) migration, see this GitHub repository

Graviton NodePool with Karpenter

For sample commands to configured Karpenter NodePool configured to prefer Graviton (ARM64) instances, see this GitHub repository **Verify Graviton Usage and Cost Comparison**

For commands to verify Graviton nodes are being used, compare cost, and validate performance, see this verify-graviton-usage.sh

Actions

Start by verifying your container images support multi-arch (most official images do).

- Create a Graviton-preferred NodePool with `weight: 80` and let Karpenter schedule new workloads there. Monitor for 7 days, then remove amd64 from the pool for workloads that run cleanly on arm64.
- Build multi-arch container images using `docker buildx` with `--platform linux/amd64,linux/arm64`
- Use Karpenter's `preference-policy` to prefer Graviton but fall back to x86 if ARM images aren't available
- Monitor performance metrics closely during initial migration (latency, error rate, throughput)
- Estimate savings: Graviton typically provides 20–40% cost reduction for compute workloads

Migrate plugins and infrastructure components

Start with components that already have multi-arch support:

Component	Graviton Support	Migration Complexity
CoreDNS	# Native	Low
AWS Load Balancer Controller	# Native	Low
Karpenter	# Native	Low
Nginx Ingress	# Native	Low

Key takeaway: Graviton migration is low-risk, high-reward. Most containerized workloads run on arm64 without code changes, you're simply paying less for the same (or better) performance.

Vendor and custom metrics cost reduction

Amazon CloudWatch, Prometheus, and third-party monitoring can become significant cost drivers in Amazon EKS.

Check container insights, prometheus, ADOT (AWS Distro for OpenTelemetry) metrics

For sample commands to audit Container Insights metrics and estimate cost, see the audit-container-insights-metrics.sh

Reduce Prometheus/ADOT Metric Cardinality

For sample commands to identify and reduce high-cardinality Prometheus metrics, see this reduce-prometheus-cardinality.sh

Prometheus Relabeling to Drop Expensive Metrics

For a sample Prometheus ServiceMonitor with metric relabeling to drop expensive metrics, see this servicemonitor-cost-optimized.yaml

CloudWatch Metric Streams: Cost-Effective Alternative

Use CloudWatch metric filters to extract only the signals you need from existing log data instead of publishing continuous custom metrics, see this cloudwatch-metric-filter.sh.

Vendor Metrics Cost Checklist

Metric Source	Cost Driver	Optimization
CloudWatch Container Insights	Per-metric pricing ($0.30/metric/month)	Disable enhanced metrics for non-prod
Amazon Managed Prometheus	Ingested samples ($0.03/10M samples)	Drop unused metrics via relabeling
CloudWatch Logs	Ingestion + storage ($0.50/GB + $0.03/GB)	Filter at source, reduce log verbosity

Third-party (Datadog, New Relic)	Per-host or per-container pricing	Use namespace-level sampling
Custom metrics via PutMetricData	$0.30/metric/month + API calls	Batch calls, reduce dimensions

Actions

Audit custom metrics and replace them with default metrics or built-in integrations wherever possible.

Audit process

For sample commands to list custom metrics, estimate their monthly cost, and identify ones that duplicate Container Insights defaults, see this audit-custom-metrics.sh

Common replacements

Custom Metric	Default Alternative	Source
Custom CPU per pod	`pod_cpu_utilization`	Container Insights
Custom memory per pod	`pod_memory_utilization`	Container Insights
Custom node count	`cluster_node_count`	Container Insights
Custom network bytes	`pod_network_rx_bytes` / `pod_network_tx_bytes`	Container Insights
Custom disk usage	`node_filesystem_utilization`	Container Insights
Custom request count	ALB metrics / Service Mesh	Built-in

Cleanup steps

1. **Inventory** – List all custom metrics and their associated costs
2. **Map** – Identify which default metrics or integrations provide equivalent data
3. **Validate** – Confirm dashboards and alarms work with default metrics
4. **Remove** – Delete custom metric publishing from application configurations
5. **Monitor** – Verify no alerting gaps after removal

Audit your metric count with `aws cloudwatch list-metrics`, drop unused metrics via Prometheus relabeling, disable enhanced Container Insights in non-production clusters, and set `ttlSecondsAfterFinished` on jobs to auto-clean metric sources.

key takeaway: Monitoring costs scale with cluster size unless you actively manage cardinality. A 100-node cluster can easily generate thousands of metrics, most of which nobody looks at. Filter at the source, not after ingestion.

Recommendations

- Enable CloudWatch Container Insights as the default observability layer, it provides comprehensive metrics at lower cost than custom metrics
- Use the CloudWatch Metrics Calculator to estimate savings from custom metric removal
- Retain custom metrics only when they provide unique business-level data not available from infrastructure metrics
- Review metric publishing frequency, reduce from 1-second to 60-second resolution where real-time precision isn't required (saves on high-resolution metric charges)

Spark jobs optimization

Apache Spark jobs on EKS have unique characteristics, they require large burst capacity, run for variable durations, and can waste resources if drivers outlive executors or if executor pools are over-provisioned.

Actions

Implement Spark-specific node management and scheduling optimizations. Refer to this GitHub repository for sample commands to create dedicated nodepool for Spark workloads.

Spark configuration for cost optimization

For sample commands to create dynamic allocation scales executors based on pending tasks,see this GitHub repository

Spark job lifecycle optimization

Optimization	Configuration	Savings
For the complete scripts and manifests, see the 09-spark-jobs-optimization folder in the code repository.		
Dynamic allocation	`spark.dynamicAllocation.enabled=true`	30–50% fewer executor-hours
Executor idle timeout	`spark.dynamicAllocation.executorIdleTimeout=60s`	Rapid scale-down
Spot instances for executors	Node selector for spot capacity type	60–90% off on-demand pricing
On-demand for driver	Prevent job failure from Spot interruption	N/A (reliability)

Node decommissioning	`spark.decommission.enabled=true`	Graceful Spot handling

Recommendations:

- Use Spot instances for Spark executors (they are fault-tolerant by design)
- Keep Spark drivers on On-Demand instances to prevent job failures
- Enable dynamic allocation to scale executors based on pending tasks
- Set aggressive `executorIdleTimeout` (30–60s) to release resources quickly
- Use Karpenter's `consolidationPolicy: WhenEmpty` for Spark NodePools to avoid disrupting running executors
- Schedule large batch Spark jobs during off-peak hours for better Spot availability and lower contention

Additional cost saving strategies

Beyond the core optimization areas above, these supplementary strategies address storage, networking, scheduling, and governance, each delivering incremental savings that compound across clusters.

EBS volume optimization

Storage costs accumulate silently — orphaned volumes, oversized PVCs, and legacy gp2 volumes all add up without triggering any alerts.

For commands to find unattached (orphaned) volumes, compare PVC provisioned sizes, and identify legacy gp2 volumes, see this identify-ebs-waste.sh

Migrate gp2 → gp3 for immediate savings

gp3 volumes are ~20% cheaper than gp2 at baseline and include a free 3000 IOPS / 125 MB/s baseline. For a cost-optimized StorageClass definition, see this gp3-storageclass.yaml

For the complete scripts and manifests, see the 10-additional-cost-saving-strategies folder in the code repository.

Namespace resource quotas (prevent waste)

Cap resource consumption per namespace to prevent any single team or environment from over-provisioning and inflating cluster-wide costs. For sample commands to Namespace resource quota to cap consumption and prevent over-provisioning, see this GitHub repository

Schedule non-production workloads

Scale non-production environments to zero outside business hours — dev and staging clusters sitting idle overnight and on weekends represent pure waste. For sample commands to create CronJobs to scale dev/staging to zero outside business hours, see this GitHub repository

Spot instance interruption handling

Make Spot instances viable for more workloads by handling interruptions gracefully — the more workloads you can safely run on Spot, the greater your savings.

For a sample pod configuration with a graceful termination grace period, see this spot-interruption-handling.yaml , a preStop hook to drain connections, and topology spread constraints for resilience across Spot pools.

For the complete scripts and manifests, see the 10-additional-cost-saving-strategies folder in the code repository.

Network cost reduction

Cross-AZ data transfer is a hidden cost that grows with traffic — keeping service-to-service communication within the same AZ eliminates per-GB transfer charges ($0.01/GB each way).

For commands to check pod distribution across AZs and identify services that would benefit from topology-aware routing, see check-cross-az-traffic.sh , and topology-aware-routing.yaml for a sample Service configured to keep traffic in-zone.

For the complete scripts and manifests, see the 10-additional-cost-saving-strategies folder in the code repository.

Next steps

Optimization without measurement is guesswork, track utilization and node count trends over time to confirm your changes are delivering real cost reduction and to catch regressions early.

Monitoring your savings

Monitoring Your Savings

Optimization without measurement is guesswork — track utilization and node count trends over time to confirm your changes are delivering real cost reduction and to catch regressions early.

Create a Cost Dashboard

To set up a CloudWatch dashboard to track cluster efficiency (node utilization and node count over time), see this create-cost-dashboard.sh.

Weekly Cost Review Script

Run a weekly report covering cluster version, node utilization, Spot/Graviton distribution, top consumers, pending pods, and orphaned volumes. Refer to weekly-cost-review.sh.

CloudWatch: Monitor Karpenter Savings

Use CloudWatch and Cost Explorer to validate that Karpenter's decisions translate into actual spend reduction — Spot adoption, instance right-selection, and consolidation should all be visible in your billing data. Refer to monitor-karpenter-savings.sh.

Resources

References

- Amazon EKS Best Practices Guide: Cost Optimization
- Karpenter Documentation
- AWS Graviton Getting Started Guide
- Kubernetes Cluster Autoscaler FAQ
- Spark on Kubernetes with Amazon EKS

Contributors

Contributors to this document include:

- Abhay kumar, Lead Consultant, AWS
- Akash Kumar, Sr. Lead Consultant, AWS
- Vinodkumar Mandalapu, Lead Consultant, AWS
- Viyoma Sachdeva, Pr. PS Cloud Architect, AWS

Document history

The following table describes significant changes to this guide.

Change	Description	Date
Initial publication	—	August 10, 2026