We have built, migrated, and rescued over 20 EKS clusters across industries ranging from fintech to healthcare. Most of the clusters we inherit during audits share the same set of problems. This post covers what we have learned about making EKS actually production-ready, not just "it works in staging" ready.

Why Most EKS Clusters Fail in Production

The pattern is predictable. A team spins up EKS using `eksctl` or a basic Terraform module, deploys their app, and calls it done. Six months later they are dealing with IP exhaustion, surprise $15k bills, failed node drains during upgrades, and zero visibility into what is actually happening inside the cluster.

The root causes are almost always the same:

- **No networking plan.** They picked a VPC CIDR without thinking about IP consumption patterns.
- **Default managed node groups with no strategy.** Everything runs on the same nodes.
- **No GitOps.** Deployments are `kubectl apply` from laptops.
- **Kubernetes Secrets in plaintext etcd.** No external secrets management.
- **No observability from day one.** Prometheus gets bolted on after the first outage.

Let us address each of these.

VPC and Networking Design That Actually Works

The most common mistake we see: a `/16` VPC with two `/24` subnets. This seems generous until you realize that each pod in EKS gets a real VPC IP address by default. A single `m5.xlarge` node can consume 58 IP addresses. Run 30 nodes and you have burned through your subnet capacity.

Here is what we deploy:

- **VPC CIDR:** `/16` for the VPC, but subnet design is where it matters.
- **Private subnets:** At least `/19` per AZ for pod networking. That gives you 8,190 IPs per subnet.
- **Public subnets:** `/24` per AZ for load balancers and NAT gateways. You do not need many IPs here.
- **Secondary CIDR blocks** from the `100.64.0.0/10` range for pod networking when you expect large clusters.

Enable VPC CNI prefix delegation. This is the single most impactful networking change you can make. Instead of assigning one IP per ENI slot, prefix delegation assigns a `/28` prefix (16 IPs) per slot. This dramatically increases pod density per node.

Set these environment variables on the `aws-node` DaemonSet:

```
env:
  - name: ENABLE_PREFIX_DELEGATION
    value: "true"
  - name: WARM_PREFIX_TARGET
    value: "1"
  - name: MINIMUM_IP_TARGET
    value: "3"
```

Do not set `WARM_IP_TARGET` and `WARM_PREFIX_TARGET` at the same time. Pick one strategy. We use prefix delegation on every cluster now.

When NOT to use prefix delegation: If you are running security groups for pods (SGP), prefix delegation is incompatible. In that case, plan for larger subnets or secondary CIDRs.

Karpenter vs Cluster Autoscaler: Why We Switched

We ran Cluster Autoscaler (CA) for years. It works, but Karpenter is better in every dimension that matters for production:

- **Speed.** Karpenter provisions nodes in under 60 seconds. CA waits for the ASG to scale, which can take 3-5 minutes.
- **Right-sizing.** Karpenter picks instance types based on pending pod requirements. CA scales existing node groups, so you are stuck with whatever instance type you pre-defined.
- **Consolidation.** Karpenter actively moves workloads to reduce node count. CA only scales down idle nodes.
- **Spot handling.** Karpenter natively handles spot interruptions with graceful node drain.

Here is our standard Karpenter `NodePool` configuration:

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: default
spec:
  template:
    metadata:
      labels:
        team: platform
    spec:
      requirements:
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64"]
```

```
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["on-demand", "spot"]
        - key: karpenter.k8s.aws/instance-category
          operator: In
          values: ["c", "m", "r"]
        - key: karpenter.k8s.aws/instance-generation
          operator: Gt
          values: ["5"]
        - key: karpenter.k8s.aws/instance-size
          operator: NotIn
          values: ["nano", "micro", "small"]
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      expireAfter: 720h
  disruption:
    consolidationPolicy: WhenEmptyOrUnderutilized
    consolidateAfter: 60s
  limits:
    cpu: "400"
    memory: 1600Gi
  weight: 50
---
apiVersion: karpenter.k8s.aws/v1
kind: EC2NodeClass
metadata:
  name: default
spec:
  role: KarpenterNodeRole-${CLUSTER_NAME}
  amiSelectorTerms:
    - alias: al2023@latest
  subnetSelectorTerms:
    - tags:
        karpenter.sh/discovery: ${CLUSTER_NAME}
  securityGroupSelectorTerms:
    - tags:
        karpenter.sh/discovery: ${CLUSTER_NAME}
  blockDeviceMappings:
    - deviceName: /dev/xvda
```

```
      ebs:
        volumeSize: 100Gi
        volumeType: gp3
        iops: 3000
        throughput: 125
        deleteOnTermination: true
```

Key decisions in this config:

- **Instance generation > 5.** No reason to run old hardware. Graviton and current-gen Intel/AMD are cheaper and faster.
- **Exclude nano/micro/small.** The kubelet and system pods consume too much overhead on tiny instances.
- `expireAfter: 720h`. Nodes get recycled every 30 days. This forces you to handle node churn gracefully and keeps AMIs fresh.
- `consolidationPolicy: WhenEmptyOrUnderutilized`. Karpenter actively consolidates workloads. This alone saved one client 35% on compute costs.

When NOT to use Karpenter: If you need GPU node groups with specific scheduling requirements or if your organization mandates ASG-based scaling for compliance reasons, stick with managed node groups and CA. Also, Karpenter requires the pod disruption budget game to be tight - if your PDBs are misconfigured, consolidation will stall.

GitOps with ArgoCD: Multi-Cluster ApplicationSets

Every cluster we build uses ArgoCD. No exceptions. `kubectl apply` from a CI pipeline is not GitOps. It is "push and pray."

For multi-cluster environments, we use the app-of-apps pattern with `ApplicationSet` generators. Here is how we structure it:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: platform-services
  namespace: argocd
spec:
  goTemplate: true
  goTemplateOptions: ["missingkey=error"]
  generators:
    - matrix:
        generators:
          - git:
              repoURL: https://github.com/org/platform-config.git
              revision: HEAD
              directories:
                - path: "clusters/*/platform-services/*"
          - clusters:
              selector:
                matchLabels:
                  env: production
  template:
    metadata:
      name: "{{.name}}-{{index .path.segments 1}}"
      annotations:
        argocd.argoproj.io/sync-wave: "{{.path.basename}}"
    spec:
      project: platform
      source:
        repoURL: https://github.com/org/platform-config.git
        targetRevision: HEAD
        path: "{{.path.path}}"
      destination:
        server: "{{.server}}"
        namespace: "{{.path.basename}}"
      syncPolicy:
        automated:
          prune: true
          selfHeal: true
        syncOptions:
          - CreateNamespace=true
```

```
      - ServerSideApply=true
    retry:
      limit: 3
      backoff:
        duration: 30s
        factor: 2
        maxDuration: 3m
```

The matrix generator combines cluster selection with directory-based app discovery. Each directory under `clusters/<cluster-name>/platform-services/` becomes an application targeting the matching cluster.

Sync waves matter. We deploy in this order: namespaces and CRDs (wave 0), operators (wave 1), operator configs (wave 2), applications (wave 3). Getting this wrong means ArgoCD tries to create a resource before the CRD exists and the sync fails.

Always enable `selfHeal`**.** Without it, someone will `kubectl edit` a deployment in production and you will have drift you do not know about.

Node Group Strategy

Not everything should run on Karpenter-managed nodes. We use dedicated managed node groups for:

- **System components (CoreDNS, kube-proxy, EBS CSI driver).** These run on a small, always-on node group (`2x m6i.large`). Taint these nodes with `CriticalAddonsOnly`.
- **Monitoring stack (Prometheus, Grafana, Loki).** Prometheus with 15-day retention can consume 60GB+ of memory. Put it on dedicated `r6i.xlarge` nodes so it does not get evicted.
- **Ingress controllers (NGINX or ALB controller).** Dedicated nodes with known capacity. You do not want your ingress competing with application pods for resources.

Everything else goes on Karpenter-managed nodes with appropriate pod topology spread constraints.

Secrets Management: External Secrets Operator

If your secrets are in Kubernetes `Secret` objects and that is your entire strategy, you have a problem. Kubernetes Secrets are base64-encoded, not encrypted (at rest encryption via KMS helps, but it is not enough).

We deploy the External Secrets Operator (ESO) on every cluster, backed by AWS Secrets Manager:

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: api-credentials
  namespace: production
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: aws-secrets-manager
    kind: ClusterSecretStore
  target:
    name: api-credentials
    creationPolicy: Owner
    deletionPolicy: Retain
  data:
    - secretKey: api-key
      remoteRef:
        key: production/api-credentials
        property: api_key
    - secretKey: db-password
      remoteRef:
        key: production/database
        property: password
```

Use `ClusterSecretStore` with IRSA (IAM Roles for Service Accounts) for authentication. Never use static AWS credentials. Set `refreshInterval` based on your rotation policy. We default to `1h` but use `5m` for secrets that rotate frequently.

Monitoring from Day One

Every cluster gets the kube-prometheus-stack Helm chart deployed before any application workload. No exceptions.

We deploy standard `ServiceMonitor` resources for every component. The critical pattern is recording rules for SLO-based alerting rather than threshold alerts. We cover this in depth in our SLO post, but the key point is: alert on error budget burn rate, not on "CPU > 80%."

The dashboards we deploy on every cluster:

1. **Cluster overview** - node count, pod count, resource utilization, API server latency.
2. **Namespace resource consumption** - requests vs actual usage per namespace. This is where you find cost savings.
3. **Pod restart tracker** - any pod restarting more than twice in an hour gets flagged.
4. **Persistent volume usage** - catch volumes approaching capacity before they cause outages.

Cluster Upgrades: Blue-Green Strategy

In-place EKS upgrades are risky. We have seen control plane upgrades that broke webhook configurations, CRD compatibility issues that surfaced only after the upgrade, and node group upgrades that drained pods faster than PDBs could handle.

Our approach: **blue-green cluster upgrades.**

1. Provision a new cluster at the target Kubernetes version alongside the existing cluster.
2. Deploy all platform services using the same ArgoCD repository (this is why GitOps matters).
3. Run smoke tests and synthetic traffic against the new cluster.
4. Shift traffic incrementally using weighted DNS (Route 53) or service mesh.
5. Decommission the old cluster after 48 hours of stable operation.

This sounds expensive. It is. But the cost of a failed in-place upgrade during business hours is worse. For smaller clusters or non-critical environments, in-place upgrades are fine - just test the upgrade path in a staging cluster first.

Cost Optimization

The three highest-impact cost levers:

1. **Right-size resource requests.** We see over-provisioning of 3-5x on most clusters. Use the Vertical Pod Autoscaler in recommend-only mode for two weeks, then adjust requests based on P95 actual usage.
2. **Karpenter consolidation.** The config above with `WhenEmptyOrUnderutilized` continuously right-sizes your node fleet. We have seen 30-40% compute savings from this alone.
3. **Spot instances for non-critical workloads.** Batch jobs, dev environments, CI runners - all of these should be on spot. Use Karpenter's capacity-type requirement and set pod disruption budgets so spot interruptions are graceful.

Do not use Spot for: Stateful workloads, databases, single-replica services, anything where a 2-minute interruption notice is not enough time to drain gracefully.

The 10 Things We Check on Every EKS Audit

1. VPC CNI version and prefix delegation status.
2. Pod security standards enforcement (not just Pod Security Policies - those are gone).
3. IRSA configured for all AWS service access (no node-level IAM roles for application access).
4. Network policies in place (default deny per namespace at minimum).
5. Resource requests and limits set on every pod (no unbounded pods).
6. PodDisruptionBudgets on every production deployment.
7. External secrets management (not raw Kubernetes Secrets).
8. Prometheus and alerting configured with SLO-based alerts.
9. Cluster autoscaling strategy documented and tested.
10. Upgrade runbook exists and has been tested in the last 90 days.

If you fail more than three of these, your cluster is not production-ready. It might be running production traffic, but it is one incident away from a very bad day.

Final Thoughts

EKS is a solid managed Kubernetes offering, but "managed" does not mean "hands-off." The control plane is managed. Everything else - networking, scaling, security, observability, upgrades - is your responsibility. Treat your cluster like the production infrastructure it is, not like a playground where you deploy and forget.

If your team is building on EKS and wants an independent assessment of your architecture and operational practices, reach out to us. We have seen enough clusters to know what works and what does not.

`kubernetes` `aws` `eks` `production` `karpenter` `argocd`

Need help with this?

Book a free 30-minute architecture review. We'll look at your specific setup and give you honest recommendations.

Book Architecture Review →

← Back to all articles