Optimizing Kubernetes (EKS) for Telecom Applications in Multi-AZ AWS Environments

Jayavelan Jayabalan
Independent Researcher
University of Madras, India

***Abstract*— Amazon Elastic Kubernetes Service (EKS) stands out as a dependable, scalable, and fully managed platform tailored for running modern containerized applications in the cloud. Still, many organizations transitioning to cloud-native setups find it challenging to strike the right balance between keeping costs low and ensuring high availability. This paper explores a range of practical techniques and proven strategies aimed at optimizing costs while building robust EKS environments. It delves into the architectural factors influencing operational expenses—such as compute resources, storage systems, networking layers, and control plane operations—and evaluates solutions like workload rightsizing, smart autoscaling, and fine-tuning network traffic. Additionally, the discussion includes how observability tools, governance models, and thoughtful deployment methods can shape both performance and cost-effectiveness in real-world scenarios. Drawing on insights from seven trusted technical sources, it emphasizes the vital importance of integrated monitoring, adaptive autoscaling, and workload-specific resource planning in building efficient, scalable, and cost-optimized EKS environments. Advanced deployment patterns—including multi-AZ and multi-region architectures—are analyzed using service mesh technologies and traffic routing methods that enhance availability and resilience. The primary aim of the paper is to offer actionable guidance for engineers, architects, and IT leaders in designing efficient and fault-tolerant Kubernetes environments on AWS. It positions EKS not only as a powerful compute platform but also as a foundational element for achieving both operational stability and financial sustainability in cloud-native environments.**

***Index Terms*—Amazon EKS, Kubernetes, Cost Optimization, High Availability, Autoscaling, Cloud Networking, Kubernetes Governance, Pod Scaling, FinOps, Multi-AZ, Multi-region Clusters, Cloud Monitoring, Service Mesh, Cluster Autoscaler, Resource Rightsizing**

1. Introduction

Amazon Elastic Kubernetes Service (EKS) has quickly become an essential platform for enterprises scaling containerized application deployments. As a fully managed Kubernetes solution provided by AWS, EKS simplifies the complexities of orchestration infrastructure, allowing development and DevOps teams to concentrate on building, deploying, and iterating applications instead of managing the underlying systems. The service automates the setup and operation of Kubernetes control plane components and integrates tightly with a broad range of AWS-native services. This integration boosts the scalability, availability, and overall performance of cloud-based applications. Nonetheless, while EKS streamlines the Kubernetes experience, it also brings a set of financial and operational challenges that organizations must address thoughtfully to maximize its value [1][4].

One of the primary challenges in deploying applications on EKS is managing costs effectively. Organizations frequently face rising infrastructure expenses due to inefficient resource utilization, including overprovisioned compute, underused storage, and unnecessary network overhead. In contrast to traditional on-premises setups—where capital costs are typically sunk—the pay-as-you-go nature of cloud platforms like EKS makes every inefficiency translate directly into real-time financial impact. As workloads scale and clusters grow more complex in both architecture and traffic patterns, manual tracking of resource utilization becomes increasingly impractical. As a result, engineering teams are turning to automated monitoring tools, intelligent autoscaling solutions, and strategic optimization techniques to maintain cost efficiency while preserving application performance and system availability [3][7].

Furthermore, the need for high availability and operational resilience in EKS clusters adds another layer of importance to infrastructure planning. Today's applications—especially those expected to serve users around the globe or run without interruption—need the ability to recover fast when things go sideways. EKS helps tackle this by supporting multi-Availability Zone (multi-AZ) setups out of the box and by working seamlessly with auto-scaling tools that react on the fly to shifting workloads. This mix of adaptability and resilience is what gives modern cloud-native systems their edge, keeping critical services online even under pressure. That said, if these features are rolled out without careful planning or a clear view of cost implications, organizations might end up facing ballooning expenses—essentially defeating the purpose of infrastructure modernization [1][2].

Layered on top of these concerns is the fact that Kubernetes, while undeniably powerful, isn't exactly beginner-friendly. It expects you to be well-versed in a range of moving parts—pods, nodes, services, deployments, replica sets—and that's just

scratching the surface. Every one of these elements plays a role in shaping how your application behaves and what it costs to run. When oversight is loose and observability tools aren't dialed in, clusters have a way of growing out of hand—quietly soaking up resources and turning into a headache to manage. Staying lean while still building something that can take a hit means teams need to be intentional: fine-tune configurations, invest in clear visibility, trim excess resource use, and steer traffic intelligently [6][4].

This review takes a hard look at what goes into deploying Amazon EKS—with a particular focus on what it all means for your bottom line. It unpacks key architectural decisions, how scaling is approached, the nuts and bolts of autoscaling, network setup, monitoring tools, and techniques for keeping systems resilient. By leaning on real-world scenarios and trusted industry practices, the paper aims to help organizations navigate the tricky balance between running at top speed and keeping costs in check in today's fast-moving cloud environment.

2. EKS Architecture and Cost Components

If you really want to make sense of what you're paying for with Amazon EKS, it helps to first get familiar with how the system is put together. The way AWS prices things isn't random—it reflects the underlying architecture. At its core, EKS splits costs into two main chunks: the control plane and the data plane. AWS charges a flat rate for this piece, no matter how many nodes or workloads you're running. The data plane, though, is where costs tend to fluctuate. It's made up of the compute layer—usually EC2 instances or AWS Fargate—that runs the actual application pods. Since this layer scales with usage, costs vary based on instance types, how long they're running, and how much your workloads grow or shrink [1][4].

One of the earliest—and arguably most important—choices that affects cost is picking the right type of compute instance. With EKS, workloads can run on EC2 using all kinds of instance families: general-purpose, compute-optimized, memory-heavy, even GPU-backed if needed. Each comes with its own pricing quirks, depending on the family and the size you choose. For example, compute-optimized instances are cost-effective for CPU-intensive tasks, while memory-optimized instances are better suited for workloads with large in-memory datasets. Inefficiencies arise when instances are misaligned with workload requirements, such as over-provisioning memory for CPU-bound applications, leading to inflated costs. EKS also supports the use of spot instances, which are up to 90% cheaper than on-demand instances but come with the risk of interruptions. Reserved instances, though less flexible, provide significant discounts for predictable long-term usage [3][7].

Storage is another pivotal element in the EKS architecture that directly affects cost. Kubernetes supports persistent volumes using AWS Elastic Block Store (EBS), Elastic File System (EFS), and FSx for Lustre. The choice of storage backend depends on the application's IOPS requirements and access patterns. EBS volumes, for instance, are charged based on provisioned size and type (gp3, io1, etc.), and underutilization of allocated volumes contributes to silent cost accrual. Moreover, data replication, backup snapshots, and high availability configurations such as multi-AZ replication introduce additional layers of expense that must be weighed against reliability needs [4][6].

The networking layer is also a significant contributor to EKS costs and is often overlooked during planning. AWS charges for data transfer between Availability Zones, across VPCs, and through public internet gateways. Within an EKS cluster, pod-to-pod and pod-to-service communication may traverse different subnets or AZs, thereby incurring inter-AZ data transfer charges. Similarly, the use of external load balancers (such as Application Load Balancers or Network Load Balancers) introduces hourly charges and per-GB traffic costs. This is especially pronounced in architectures that rely on ingress controllers or service meshes that perform layer 7 routing across services deployed in different zones or regions [2][5].

To manage these distributed cost components, observability tools and FinOps practices become essential. AWS provides mechanisms such as the Cost and Usage Report (CUR), CloudWatch metrics, and tagging to enable detailed visibility into where and how resources are being consumed. Tagging Kubernetes resources using tools like kube-janitor or external integrations allows finance and engineering teams to correlate costs with specific teams, applications, or environments. Without this granularity, it becomes extremely difficult to identify optimization opportunities or enforce governance standards [4][7].

3. Rightsizing and Scaling Strategies

Rightsizing is one of the most impactful cost optimization strategies for Amazon EKS and involves aligning infrastructure resources—particularly compute instances—with actual application requirements. The goal is to avoid over-provisioning or under-provisioning resources, both of which can negatively affect costs and performance. EKS provides flexibility in choosing EC2 instance types and supports AWS Fargate for serverless compute. However, without intelligent sizing decisions, organizations often waste significant financial resources by allocating larger instance types than needed or failing to consolidate workloads effectively across nodes [3][4].

The choice of EC2 instance type should be driven by workload characteristics. General-purpose instances such as the M5 or M6 families offer balanced performance for a variety of applications, while compute-optimized (C series) or memory-optimized (R series) instances are better suited for specialized workloads. Sedai offers a handy visual that lays out how EC2 instance families compare when it comes to usage and bang-for-your-buck efficiency. No big surprise here—general-purpose instances lead the pack in EKS setups. They're kind of the Swiss Army knife of compute options: versatile, balanced, and rarely overkill for most everyday workloads. On the flip side, memory-optimized and compute-optimized types tend to show up in more demanding scenarios, like large-scale analytics, data crunching, or scientific computing, where the hardware needs to match the intensity of

the task. This kind of visualization is invaluable for teams trying to strike the right balance—getting the performance they need without racking up unnecessary expenses [3].

Take a look at the figure below—it gives a snapshot of how various EC2 instance types are spread out, depending on what kind of workloads they're handling and how they stack up in terms of cost versus performance.

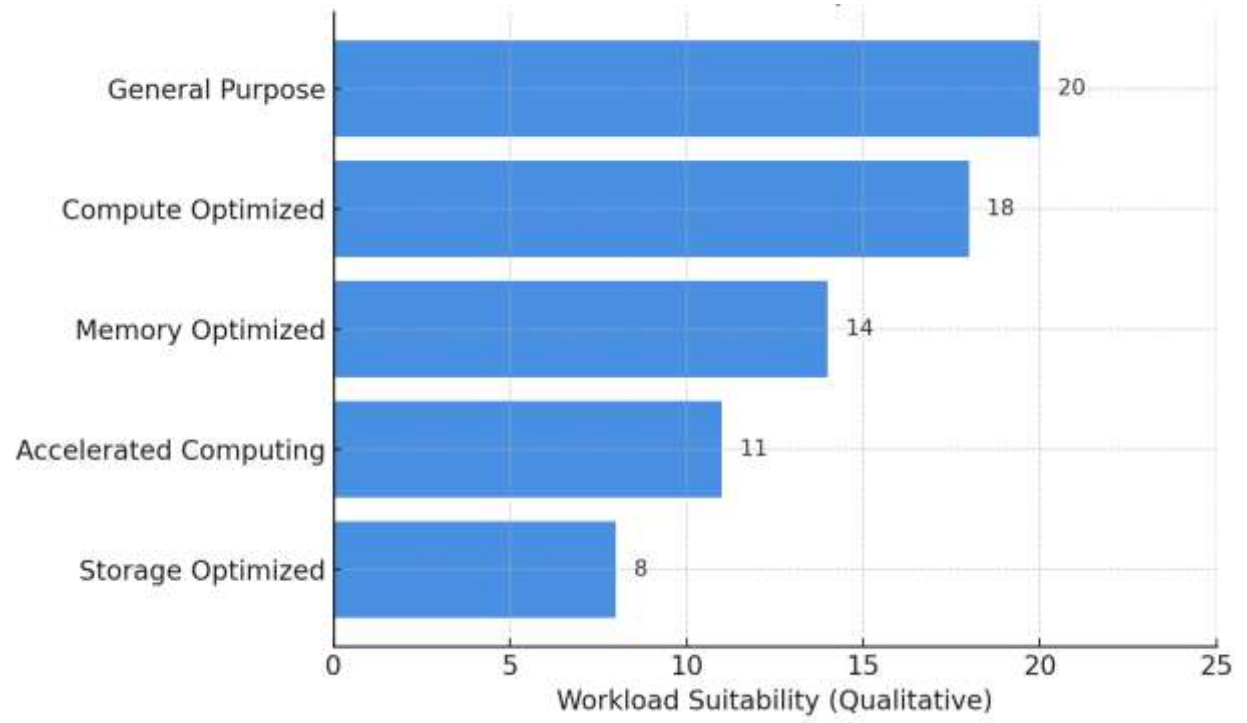


Figure: Distribution of EC2 Instance Types for EKS Cost Optimization (Sedai, 2025) [3].

Getting the instance type right is only part of the equation. There's still plenty of room to cut costs if you know how to work AWS's pricing models—things like spot instances, reserved instances, and savings plans. Take spot instances: they let you tap into unused EC2 capacity for a fraction of the price. They're perfect for jobs that don't mind getting interrupted—stateless processes, batch tasks, that sort of thing. The catch? AWS can yank them with little notice, so they're not ideal for anything critical or stateful—unless you've architected around that risk. Then there are reserved instances. These offer solid savings too, but you've got to commit for one or even three years. They're a better fit for workloads that don't change much and just keep chugging along [7][4].

When it comes to EKS, how you handle scaling really shapes your ability to rightsize. Kubernetes gives you a couple of go-to tools: horizontal pod autoscaling (HPA) and vertical pod autoscaling (VPA). HPA's the one that spins up more pods when traffic climbs. VPA? That one quietly adjusts CPU and memory settings for a single pod when it's running too hot—or barely working. These tools are great, no doubt, but they're not magic. If you dial them in too aggressively, you might get hit with jittery performance or bills you didn't see coming. Play it too safe, though, and your apps might crawl when things get busy [1][6].

On top of that, tools like the Cluster Autoscaler and Karpetner help EKS scale up—or down—the number of worker nodes based on what your pods actually need at any given moment. They keep an eye on resource usage and adjust the cluster in real time, which is a big win for keeping costs under control in environments that are constantly changing. Karpetner's especially sharp here—it's built for just-in-time provisioning and smart enough to pick the cheapest, most efficient instance type across different availability zones. The result? Less waste, more bang for your buck [4][6].

4. Autoscaling Mechanisms and Performance Optimization

Autoscaling—when done right—is one of the most powerful tools for balancing cost and performance in Amazon EKS. Application demand rarely stays flat; it ebbs, it spikes, and occasionally it just takes off out of nowhere. That's why having the ability to flex your compute resources up or down in real time isn't just helpful—it's essential. It keeps things running smoothly without throwing money at unused capacity. Kubernetes gives you a few built-in options for this: Horizontal Pod Autoscaler

(HPA), Vertical Pod Autoscaler (VPA), and the Cluster Autoscaler. Each handles a different piece of the puzzle. Get the tuning right, and your cluster stays fast, lean, and responsive—without racking up charges from idle resources [1][4].

HPA, or Horizontal Pod Autoscaler, is one of those must-have tools in Kubernetes when it comes to scaling. It watches how much CPU or memory your workloads are using—usually pulling that data from the metrics server or, in some setups, from Prometheus—and then adjusts the number of pod replicas to match the demand. So, say your app's average CPU use pushes past 70%. HPA jumps in and spins up extra pods to keep things moving. When things quiet down, it scales them back to avoid wasting resources. It's a reactive system, and it works especially well for stateless workloads that can scale sideways—like microservices or frontend apps handling bursts of traffic. That kind of flexibility is exactly why HPA has become such a key piece of elastic cloud infrastructure [1][6].

So the Vertical Pod Autoscaler—VPA for short—doesn't really scale out. Instead, it kind of looks back at how a pod has been performing—like, has it been consistently maxing out CPU? Is it barely touching its memory?—and then adjusts those resource settings. It's the kind of tool you lean on when you're dealing with workloads that can't just be copied and spread out. Think stateful databases, or those clunky legacy apps that are basically allergic to horizontal scaling. VPA can either suggest new resource configurations or go ahead and apply them automatically, helping make sure each pod gets just the right amount of compute. That said, there's a catch: most versions of VPA need to restart the pod for the changes to stick. Not ideal if you're running something that needs to stay up around the clock. Because of that, teams often pair VPA with maintenance windows or rolling update strategies to avoid downtime [6][3].

The Cluster Autoscaler, on the other hand, steps in at the node level—it's a bit of a higher-level operator. Basically, it watches for pods that are hanging around unscheduled because there's just not enough capacity. When it spots that kind of bottleneck, it fires up new nodes to free up space and keep things moving. And when some nodes are just sitting around doing nothing? It quietly shuts them down to keep resource usage tight. This kind of smart scaling is super useful in environments where traffic can spike or drop without much warning—it helps strike that tricky balance between staying responsive and not overspending. Pair it with a tool like Karpetner, and now you're talking real efficiency; the system can actually choose the cheapest instance type and zone for the job, making the whole thing even leaner [4][7].

To make it easier to see how these autoscaling tools differ—and where they actually work best together—the table below offers a side-by-side comparison, drawing from guidance shared by RST Software.

Autoscaler	**Scope**	**Best Use Case**	**Limitations**
Horizontal (HPA)	Pod level	Stateless apps with variable traffic	Depends on metrics accuracy; CPU-based only
Vertical (VPA)	Pod level	Stateful apps, non-replicable pods	Requires pod restarts for resizing
Cluster Autoscaler	Node level	Variable node-level capacity demands	Can increase launch latency

Table: Comparison of Autoscaling Tools in EKS (Jackiewicz, 2025) [6].

You throw all these autoscaling tools into the mix, and what you end up with is something that's kind of surprisingly agile. It bends when traffic spikes, stretches when demand drags out longer, and doesn't just sit there eating up budget. The magic isn't just in how fast it reacts—it's also in how it sort of learns, adjusts quietly as patterns shift. It's not perfect, but when it works, it really works. That's really the sweet spot: giving your apps just enough muscle to perform well, without throwing money at unused resources. And honestly, autoscaling isn't just about speed or keeping things smooth—it's one of the smarter ways to stay on top of cloud costs. Of course, it's not set-and-forget. You've got to keep an eye on things, tighten the thresholds, tweak the configs now and then. It's that ongoing tuning that makes autoscaling such a vital part of staying efficient in EKS.

5. CNF Integration in Telecom 5G Core Networks

The shift from Virtual Network Functions (VNFs) to Containerized Network Functions (CNFs) marks a major turning point in the telecom world—especially as the industry pushes deeper into cloud-native territory with 5G core (5GC) networks. As more operators roll out 5G Standalone (SA) architectures, CNFs are becoming a key piece of the puzzle. They help cut down deployment complexity, scale more easily, and support more fluid, on-demand service orchestration [8]. Think of CNFs as the next logical step in network softwarization—built around microservices and orchestrated through Kubernetes, but tailored for telecom-grade workloads like AMF, SMF, UPF, PCF, AUSF, and UDM.

Unlike VNFs—which usually run inside virtual machines and rely on older NFV orchestration methods—CNFs are built to be lean, nimble, and right at home in Kubernetes environments. They fire up faster, take up less space, and slot neatly into modern CI/CD pipelines, making them a better fit for today's fast-moving telecom needs [8]. That kind of agility really matters in 5G,

where speed and throughput aren't just nice to have—they're non-negotiable. CNFs make it easier to scale control plane components on the fly, which is critical for deploying and tweaking network slices in real time. Whether it's enhanced Mobile Broadband (eMBB), ultra-reliable low-latency communications (uRLLC), or massive Machine-Type Communications (mMTC), CNFs help networks adapt quickly without missing a beat.

One of the significant technical challenges introduced by CNFs is **container placement** within edge and core infrastructures. Unlike VM-based deployments, containers share the host OS, and thus performance can vary based on co-located workloads. Efficient placement must consider inter-CNF traffic patterns, compute and memory constraints, service isolation, and proximity to the data source or user endpoint. For example, CNFs such as SMF and UPF must be tightly coupled in edge locations to reduce PDU session setup latency Likewise, functions like AMF and AUSF have to stay sharp—always on, always responsive—since they're at the heart of subscriber authentication and making sure handovers happen without a glitch.

The 5G Core's Service-Based Architecture (SBA)—powered by CNFs—leans on RESTful APIs to let microservices like AMF, PCF, and UDM talk to each other over a shared message bus. This kind of decoupling definitely makes the architecture more modular and easier to adapt—but it also raises the bar. Now, things like ultra-low latency and detailed, container-level observability aren't just nice to have—they're essential [8]. Orchestration can't rely on static placement anymore. It has to respond to what's actually happening in real time—whether that's CPU throttling, I/O lag, or packets mysteriously dropping at the network edge. That's where tools like KubeEdge and Karmada start to earn their keep. They're designed with CNFs in mind and built to handle the messy realities of edge computing and distributed environments.

On top of all that, placement strategies have to consider service function chaining (SFC)—especially when CNFs are spread across multiple clusters. A common example would be chaining AMF → SMF → UPF → PCF to manage a mobile session from start to finish. Now, if you don't place those components carefully, the whole chain kind of bogs down—and yeah, that's exactly the sort of thing that tanks user experience. Nobody likes laggy sessions. What's interesting is, smarter placement strategies—stuff like Genetic Algorithms, Particle Swarm Optimization, even Deep Reinforcement Learning—have actually been shown to cut down that end-to-end latency pretty dramatically. They're not just clever—they're efficient, too. Compared to basic methods like greedy placement or bin-packing (which, let's be honest, can be a bit one-size-fits-all), these advanced techniques tend to get way better results in the real world.

In edge environments, placing CNFs isn't just about performance—it's about staying responsive to shifting bandwidth and the unpredictability of edge infrastructure. Take UPF instances, for example. When they're deployed at edge data centers, they've got to scale up or down depending on how much traffic is coming from nearby base stations (gNBs). And all the while, they need to keep backhaul traffic to the core as low as possible [8]. Smart scheduling strategies—like Gradient-Based Minimum Delay (GBMD) algorithms—help with this by choosing where and when to spin up CNFs in a way that keeps delays low and uses distributed compute efficiently.

Then there's security, which becomes even more complicated in a CNF world. Unlike VNFs that live in their own isolated VMs, CNFs often share space—they're packed into pods and sit side by side on the same nodes. That opens the door to risks like co-residency attacks or kernel-level vulnerabilities, especially if something's misconfigured or compromised. Some promising work is being done here using advanced orchestration—things like Genetic Algorithms and Simulated Annealing—to securely manage how pods are placed and moved around. The idea is to reduce exposure without wasting resources.

All in all, bringing CNFs into the fold for 5G telecom networks marks a major step forward—one that pushes network functions to be smarter, more scalable, and a whole lot more resilient. However, achieving optimal performance requires CNF-aware placement strategies, integration with orchestration platforms that consider telecom-specific SLAs, and enhanced security frameworks. As 5G networks evolve and scale globally, containerized core functions such as AMF, SMF, UPF, PCF, AUSF, and UDM will serve as the backbone of telecom innovation, driven by the flexible and programmable infrastructure that CNFs enable.

6. Cost Optimization through Networking

Networking costs in Amazon EKS are often underestimated during the design of Kubernetes clusters, yet they can constitute a significant portion of the overall cloud bill. Unlike compute and storage costs, which are more straightforward to monitor and attribute, networking expenses are influenced by the architecture of the application, the configuration of services, and the data flow patterns within and outside the cluster. In an environment where microservices communicate extensively and traffic traverses multiple layers of infrastructure, understanding and optimizing networking becomes critical for both cost efficiency and performance reliability [2][4].

A major contributor to networking cost is inter-AZ data transfer, which occurs when pods running in different Availability Zones communicate with each other. While EKS clusters distributed across multiple AZs offer resilience and fault tolerance, they also introduce data transfer charges each time traffic crosses AZ boundaries. This is particularly problematic when applications have high-throughput, chatty services that are not affinity-aware, leading to excessive east-west traffic between zones. One way to keep those sneaky cross-AZ costs in check? Design your service layout carefully. That means leaning on pod affinity and anti-affinity rules to keep pods that rely on each other in the same availability zone whenever possible. It's a simple move, but it can make a real difference [2][5].

Something else that tends to catch teams off guard is the way load balancers and ingress controllers get used. With EKS, it's pretty standard to rely on AWS Application Load Balancers (ALBs) or Network Load Balancers (NLBs) to open services up to the outside world. The catch? These load balancers rack up both hourly charges and per-GB data transfer fees. If each service gets its own ALB, those costs can add up fast. A smarter move is to use a single ingress controller tied to one load balancer and then route traffic to different services using path- or host-based rules. It's a clean way to keep everything functional while trimming down on how many ALBs you actually need [4][7].

Using VPC endpoints and PrivateLink is another smart move when it comes to optimizing network traffic. Rather than sending data out over the public internet, these tools let services talk to each other privately within AWS—which can boost both security and efficiency. This not only enhances security but also reduces data transfer costs associated with traversing public IP space. For Kubernetes clusters that integrate with AWS-managed services like S3, DynamoDB, or RDS, enabling interface endpoints ensures that communication remains within the AWS network and is billed at intra-VPC rates rather than internet gateway rates, resulting in substantial savings over time [2][4].

Service meshes, such as AWS App Mesh, also influence networking costs. By inserting a control plane and data plane for traffic routing, service meshes enable sophisticated observability and failover mechanisms. However, they also introduce additional network hops and traffic inspection layers, which can elevate data transfer volumes. Proper configuration of sidecar proxies, traffic routing policies, and circuit breakers is necessary to maintain the balance between operational control and cost. In multi-region deployments, App Mesh can be integrated with Route 53 to create an active-active architecture, but this model demands careful attention to cross-region data flows which are billed at higher rates than intra-region traffic [5][3].

Observability tools play an important role in identifying and mitigating networking inefficiencies. CloudWatch metrics, VPC flow logs, and AWS Cost and Usage Reports (CUR) can provide detailed insights into the direction, volume, and cost of traffic. Tagging resources and correlating them with specific services or teams enables fine-grained attribution of networking costs, which is essential for implementing internal chargeback or showback models. As network patterns evolve with changes in application architecture, continuous monitoring is necessary to ensure that optimizations remain effective over time [4][7].

7. Monitoring, Governance & Observability

Monitoring and observability are critical to maintaining cost-effective and reliable Amazon EKS environments. As Kubernetes clusters grow in complexity and scale, organizations must develop robust visibility frameworks to track application performance, cluster health, and resource consumption. Without precise observability, over-provisioning or inefficient utilization can go unnoticed, resulting in spiraling costs and degraded performance. Amazon EKS provides integration with a variety of AWS-native and open-source observability tools that empower teams to monitor, debug, and optimize their clusters in real time [3][4].

A central component of EKS observability is Amazon CloudWatch**,** which offers metrics, logs, and alarms for cluster components and application-level events. Metrics such as CPU utilization, memory pressure, pod restart counts, and node availability can be visualized through dashboards, enabling quick detection of anomalies. EKS integrates natively with CloudWatch Container Insights, which aggregates logs and metrics from pods, services, and containers for detailed workload analysis. However, effective use of CloudWatch requires thoughtful configuration to avoid incurring high costs associated with excessive metric ingestion and log retention. By selectively enabling only essential metrics and controlling log granularity, organizations can strike a balance between visibility and affordability [6][4].

Complementing CloudWatch are open-source tools like Prometheus and Grafana, which are frequently used for collecting and visualizing Kubernetes metrics. Prometheus scrapes data from Kubernetes endpoints and stores it in a time-series database, while Grafana provides customizable dashboards for visualizing trends over time. You can roll these tools out inside your cluster using Helm charts, and they tend to offer more flexible, fine-grained monitoring than the default AWS options. And the best part? With the right setup—like smart data retention policies and federation—they can be a much more budget-friendly alternative to fully managed observability platforms [3][6].

Governance in EKS isn't just about putting up walls—it's more about visibility and control. Who's touching what? Where are they making changes? And is everything still playing by the rules? AWS IAM and Kubernetes RBAC kind of team up here to shape those access boundaries. You might, for example, give developers full reign in dev but lock them down to read-only in prod—just enough freedom without risking chaos. It's flexible—but only if it's set up right. And for visibility? CloudTrail logs every API move. Then, for broader guardrails—especially across multiple accounts or clusters—you can lean on Config Rules or Service Control Policies. It's not glamorous, but it keeps everything running cleanly [6][2].

If you're serious about keeping cloud spend under control, then yeah—things like AWS Cost and Usage Reports (CUR) and proper tagging aren't optional. CUR gives you all the gritty details, even hour by hour, so it's super useful for catching weird cost jumps or spotting slow creeps over time. Kubernetes does offer tagging through labels and annotations, but here's the snag: unless you map those to AWS tags, they won't show up in CUR. And doing that manually? Not fun. That's where tools like kube-cost—or services like PerfectScale—really pull their weight. They handle the grunt work and can crank out chargeback reports, so your engineers finally get a clear picture of what their code's actually costing.

Service meshes aren't just about managing traffic—they also play a big role in observability. Whether you're using AWS App Mesh or open-source options like Istio, these tools work by injecting sidecar proxies into each pod. That setup makes it possible

to trace requests as they move across services, giving you a clear end-to-end view. You can track latency, see which calls succeed or fail, and measure how things are stacking up against your service-level objectives (SLOs). However, the added telemetry comes at the cost of increased CPU, memory, and network usage, so it is important to monitor the overhead and fine-tune collection settings to avoid unnecessary expenses [5][3].

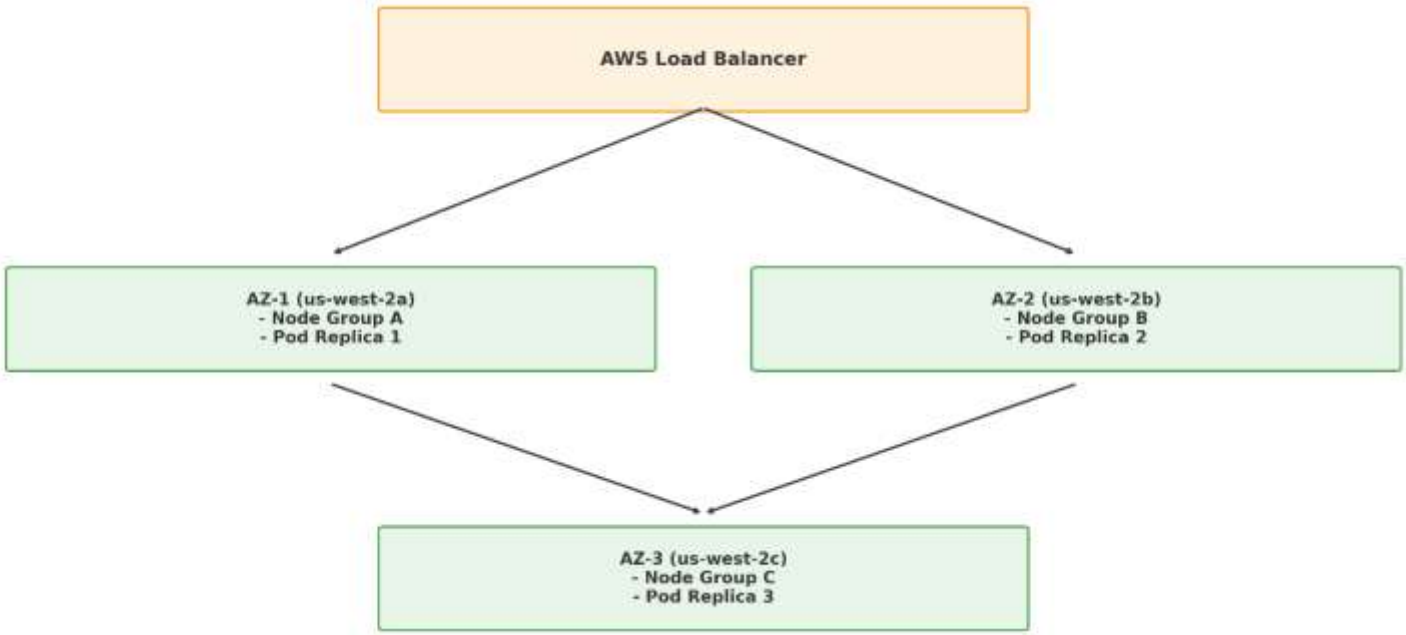
8. Best Practices for High Availability and Resilience

High availability (HA) and resilience are critical design goals for applications deployed in Amazon EKS, particularly those with strict uptime requirements or global user bases. In Kubernetes-based architectures, failure is considered a normal event, and infrastructure must be architected to gracefully tolerate and recover from disruptions. EKS provides several tools and configuration options to support fault-tolerant deployments, but their effective use requires deliberate planning around resource distribution, health checks, deployment strategies, and redundancy across availability zones. Implementing HA and resilience strategies not only ensures reliability but also improves the end-user experience and protects against revenue loss from downtime [1][5].

One of the foundational principles for achieving high availability in EKS is distributing resources across multiple Availability Zones (AZs) within a region. This involves configuring worker nodes and critical workloads—such as front-end services, API layers, and databases—to be deployed across at least two or more AZs. Doing so minimizes the risk of a single point of failure; if one AZ experiences an outage, traffic can automatically be rerouted to healthy nodes in another AZ. Kubernetes supports this through topology-aware scheduling and node affinity rules that guide the placement of pods to ensure even distribution across zones [1][4].

This diagram from the AWS docs lays out what a multi-AZ Kubernetes deployment looks like in EKS. It shows a web server app running across three different Availability Zones, with each zone hosting its own replica of the pod—all sitting behind a load balancer. The real win here? Resilience. Say one AZ drops—yeah, it's not ideal, but it's also not a disaster. The load balancer just shrugs and sends traffic to whatever's still running. No panic, no downtime. The app just keeps doing its thing.

Figure: High Availability Deployment of Kubernetes Pods Across Availability Zones (Amazon Web Services, 2025) [1].



On top of that zonal setup, Kubernetes throws in a few extra safety nets—like Pod Disruption Budgets (PDBs) and readiness probes—to help keep things steady during updates or maintenance. PDBs are basically there to make sure you don't pull too many pods offline at once. They draw a line: "this many need to stay up," which helps avoid surprise outages. And then you've got readiness and liveness probes. These are like little health checks—telling Kubernetes if a pod's ready to handle traffic or if it's flaking out and might need a reboot. Put together, they soften the blow of crashes, config issues, or rolling deployments, so things stay more or less stable even when stuff shifts around [1][6].

How you roll out updates matters just as much as what you're rolling out—deployment strategy plays a big role in keeping things resilient. Approaches like blue/green deployments or canary releases help lower the risk of shipping bad code into production. With blue/green, you spin up a new version right next to the live one, but you don't flip traffic over until it clears all the health checks. If something goes sideways, you can instantly roll back to the old version—no drama. Canary deployments take a slower path, easing the new version out to a small group of users first, just to make sure everything behaves. When tied into a solid CI/CD pipeline in EKS, these strategies offer a much safer, more controlled way to push changes without risking a full-blown outage [6][4].

For multi-region setups, resilience gets kicked up a notch with an active-active architecture—especially when you pair AWS App Mesh with Route 53. In this model, you've got clusters in different regions all serving traffic at the same time. Route 53 runs health checks on each region's endpoints and, if something goes down, it automatically reroutes traffic to the healthy regions. Sure, cross-region deployments come with added complexity—and yes, higher data transfer costs—but for mission-critical applications that can't afford a second of downtime, this kind of setup delivers the strongest layer of failover protection you can get [5][2].

9. Conclusion and Future Outlook

Running containerized apps on Amazon EKS can seriously level up your operations—it gives teams a solid foundation to scale well, stay agile, and get a better grip on cloud costs. That said, just lifting and shifting workloads to a managed Kubernetes service doesn't magically make all that happen. To really make it work, you need to understand the ins and outs of EKS—how it's built, how it bills, how you monitor and fine-tune it, and how to keep it resilient under pressure. This paper has unpacked those core areas, with a focus on why technical decisions shouldn't be made in a vacuum. If you want to get the most out of EKS, you've got to line up your infrastructure choices with your business priorities—especially when it comes to keeping things fast, available, and cost-efficient.

This analysis makes it clear—optimizing costs in EKS isn't a one-and-done task. It's a layered, ongoing effort with a lot of moving parts. It starts with understanding the economic implications of every architectural layer—compute, storage, control plane, and networking. Compute rightsizing continues to be one of the most effective ways to manage cloud costs. By selecting the right EC2 instance types, thoughtfully blending on-demand, reserved, and spot instances, and using autoscaling tools like HPA, VPA, and Cluster Autoscaler, teams can eliminate resource waste. Tools like Karpenter improve intelligent scaling by automatically deploying the most cost-effective infrastructure exactly when it's required.

At the same time, networking optimization—frequently underestimated—remains essential, as excessive inter-AZ traffic, poorly configured ingress, and unnecessary use of public-facing services can drive substantial operational costs. By designing architectures that minimize cross-AZ communication, consolidating ingress through shared load balancers, and using VPC endpoints to reduce public traffic charges, teams can significantly reduce their networking bills. Furthermore, observability tools and FinOps practices help maintain financial accountability. Resource tagging, CUR integration, and real-time cost dashboards allow organizations to tie consumption directly to engineering teams or application environments, fostering a culture of financial ownership in technical operations.

High availability and resilience are key components of a strong EKS environment. Keeping apps from wobbling—whether it's during a rollout or when something unexpectedly breaks—isn't just a matter of tossing more infrastructure at the problem. It takes some thoughtful engineering. That means spreading workloads across AZs, wiring in solid health checks, setting clear Pod Disruption Budgets, and rolling out changes carefully with blue/green or canary strategies so you're not flipping a switch and hoping for the best. And when it really matters—like with systems that can't go down—multi-region setups using App Mesh and Route 53 kick in. They've got your back even if an entire region takes a hit. Yeah, it's more complex, and yeah, it can cost more.But if an outage means lost revenue or puts your reputation on the line, those extra costs start to feel a lot more justifiable. At the end of the day, yeah—it's more moving parts, more to manage. But when uptime is non-negotiable, that complexity? It's a trade-off most teams are more than willing to make.

Looking ahead, EKS optimization is clearly heading in a more automated, intelligent direction. Platforms like Sedai are already using machine learning to spot usage patterns and make smart cost-saving tweaks—things like adjusting resource limits or fine-tuning scaling policies—without needing a human to step in. At the same time, AI-driven observability tools are sliding into everyday DevOps workflows, helping teams with real-time scaling predictions, anomaly spotting, and even cost forecasting. As the cloud-native world keeps evolving, we'll likely see more reliance on service meshes and GitOps-driven infrastructure, all aimed at making systems more consistent, resilient, and easier to manage.

Wrapping it all up—getting the most out of Amazon EKS isn't just about knowing the tech. It's about thinking strategically, staying mindful of costs, and making smart calls across every layer of your setup. When teams weave together solid practices around scaling, rightsizing, networking, observability, and governance, that's when Kubernetes on AWS really starts to pay off. And as the platform matures, we're heading toward a future where a lot of the heavy lifting—tuning, scaling, optimizing—just happens automatically. That's where cloud-native is going: smarter, faster, and a whole lot easier to manage.

9. References

[1] Amazon Web Services. (2025). *Running highly-available applications*. Retrieved from https://docs.aws.amazon.com/eks/latest/best-practices/application.html

[2] Amazon Web Services. (2025). *Cost Optimization - Networking*. Retrieved from https://docs.aws.amazon.com/eks/latest/best-practices/cost-opt-networking.html

[3] Harchandrasher, H. (2025). *How to Optimize for Cost in EKS: Best Practices for AWS Efficiency*. Sedai. https://www.sedai.io/blog/how-to-optimize-for-cost-in-eks-best-practices-for-aws-efficiency

[4] Duggal, T. (2025). *Amazon EKS Cost Optimization Best Practices*. PerfectScale by DoiT.

https://www.perfectscale.io/blog/eks-cost-optimization

[5] Khanna, A., & Thangaraj, N. (2022). *Run an active-active multi-region Kubernetes application with AppMesh and EKS*. Amazon Containers Blog. https://aws.amazon.com/blogs/containers/run-an-active-active-multi-region-kubernetes-application-with-appmesh-and-eks/

[6] Jackiewicz, M. (2025). *Practical guidelines for using Kubernetes on AWS EKS*. RST Software. https://www.rst.software/blog/practical-guidelines-for-using-kubernetes-on-aws-eks

[7] Atmosly. (2025). *Optimizing Amazon EKS Costs: A Comprehensive Guide to EKS Pricing*. Retrieved from https://www.atmosly.com/blog/optimizing-amazon-eks-costs

[8] Attaoui, W., Sabir, E., Elbiaze, H., & Guizani, M. (2023). *VNF and CNF Placement in 5G: Recent Advances and Future Trends*. IEEE Transactions on Network and Service Management, 20(4), 4698–4730. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=10090468