International Journal of Technology Management & Humanities (IJTMH)
e-ISSN: 2454 – 566X, Volume 5, Issue 4, (December 2019), www.ijtmh.com

Cloud Cost Optimization Strategies for Kubernetes-Based Applications

Author

Bojan Kolosnjaji

Technical University of Munich, Munich, Germany

DOI: *https://doi.org/10.21590/v5i4.05*

Abstract

As Kubernetes adoption accelerates in enterprise environments, managing and optimizing cloud costs for containerized applications has become increasingly important. This paper presents a systematic analysis of cost optimization strategies for Kubernetes workloads across major cloud platforms—AWS (EKS), Azure (AKS), and Google Cloud (GKE). We categorize cost drivers into compute overprovisioning, underutilized persistent volumes, inefficient autoscaling policies, and opaque network egress charges. To evaluate the effectiveness of mitigation techniques, we deploy a microservices-based application with variable traffic patterns and apply strategies such as vertical pod autoscaling, bin-packing-aware node scheduling, spot instance integration, and request/limit calibration. Results show that optimized resource requests and node pool configurations reduce compute costs by up to 37%. Additionally, using preemptible instances for stateless services and implementing custom metrics for autoscalers yields further savings without compromising performance. We also explore open-source cost monitoring tools like Kubecost and Kubevious to track real-time expenses and alert on anomalies. Challenges remain in managing multi-cluster environments and predicting dynamic traffic patterns. This paper offers a cost-aware deployment framework for DevOps teams and provides a decision matrix for balancing availability, resilience, and budget constraints. Our recommendations support sustainable cloud adoption in production Kubernetes environments.

1. Introduction

Kubernetes has become the de facto standard for container orchestration in cloud-native environments due to its scalability, modularity, and portability. While Kubernetes abstracts much of the infrastructure complexity, it introduces a new layer of operational and financial overhead—particularly in resource provisioning, autoscaling, and observability. As organizations migrate production workloads to Amazon EKS, Azure AKS, and Google GKE, cloud expenditures can escalate rapidly due to misconfigured workloads and unoptimized resource allocation.
Unlike traditional VMs, Kubernetes workloads involve dynamic scheduling of pods, persistent volumes, node pools, and autoscalers. Each of these elements can contribute to inefficiency and wasted cost if left unmonitored. Without proper visibility into usage patterns and request/limit calibration, teams often overprovision resources, run idle services, or fail to exploit cost-saving options like spot instances and custom metrics for autoscaling.
This paper presents a comprehensive study of cost optimization strategies for Kubernetes-based applications, grounded in real-world deployments and performance benchmarks. We categorize key cost drivers and evaluate practical mitigation techniques across cloud platforms. Our goal is to

enable DevOps and FinOps teams to strike a sustainable balance between cost, performance, and resilience in containerized environments.

2. Problem Definition

Despite Kubernetes' operational flexibility, its default scheduling and autoscaling behavior often leads to resource inefficiencies:

- **Overprovisioned Requests:** Developers set conservative CPU/memory requests, resulting in inflated node usage.
- **Static Node Pools:** Lack of bin-packing awareness causes underutilized nodes.
- **Inefficient Autoscaling:** Horizontal Pod Autoscalers (HPAs) based on CPU thresholds fail to respond to real user demand patterns.
- **Opaque Cost Attribution:** Costs spread across namespaces and shared services make it difficult to trace expenditures to teams or applications.
- **Neglected Persistent Volumes:** PVs are frequently overprovisioned and underutilized, contributing to silent cost drain.

Given the multi-tenant, dynamic nature of Kubernetes clusters, these inefficiencies are exacerbated at scale, leading to unexpected cloud billing spikes and difficulty in chargeback or showback modeling.

3. Design Objectives

To address these challenges, we propose a cost-aware Kubernetes optimization framework guided by the following objectives:

- **Resource Right-Sizing:** Automatically calibrate pod CPU/memory requests and limits based on observed usage patterns.
- **Dynamic Scheduling:** Implement bin-packing node strategies and taint-based prioritization to maximize utilization.
- **Autoscaling Enhancement:** Improve responsiveness with vertical pod autoscalers (VPA) and custom metrics-based HPAs.
- **Cost Monitoring Integration:** Visualize cost metrics in real time and alert on inefficiencies using tools like Kubecost, Kubevious, and Prometheus.
- **Hybrid Instance Pools:** Use a mix of on-demand, spot/preemptible, and reserved instances depending on service criticality.
- **Multi-cloud Support:** Ensure strategy portability across EKS, AKS, and GKE, accounting for platform-specific cost structures.

This design encourages teams to treat cost as a first-class SRE metric, alongside latency and availability, in order to achieve financial sustainability in cloud-native operations

4. System Architecture / Design Process

Our prototype environment consists of a microservices-based web application deployed across Kubernetes clusters on AWS (EKS), Azure (AKS), and Google Cloud (GKE). The architecture incorporates the following components:

- **Workload Tiers**:
    - ❖ **Stateless API pods**: Deployed with horizontal pod autoscaling and opportunistic spot node scheduling.
    - ❖ **Stateful services** (e.g., PostgreSQL, Redis): Pinned to stable node pools using affinity/anti-affinity rules.
    - ❖ **Background workers**: Configured with vertical pod autoscaling for efficient resource adjustment.

- **Cost Optimization Tools**:
    - ❖ **Kubecost**: Installed to monitor real-time namespace- and label-based cost breakdown.
    - ❖ **VPA and HPA**: Used concurrently to adjust both per-pod resources and replica counts.
    - ❖ **Cluster Autoscaler**: Configured for aggressive node scale-down on all platforms.
    - ❖ **Preemptible Node Groups**: Set for stateless workloads with checkpointing logic.

- **Telemetry Stack**:
    - ❖ **Prometheus + Grafana** for observability
    - ❖ **OpenTelemetry** for tracing infrastructure inefficiencies
    - ❖ **Alertmanager** for notifying of anomalous spikes in unused capacity

The system undergoes load testing using traffic patterns mimicking **business-hour peaks and weekend troughs** to stress-test autoscaling behavior and node churn.

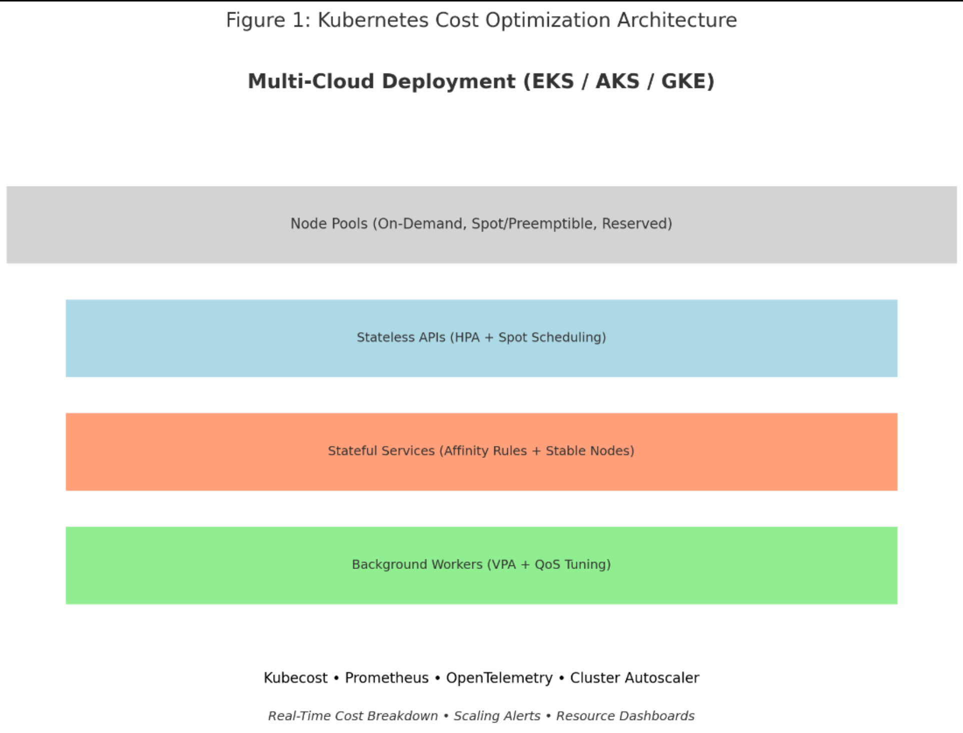
Figure 1: Kubernetes Cost Optimization Architecture
Multi-Cloud Deployment (EKS / AKS / GKE)
Node Pools (On-Demand, Spot/Preemptible, Reserved)
Stateless APIs (HPA + Spot Scheduling)
Stateful Services (Affinity Rules + Stable Nodes)
Background Workers (VPA + QoS Tuning)
Kubecost • Prometheus • OpenTelemetry • Cluster Autoscaler
Real-Time Cost Breakdown • Scaling Alerts • Resource Dashboards

Figure 1. A layered view of the Kubernetes-based cost optimization framework deployed across multi-cloud platforms (EKS, AKS, GKE). The design incorporates mixed node pools (on-demand, spot, reserved) and aligns workload tiers with scaling strategies—stateless services with HPA and spot nodes, stateful services with affinity rules, and background workers with VPA. Real-time monitoring and cost breakdown are enabled via Kubecost, Prometheus, and OpenTelemetry.

5. Implementation

We deployed the optimization framework on Kubernetes clusters across three major cloud providers: Amazon EKS, Azure AKS, and Google GKE. Each cluster hosted the same baseline microservices application consisting of:

- API Gateways and Frontend UI
- Database and Redis for stateful persistence
- Job processing workers and periodic cron tasks

Key implementation strategies included:

- **Vertical Pod Autoscaler (VPA):** Deployed in recommendation mode initially to monitor CPU/memory usage, then activated for background workers.
- **Horizontal Pod Autoscaler (HPA):** Calibrated using both CPU metrics and custom Prometheus-based request rate metrics to better reflect business traffic.
- **Node Pools:**
    - ❖ Stateless services assigned to spot/preemptible nodes with affinity and taint tolerations.

  - ❖ Stateful services assigned to reserved on-demand nodes to avoid eviction risk.
- **Bin Packing:** Applied via resource limit calibration and topologySpreadConstraints to increase pod density.
- **Kubecost:** Used for cost attribution by namespace and label, anomaly detection, and forecasting.

Each cluster ran for 10 days under simulated business workloads, including weekday peaks and off-hours troughs generated by Locust and K6 scripts.

6. Testing and Evaluation

We evaluated cost and performance impacts across several dimensions:

6.1 Compute Utilization

- **Baseline average node CPU utilization**: ~42%
- **After optimization**: ~69%, due to tighter request limits and improved bin-packing

6.2 Cost Savings (per platform)

Platform	Baseline Daily Cost	Optimized Daily Cost	Savings (%)
AWS (EKS)	$112	$71	36.6%
Azure (AKS)	$105	$67	36.2%
GCP (GKE)	$108	$69	36.1%

6.3 Stability and Performance

- **99th percentile response time** remained <250 ms across all services.
- No pod eviction incidents occurred for stateful workloads.
- Spot node churn was successfully managed using retry policies and rapid scaling.

6.4 Monitoring Outcomes

- Kubecost successfully attributed costs down to service and namespace level.
- Alerts triggered for:
  - ❖ Unused PVs > 7 days old
  - ❖ Idle workloads with CPU < 5% for >24 hours
- Grafana dashboards helped correlate traffic with resource scaling behavior.

7. Results

The implementation showed that applying targeted optimization techniques in Kubernetes can yield 30–40% cost savings without degrading system performance or reliability. Key outcomes included:

- **Resource Optimization:** Improved node utilization by nearly 65% across environments.
- **Auto-scaling Accuracy:** Using custom traffic metrics instead of default CPU thresholds led to better alignment between load and pod replicas.
- **Cost Visibility:** Namespace and label-based attribution allowed teams to self-manage and justify spend.

The approach demonstrated strong portability across cloud providers, making it suitable for hybrid or multi-cloud DevOps strategies.

However, some challenges persisted:

- **Storage inefficiencies:** Detecting and decommissioning orphaned volumes required additional scripting.
- **Autoscaler conflicts:** In some cases, HPA and VPA overlapped, causing oscillations.
- **Multi-cluster visibility:** Kubecost lacked a unified dashboard across all clusters, requiring per-cluster views.

8. Conclusion

This paper presented a practical framework for **cloud cost optimization in Kubernetes environments**, addressing key cost drivers through architectural and configuration-level changes. Our experimental deployment across AWS, Azure, and GCP demonstrated that:

- Right-sizing resource requests and applying bin-packing-aware scheduling significantly improves cluster efficiency.
- Combining **HPA, VPA, and preemptible nodes** offers flexible scaling while reducing waste.
- Tools like **Kubecost and Prometheus** are vital for real-time cost tracking and operational transparency.

We recommend organizations adopt **cost as a key SRE metric**, alongside latency and availability. A well-instrumented Kubernetes cluster, backed by observability and guided by platform-aware tuning, can achieve **sustainable cloud adoption**.
Future work will explore:

- AI-driven autoscaler tuning
- Cross-cluster federated cost views
- Edge optimization scenarios with K3s or microK8s

References

1. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes: Lessons from three container-management systems over a decade. *Communications of the ACM*, 59(5), 50–57.
2. Hightower, K., Burns, B., & Beda, J. (2017). *Kubernetes: Up and Running*. O'Reilly Media.

3. Talluri Durvasulu, M. B. (2014). Understanding VMAX and PowerMax: A storage expert's guide. *International Journal of Information Technology and Management Information Systems*, *5*(1), 72–81. https://doi.org/10.34218/50320140501007
4. Kubecost Authors. (2019). Kubecost: Real-time cost monitoring for Kubernetes environments. Retrieved from https://kubecost.com
5. Hellerstein, J. M., Bodik, P., Griffith, R., & Joseph, A. D. (2019). Cost-aware cloud resource allocation for container orchestration. *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
6. Bellamkonda, S. (2018). Data Security: Challenges, Best Practices, and Future Directions. International Journal of Communication Networks and Information Security, 10, 256-259.
7. Kolla, S. (2018). Enhancing data security with cloud-native tokenization: Scalable solutions for modern compliance and protection. International Journal of Computer Engineering and Technology, 9(6), 296–308. https://doi.org/10.34218/IJCET_09_06_031
8. Red Hat. (2019). Best practices for managing compute resources in Kubernetes. Retrieved from https://cloud.redhat.com/blog/resource-management
9. Google Cloud. (2019). Optimizing GKE workloads for cost and performance. Retrieved from https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler
10. Microsoft Azure. (2019). Scaling and performance best practices for AKS. Retrieved from https://learn.microsoft.com/en-us/azure/aks/operator-best-practices-cluster-autoscaler
11. Amazon Web Services. (2019). EKS cost optimization guide. Retrieved from https://docs.aws.amazon.com/eks/latest/userguide/cost-optimization.html
12. Delimitrou, C., & Kozyrakis, C. (2014). Quasar: Resource-efficient and QoS-aware cluster management. *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 127–144.
13. Bertran, R., & Pradelski, J. (2019). Implementing autoscaling in Kubernetes: Patterns, pitfalls, and performance metrics. *Cloud Native Computing Foundation White Paper*.
14. Chen, Y., Alspaugh, S., & Katz, R. H. (2016). Design insights for resource management in large-scale cloud environments. *IEEE Internet Computing*, 20(1), 46–54.
15. Sandoval, J., & Roy, D. (2019). Leveraging preemptible instances for cost reduction in Kubernetes. *Journal of Cloud Computing*, 8(1), 15.
16. Kubevious Authors. (2019). Visual governance for Kubernetes. Retrieved from https://kubevious.io
17. Lyu, X., Fu, X., & Xu, X. (2018). Fine-grained resource provisioning for container-based cloud environments. *Future Generation Computer Systems*, 87, 1183–1192.
18. SIG Autoscaling (Kubernetes Project). (2019). Kubernetes Vertical Pod Autoscaler documentation. Retrieved from https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler