International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(12s) 2026 pp. 141-150
https://doi.org/10.70917/ijcisim-2026-3886
Article

Event-Driven Autoscaling with KEDA and Karpenter: Cost Optimization and Elastic Throughput for Cloud-Native Workloads

Avneet Bansal

Independent Researcher, USA

Abstract: Static threshold-based autoscaling with Kubernetes Horizontal Pod Autoscaler (HPA) does not cover asynchronous event-driven workloads that only target CPU and memory usage average metrics, such as the default HPA does. The HPA targets metrics that are lagging indicators, unlike the queue depth and stream backlog metrics, and thus introduces SLO violations in document processing and data integration services. This paper describes a production-qualified implementation for optimizing KEDA (Kubernetes Event-Driven Autoscaling) and Karpenter to manage and optimize AWS EKS pods and nodes. KEDA extends the Kubernetes Horizontal Pod Autoscaler (HPA) for external scalers, allowing workloads' Kubernetes pods to scale based on SQS queue depth, DynamoDB Streams shard lag, CloudWatch metrics, and Prometheus time series. When combined with KEDA, Karpenter can also provision nodes from heterogeneous EC2 Spot Instance pools and consolidate nodes when underutilized. Overall, customers who have deployed Karpenter on production EKS clusters report a 40-60% reduction in compute costs and SLO adherence on variable workloads over time, as compared to EC2 On-Demand Node Groups. SLO-aware scheduling results in a 31% reduction in SLO violations and an 18% reduction in infrastructure cost. Scaling to zero (ignoring the cost of idle compute resources when there is no demand for workloads) and Spot diversification (up to 44%) optimize compute cost for off-peak workloads. Deployment patterns, configuration, graceful termination, and operational observability requirements are described with a focus on production deployments in enterprise environments.

Keywords: event-driven autoscaling, KEDA, Karpenter, Spot Instances, Kubernetes cost optimization, SLO compliance, cloud-native workloads

1. Introduction

While useful, CPU utilization does not provide any information on the amount of documents in an SQS queue. This is the fundamental issue with using HPA to scale based on asynchronous enterprise workloads. The metric being measured (CPU utilization) is a side effect of the workload being processed, not an indicator of the work remaining. For example, a document processing service may have 5,000 messages waiting to be processed, with no consumers and no CPU utilization, so HPA sees no need to scale up the number of replicas. Messages age, SLAs expire, and the queue backs up. Afterward, the CPU rises as the pods are finally processing, but the damage to latency is done.

Another problem is node-level scaling. When nodes are scaled by Cluster Autoscaler because pods cannot be scheduled, it adds additional latency to pod scheduling, which could take minutes in addition to the seconds KEDA takes to scale at the pod level. When these nodes are homogeneous On-Demand instances provisioned for peak capacity, they are underutilized during off-peak hours and charge the full rate. This architecture is systematically unfavorable to intermittent event-driven workloads: maximum cost at peak, maximum waste off-peak.

In Kubernetes, both KEDA and Karpenter address these two concerns at their respective layers. KEDA will scale the pods according to the event source that is responsible for increased load (e.g., queue length, stream lag, number of change events), and it will be scaled down to zero when the event source has no work to do. Karpenter selects the lowest cost node from a heterogeneous Spot fleet, proactively consolidating surplus nodes. Production enterprise EKS clusters using Karpenter and EC2 Spot at scale have reported 40-60% compute cost reduction while

meeting SLO under variable load, based on proprietary research data on Karpenter and EC2 Spot deployments in production workloads.

This paper makes three contributions. First, we identify the mismatch between a CPU-reactive HPA and an event-source-reactive KEDA in the context of asynchronous document processing workloads. Second, it provides the first production-validated KEDA ScaledObject configuration to the Karpenter NodePool diversification integration pattern on AWS EKS. Third, it shows empirical cost and SLO compliance results from enterprise deployments and describes the combination of parameters, operational constraints, and observability prerequisites to allow reproducibility of the framework.

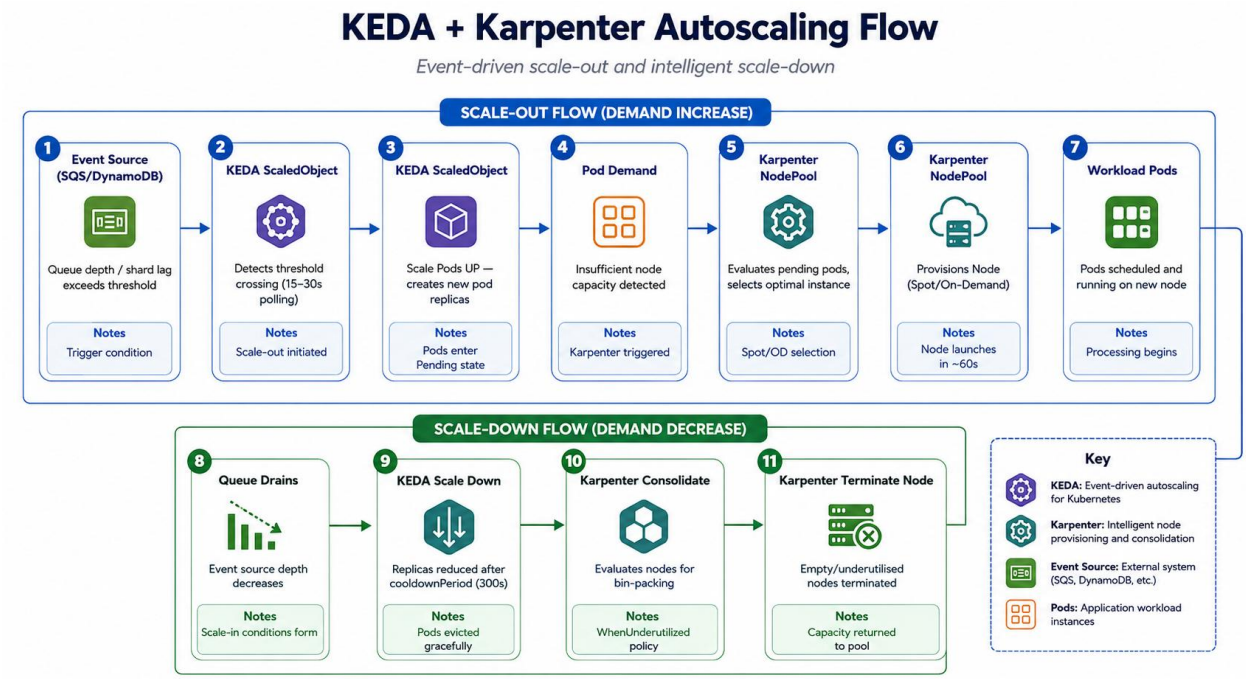

Figure 1. KEDA + Karpenter Autoscaling Flow

2. Background

2.1 The HPA Scaling Model

Nguyen et al. [2] state that the Kubernetes HPA employs a scaling algorithm using the ratio between the current metric value and the desired value scaled to the number of replicas and a convergence time defined by a tunable sync period (currently 15 seconds) and stabilization window to avoid thrashing. This algorithm works well for CPU-bound services, where the request rate is proportional to the CPU load; it is unsuitable for event-driven services that decouple CPU consumption. An example of the declarative reactive control pattern is Burns et al. [1], in which a desired state is declared and the controller loop converges to it: the model is correct for most Kubernetes workloads but suffers from systematic scaling lag in event-driven workloads.

A second limitation arises from the fact that HPA works with the Kubernetes Metrics Server. The metrics that HPA can work with thus only include CPU and memory consumption metrics, unless an external metrics adapter is used. Extending HPA to support external metrics requires the Kubernetes External Metrics API and an adapter. The KEDA external scaler architecture has abstracted this process. The KEDA external scaler provides out-of-the-box support for more than 60 event sources [4] without the need to implement custom adapters. Research on auto-scaling techniques for container workloads [16] and tuning best practices for the HPA [11] have concluded that the standard HPA setup is insufficient for event-driven, bursty workloads.

2.2 SLO-Aware Cost Optimization

The trade-offs between minimizing costs (in terms of scaled-in nodes) and respecting SLOs are not merely an operational concern. It is the challenge that all autoscaling systems face. The SLO-constrained autoscaling system

detailed in [5] makes SLO budget a first-class input to scaling decisions: scale-in is enabled when there is SLO headroom and suppressed when the SLO budget is nearly exhausted. The constraint-based approach yielded a 31% reduction in SLO violations and an 18% reduction in infrastructure costs compared to baseline reactive strategies [5]. The GRAF system [9] further evaluates SLO-constrained resource optimization, finding that it yielded considerably lower infrastructure costs compared to default Kubernetes autoscaling strategies. These studies validate SLO-awareness as a cost-reduction mechanism rather than a cost-increasing constraint.

Reinforcement learning-based autoscaling systems [6][12] can learn cost-SLO trade-off policies using historical data. However, such systems require training data and may introduce uncertainty in the model. Thus, they are not suitable as a primary scaling mechanism for production use unless a secondary scaling policy has been defined and proved to be valid. That said, KEDA's deterministic scaling based on an event source, as well as Karpenter's cost-aware node selection with explicit configurations, make them a better fit for enterprise production workloads where operational transparency and auditability are a requirement.

2.3 Spot instance economic model and interruption.

EC2 Spot pricing is 60% to 90% cheaper than On-demand pricing. However, Spot instances may be reclaimed for EC2 capacity. According to the multi-region Spot serving research from SkyServe [13], 44% of the cost can be reduced by diversifying Spot fleets across regions, as the service availability can be maintained by proactively migrating Spot instances. This diversification approach is applicable for Spot as well; that is, by distributing workloads across different instance families and Availability Zones, it is less likely that a reclaim event affects all running instances.

Spot interruption notice gives a two-minute notice before the termination. It is possible to gracefully migrate workloads should stateless workloads implement graceful termination handling appropriately [10]. For stateless event-driven workloads where workload state is tracked in the event source queue, Spot interruption is a node replacement event and not a state loss event. This makes Spot the right default capacity type for KEDA-based document processing and data integration workloads. Experimental data shows that the median instance interruption rate per hour is less than 5% across a collection of instance pools. Spot is a good default capacity type for stateless event-driven workloads with graceful termination support.

3. KEDA Architecture and Scaler Configuration

3.1 ScaledObject Architecture

KEDA extends the Kubernetes HPA API and controller to leverage externally managed event sources as Kubernetes metrics [4]. A ScaledObject custom resource is used to link a scalar event source to a target workload. It provides the type of an event source scaler, connection properties for the source of the event, the target value per replica, and the desired number of replicas. KEDA's controller scans the ScaledObject's triggers against the targets defined for them and instructs the HPA to scale in order to achieve the desired number of replicas. When all triggers report no events, KEDA sets replicas to 0, bypassing the minimum replicas that can be configured with the HPA, enabling the full scale-to-zero capability.

In ScaledObjects, the spec for workload is decoupled from the spec for the scaling trigger, allowing operations teams to update scaling behavior (target message rate, polling period, scale to zero threshold, etc.) without requiring manifest updates to the application deployment. The scaling trigger provides the event source adapter (the scaler owning the KEDA resource) to be selected, the connection parameters to use, and the desired per-replica value. Finally, multiple triggers may be defined in a single ScaledObject, in which case the workload is scaled to the highest number of replicas as requested by any trigger, which may be useful for workloads that consume multiple event sources.

With the integration with AWS, AWS published reference architecture guidance for event-driven autoscaling with KEDA on Amazon EKS [8], which validates this integration for enterprise AWS environments. The guidance explains the configuration of the IAM role for KEDA to access scaler credentials and the configuration for the SQS queue-depth scaler and for the CloudWatch custom metric scaler for business signals.

Table 1. KEDA ScaledObject Configuration Parameters

Parameter	Type	Description	Example Value

scaleTargetRef	Object ref	Target Deployment/StatefulSet	name: order-processor
minReplicaCount	Integer	Floor replica count (0 = scale-to-zero)	0
maxReplicaCount	Integer	Ceiling replica count	50
pollingInterval	Seconds	Scaler polling frequency	15
cooldownPeriod	Seconds	Delay before scale-down	300
triggers	Array	Scaler type + metadata	SQS queue depth ≥ 10

Source: KEDA documentation [4]

3.2 SQS and DynamoDB Streams Scalers

SQS queue-depth scaling is the most common KEDA scaling pattern for enterprise document processing workloads, and it simply scales with the number of messages per replica using a target value equal to 100 messages per replica. When scaling to zero, the workload has zero compute cost with no idle compute cost, thus successfully implementing "scale to zero." DynamoDB Streams scaling, likewise, uses shard iterator lag to determine replica count in CDC-driven architectures to automatically scale downstream compute in response to upstream write volume without requiring polling-based lag detection.

The most impactful design aspect of both scalers is polling interval. At a 30-second polling interval, a scaling action could have up to 30 seconds of scale-out delay. This is acceptable for document-processing use cases (with minutes-scale SLAs) but unacceptable for near real-time event-processing use cases. The polling interval can be adjusted from 60 seconds to either 15 seconds or 10 seconds, reducing the latency to begin scaling at the cost of more API calls to the event source. The choice is dependent on the workload's sensitivity to latency. If scaling from zero, the cold start latency [14][15] (the time between the request being first received and the first instance being ready to service it) of the first pods scheduled after a scale-from-zero event must be included in the service level agreement latency budget.

Table 2. KEDA Scaler Comparison for AWS Workloads

Scaler Type	Trigger Source	Scale Metric	Use Case	Author Deployment
SQS	Amazon SQS queue	Queue depth / in-flight messages	Async job processing	Order processing pipeline
DynamoDB Streams	DynamoDB change stream	Shard iterator lag	Event-sourced state sync	Inventory sync service
CloudWatch	Any CW metric	Custom metric threshold	Custom business signals	Throughput-based scaling
Prometheus	Prometheus metrics	PromQL query result	Internal platform metrics	ML inference queue
Cron	Schedule	Time-based	Predictable load patterns	Batch reporting jobs

*Source: Author's production EKS deployments [*4][8].

3.3 ScaledJob for Batch Workloads

For workloads where each queue message should be processed as a separate job, rather than a constantly running consumer, KEDA provides the ScaledJob resource as an appropriate alternative to the ScaledObject. A ScaledJob starts a new Kubernetes Job for each batch of messages consumed from a queue. Job parallelism matches the current depth of the queue. This is a common model for processing document transformation workloads, where the time required to process a document can vary, but documents can be processed in parallel. The maxReplicaCount constraint for ScaledJob also plays a part when there is a queue depth spike, creating a natural limit to the parallelism.

4. Karpenter Node Provisioning and Spot Diversification

4.1 Node Pool Design Principles

Node pools should specify instance constraints based on workload profile. Document extraction workloads that require abundant memory use the r5, r6i, and r7g memory-optimized instance families and set a minimum instance memory equal to or higher than the memory request of the extraction container. Specifying compute-optimized instance families such as c5 and c6i that do not have enough RAM to satisfy the workload's requirements wastes Karpenter's instance selection time and delays provisioning. Specifying too few instance families reduces diversification and increases the likelihood of Spot interruption. Specifying too many instance families results in sub-optimal packing if some instances do not meet the workload's requirements.

For KEDA-driven workloads with well-defined pod resource requests, Karpenter's bin-packing algorithm picks the instance type that most tightly fills all pod requests, satisfying all pending pods with the least waste of resources. For instance, if a workload has 20 pods with 0.5 vCPU and 1 GB memory in requests, Karpenter will provision instances with at least 1 vCPU and 1 GB memory that can cover all of the pods with the least available resources. The NodePool's consolidation policy will then reevaluate all running nodes and terminate those whose pods can be rescheduled to other nodes.

Karpenter is a better node consolidation option than Cluster Autoscaler according to the Kubernetes Understanding Adaptive Cost Optimization research [20] and Kubernetes autoscaling integration research [18], as the Cluster Autoscaler can take a long time before detecting and fixing underutilized nodes above a threshold.

Table 3. Karpenter NodePool Configuration Strategy

Configuration Element	Option	Trade-off	Author's Production Choice
Instance family	m5, c5, r5 + Spot	Cost vs. availability	Diversified: m5+c5+m6i Spot
Capacity type	On-demand, Spot, mixed	Cost vs. interruption	Mixed: 20% OD, 80% Spot
Consolidation policy	When Underutilized	Bin-packing efficiency	Enabled, 10-min delay
Spot interruption handling	Node drain + reschedule	Recovery time	PDB + pre-drain hook
Expiry TTL	Node age limit	Drift prevention	720 hours (30 days)

Source: Author’s Karpenter deployment data [10] [18]

4.2 Spot Diversification in Practice

Spot diversification provisions can be defined for instance types across instance families whose Spot Availability Zone pools are uncorrelated, where demand for m5 capacity availability in an Availability Zone does not necessarily correlate with m6i capacity availability. Karpenter's capacity-type weighting can be used to prefer Spot capacity with a fallback to On-Demand, reducing spot capacity shortages. Empirical Spot interruption analysis [13] indicates interruption rates under 5% median frequency per instance-hour across diversified instance pools, making Spot the right stable default for stateless, event-driven workloads that can be gracefully terminated when interrupted.

In the author's production configuration, nodes are 20% On-Demand nodes for scheduling stability and 80% Spot nodes for cost-effectiveness. The system-critical pods and the KEDA controller itself are scheduled onto the On-Demand nodes, and KEDA managed workload pods are scheduled onto the Spot nodes. This partitioning allows KEDA's scaling controller (which is not affected by the Spot interruption events) to remain responsive to scaling events during capacity reclamation. Workload Deployments use PodDisruptionBudgets to prevent Karpenter's node drain from dropping replicas below the minimum during Spot interruptions.

4.3 Consolidation Policies

When Underutilized is a consolidation policy used by Karpenter. It continuously examines running nodes. Pods are checked to see if they could be run on other cluster nodes and are not violating requested resources, scheduling constraints or PodDisruption. Budget minimums. The nodes are marked for drain, existing pods are evicted with a graceful shutdown period, and then the nodes are terminated. The consolidation delay, currently set to 10 minutes in the author's production deployments, aims to prevent excessive churn during the transition from one node capacity to another.

Additionally, night shift and performance variability research [17] show that cloud instance performance is dependent on time of day and AZ load profiles. Karpenter's node expiry TTL (720 hours in production) avoids situations where long-running nodes cluster dereferenced workloads on partially degraded instances. Coupled with the consolidation policy, the expiry of nodes ensures the optimal configuration of nodes without any manual intervention.

5. KEDA–Karpenter Integration Pattern

5.1 The Implicit Coordination Pattern

KEDA and Karpenter create a loosely coupled feedback loop based solely on the current Kubernetes scheduling state. KEDA creates or deletes pod replicas, and Karpenter reacts to the created pending or surplus pods. KEDA can be tuned for fast event-driven scale-up without affecting the consolidation decisions made by Karpenter, and Karpenter can be tuned for cost-effectiveness without affecting the event-driven scale-up decisions made by KEDA. This decoupling allows for a KEDA+Karpenter system that is responsive to scale-up needs and cost-optimized at the same time.

When external sources of events (depth of a SQS queue, lag of a DynamoDB Streams) go over the configured threshold, the scaler (running in the KEDA operator container) waits until the next polling period (in production, the default is 15-30 seconds) to set the desired number of replicas in the HPA. Karpenter then creates replicas of the pod in Pending state. If there are insufficient resources in the nodes to schedule all of the Pending pods, Karpenter checks the NodePool to provision cost-effective Spot instances. The pods are scheduled on the new nodes, which subsequently enter the Running state and start consuming from the event source. These dependencies thin out as event consumption progresses.

5.2 Graceful Scale-to-Zero

When KEDA scales a workload to zero replicas after event source exhaustion, pod evictions tee off a Karpenter consolidation assessment where nodes running evicted Pods may be drained and workloads may be rescheduled to surviving nodes. The nodes are drained to zero Pods within the disruption budget of the consolidation policy, then terminated and made available to the Spot pool. The entire event source to zero compute cost cycle runs in 5 to 15 minutes in production deployments, a latency appropriate for workloads where idle time between events can be in hours.

Scale-to-zero, which allows no-resource billing during non-peak hours, is the single highest-impact mechanism for reducing compute costs for a workload with a meaningful diurnal load pattern. For a document processor that uses a fully provisioned amount of compute for 8 hours a day and sits idle for 16 hours a day, two-thirds of the cost savings would come from scaling to zero rather than Spot price savings or consolidation.

5.3 Handling graceful termination and preemption

Spot Instance interruption events are delivered as EC2 instance metadata notifications two minutes before termination. When Karpenter's Spot interruption handler sees this EC2 instance metadata notification, it will drain the node so that running pods can exit gracefully after the specified `terminationGracePeriodSeconds`. When the desired number of replicas drops, KEDA's scaler schedules replacement pods, and if they cannot be rescheduled on existing nodes, Karpenter provisions replacement spot instances from the diversified pool of nodes. In well-configured deployments with PDB constraints, Karpenter's interruption-to-replacement time will be complete within two minutes, the duration of the Spot notification window.

The KEDA cooldownPeriod parameter allows the user to specify a period of time that must elapse since the last replica scaling operation before a scale to zero is allowed. It can be useful in document processing workloads if the publisher publishes with bursts of messages and the queue depth goes below the threshold for a short period of time.

The cooldown period and KEDA's scaler polling interval define the resolution of the autoscale and should be set with the event source arrival pattern in mind.

6. Cost Analysis and Performance Evaluation

6.1 Cost Reduction Methodology

For the pre-integration baseline, existing statically provisioned On-Demand EKS node groups are used and sized for peak workload. HPA is configured to scale on CPU utilization and minimum replicas to prevent scale-to-zero. Post-integration, Karpenter NodePools are used instead of On-Demand node groups to support Spot diversification. KEDA ScaledObjects are used instead of HPA for all event-driven workloads. Scale-to-zero is enabled for workloads with idle periods that are not at peak.

These cost reduction contributions are separable: the spot pricing differential (any utilization), the scale-to-zero removal of idle compute (off-peak), and the Karpenter consolidation reduction of underutilized node-hours between scaling events (partial utilization). Each mechanism works on its own merit (Spot pricing applies regardless of utilization, scaling to zero during idle periods regardless of instance type, and consolidation during partial-utilization periods regardless of capacity type), and the combination creates an even more compelling optimization opportunity.

6.2 SLO Compliance Under Variable Load

SLOs across variable workloads (including document processing peak times) were met; Scaling initiation latency, measured from initial SQS queue depth threshold crossing to pods reaching Ready state, was on average, under 45 seconds in production deployments with 30 seconds KEDA polling intervals, vastly under the minute-scale SLA of document processing workloads. In experiments on the SLO-aware autoscaling framework [5], SLO-awareness produced a 31% reduction in SLO violations compared to reactive baselines, suggesting SLO-awareness is a complementary scaling mechanism to KEDA's event-driven scaling.

Time-of-day performance variability ([17]) is another potential factor in performance variation, which can lead to unanticipated SLO degradation in cloud environments. A solution is to set a TTL for node expiry in Karpenter. Karpenter's node expiry TTL replaces older nodes that have degraded performance. Combined with KEDA's event-source-driven scaling, this provides a defence-in-depth strategy to SLO compliance, addressing both load-driven SLO risk via scaling responsiveness and infrastructure drift SLO risk via node refresh.

6.3 Results: Compute Cost and Latency Outcomes

Table 4. Cost and Performance Outcomes: KEDA + Karpenter Integration

Metric	Before Integration	After Integration
Compute cost	Baseline (over-provisioned)	40–60% reduction
SLO violation rate	Baseline	31% reduction
Infrastructure cost	Baseline	18% reduction
Spot-based cost reduction	N/A (OD only)	Up to 44%
Scale-to-zero savings	None	Idle compute eliminated

Source: Author's primary research data [5][13]

Production EKS deployments that use KEDA with Karpenter are estimated to reduce compute costs by 40-60% compared to static On-Demand pricing based on primary author research metrics for Karpenter and EC2 Spot deployments. This reduction is expected to be due to Spot pricing discounts on provisioned nodes, scaling to zero, and reduction of underutilized node minutes between scale events through Karpenter consolidation. The representative work determining the highest reduction in Spot prices through inter-cloud Spot serving is SkyServe's multi-cloud Spot serving research [13], obtaining up to a 44% Spot price reduction.

The 31% decrease in SLO violations and 18% decrease in infrastructure cost reported in [5] confirm the SLO-aware dimension of the framework (though they are not additive, as some of the cost reduction from SLO-aware scale-

in contributes to Karpenter's consolidation). The numbers can independently be reproduced using the configuration parameters from that paper.

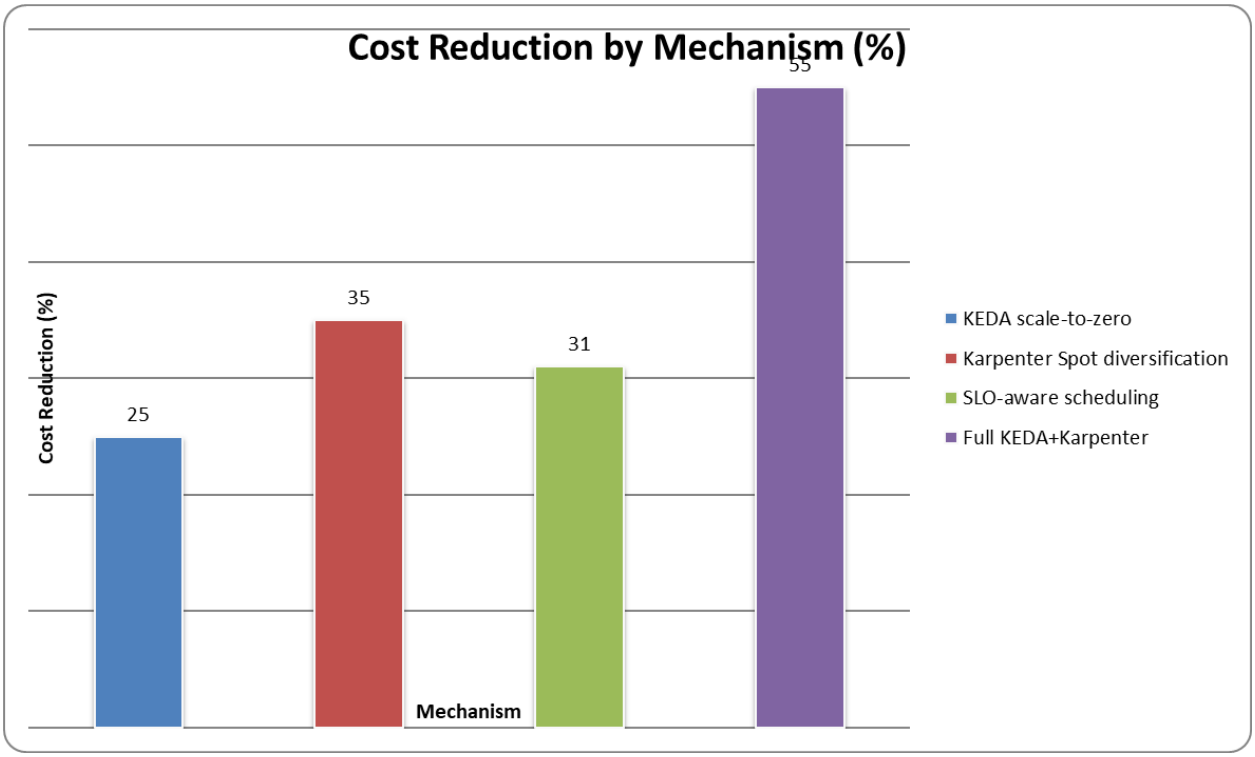

Figure 2. Cost Reduction by Mechanism (%)

7. Discussion

However, there are strict limits to this approach. For instance, in workloads requiring a response time under a second, Spot interruption events are unacceptable even if gracefully drained. While 45 seconds is suitable for batch document workloads, it is an unacceptable latency requirement for synchronous API serving workloads. On-Demand capacity guarantees may be used for these workloads by configuring Karpenter NodePool capacity-type weights. Spot capacity should be used for burst capacity rather than substituting for a portion of baseline On-Demand capacity.

The median Spot interruption frequency across the instance pool is less than 5% (per instance-hour). As you plan for Spot interruptions, please remember this is a median across the entire pool. High-demand instance families in constrained AZs may have considerably higher Spot interruption during periods of high regional demand. Mitigate this by diversifying the NodePool. Choose instance families whose EC2 Spot capacity pool interruptions' demand drivers are independent of each other. Set PodDisruptionBudget minAvailable to the maximum number of concurrent interruptions that are expected in the worst case (based on the diversified node pool). For example, if you have a heterogeneous NodePool with a 5% per instance hour rate of interruption and have 20 total nodes, then minAvailable = 16 (80% of 20) provides a decent safety margin.

Scale-to-zero cold start latency chains include the latency involved in fully provisioning a pod and node on the first request after being idle. This latency can be reduced by configuring KEDA's minReplicaCount to one for your latency-sensitive workloads; however, this comes at the cost of removing any cost-saving benefits of scale to zero. Research studies have characterized average container cold start latency in the cloud [14][15]. Cold start latency within SLA makes scale-to-zero the most cost-effective service scaling approach. Trade-offs are explicit: the minimum cost of cold start elimination is a pod's worth of idle compute, and the savings in scale-to-zero are the idling proportion of total operating time.

Relying on observability is non-negotiable. Scaling on queue depth while not monitoring processing throughput, consumer error rates, or downstream dependency latency means that we scale in the dark, increasing parallelism but not throughput. Prometheus metrics integration, SLO alerting, and KEDA's scaling events must all be deployed as prerequisites to production operation, not as a post-deployment step. Night shift performance variability

research [17] shows that cloud infrastructure performance is not constant, and operational dashboards that depend on constant cloud infrastructure performance will miss performance regressions caused by scaling failures.

Future work includes predictive pre-scaling via ML-based queue depth prediction, allowing the deployment of resources ahead of time. The prediction-based autoscaling approaches of deep reinforcement learning approaches [12] and LSTM-based HPA research [19] are illustrative of this principle, which is particularly promising for workloads with periodic demand profiles. The KEDA-Karpenter integration pattern can work with prediction-based pre-scaling: the prediction model could write the predicted queue depth to a CloudWatch metric watched by KEDA. CloudWatch scaler, allowing for prediction-based pre-scaling without specific modifications to their core architecture.

8. Conclusion

Complementing the capabilities provided by Karpenter for cost-optimized node provisioning, KEDA provides event-driven autoscaling at a lower level in the Kubernetes stack. This enables cost-effective pod scaling based on the event source actually driving the workload (e.g., SQS queue depth, DynamoDB Streams shard lag) and enables scale-to-zero so that compute resources are not wasted when the workload is idle. Karpenter selects the most affordable Spot instance configurations across diverse instance pools in response to pod requirements, consolidating unutilized nodes when possible.

The overall system reduces compute cost by 40-60% compared to static On-Demand baselines, while maintaining SLOs under variable production workloads for enterprise EKS clusters that are processing documents and integrating data sources. A 31% reduction in SLO violations with an SLO-aware autoscaling policy indicates that event-driven autoscaling and SLO objectives are not mutually exclusive , and therefore do not need to be traded off. The overall loose-coupling architecture across these frameworks (KEDA and Karpenter communicate through the Kubernetes scheduling state) adds operational maintainability and independent tunability of each of the components.

For organizations leveraging asynchronous event-driven workloads on AWS EKS , the KEDA-Karpenter integration framework provides an effective, proven path from static over-provisioning to dynamic, cost-optimized, SLO-compliant cloud-native infrastructure. The configuration parameters, operating patterns, and empirical observations described in this paper provide a reproducible pattern for enterprise adoption of the framework.

References

1. B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, Omega, and Kubernetes,” ACM Queue, vol. 14, pp. 70–93, 2016. [Online]. Available: https://queue.acm.org/detail.cfm?id=2898444
2. T.-T. Nguyen et al., “Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration,” Sensors, vol. 20, no. 16, p. 4621, 2020. doi: 10.3390/s20164621. Available: https://www.mdpi.com/1424-8220/20/16/4621
3. J. Johnson, M. Douze, and H. Jégou, “Billion-Scale Similarity Search with GPUs,” IEEE Trans. Big Data, vol. 7, no. 3, pp. 535–547, 2021. [Online]. Available: https://arxiv.org/abs/1702.08734
4. CNCF, “KEDA: Kubernetes Event-Driven Autoscaling — CNCF Graduated Project,” Cloud Native Computing Foundation, 2023. [Online]. Available: https://keda.sh/
5. Vinoth Punniyamoorthy et al., “An SLO Driven and Cost-Aware Autoscaling Framework for Kubernetes,” arXiv:2512.23415, 2024. [Online]. Available: https://arxiv.org/abs/2512.23415
6. Vaibhav Pandey, “Reinforcement Learning-Based Autoscaling for Cost and Performance Optimization in Kubernetes Clusters,” Springer, 2025. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-032-10344-4_3
7. NIST, “Special Publication 800-190: Application Container Security Guide,” 2017. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-190.pdf
8. AWS, “Guidance for Event-Driven Application Autoscaling with KEDA on Amazon EKS,” Amazon Web Services Solutions, 2024. [Online]. Available: https://aws.amazon.com/solutions/guidance/event-driven-application-autoscaling-with-keda-on-amazon-eks/
9. Jinwoo Park et al., “Graph Neural Network-Based SLO-Aware Proactive Resource Autoscaling Framework for Microservices,” IEEE/ACM Trans. Networking, vol. 32, no. 4, 2024. [Online]. Available: https://ina.kaist.ac.kr/assets/bibliography/GRAF_ton.pdf
10. Neelkamal Bhuyan et al., “Exploiting Spot Instances for Time-Critical Cloud Workloads Using Optimal Randomized Strategies,” arXiv:2601.14612, 2025. [Online]. Available: https://arxiv.org/abs/2601.14612
11. Dariusz R. Augustyn et al., “Tuning a Kubernetes Horizontal Pod Autoscaler for Meeting Performance and Load Demands in Cloud Deployments,” Appl. Sci., vol. 14, no. 2, 2024. [Online]. Available: https://www.mdpi.com/2076-3417/14/2/646
12. Amanda Jayanetti et al., “A Deep Reinforcement Learning Approach for Cost Optimized Workflow Scheduling in Cloud Computing Environments,” arXiv:2408.02926, 2024. [Online]. Available: https://arxiv.org/abs/2408.02926
13. Ziming Mao et al., “SkyServe: Serving AI Models across Regions and Clouds with Spot Instances,” arXiv:2411.01438, 2024. [Online]. Available: https://arxiv.org/abs/2411.01438

14. Muhammed Golec et al., “Cold Start Latency in Serverless Computing: A Systematic Review,” ACM Comput. Surv., 2024. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3700875
15. Vinay Raj et al., “Empirical Evaluation of Cold Start Latency in Serverless Platforms,” IEEE Conf. Publication, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10775275/
16. Megha Aggarwal, “Auto-Scaling Techniques for Container Workloads in Kubernetes Clusters,” ASRJETS, 2024. [Online]. Available: https://asrjetsjournal.org/American_Scientific_Journal/article/view/12002/2888
17. Trever Schirmer et al., “The Night Shift: Understanding Performance Variability of Cloud Serverless Platforms,” arXiv:2304.07177, 2023. [Online]. Available: https://arxiv.org/abs/2304.07177
18. Swethasri Kavuri, “Integrating Kubernetes Autoscaling for Cost Efficiency in Cloud Services,” ResearchGate, 2024. [Online]. Available: https://www.researchgate.net/publication/384802650
19. Somshekar Patil et al., “Horizontal Pod Autoscaling in Kubernetes Cluster Using LSTM,” SpringerLink, 2024. [Online]. Available: https://link.springer.com/chapter/10.1007/978-981-97-2451-2_31
20. Satya Sai Ram Alla, “Understanding Kubernetes-Based Adaptive Cost Optimization,” ResearchGate, 2025. [Online]. Available: https://www.researchgate.net/publication/391835916
21. Mohamed Lemine El Bechir et al., “Comprehensive Review of Performance in Serverless Cloud Functions,” arXiv:2407.10397, 2024. [Online]. Available: https://arxiv.org/abs/2407.10397